PHPUnit Manual

Versión latest

Sebastian Bergmann

15 de enero de 2019

Índice general

Edition for PHPUnit latest. Updated on 15 de enero de 2019.

Sebastian Bergmann

This work is licensed under the Creative Commons Attribution 3.0 Unported License.

Contents:

CAPÍTULO 1

Instalar PHPUnit

1.1 Requisitos

PHPUnit 7.0 requiere PHP 7.1. Se recomienda encarecidamente usar la última versión de PHP.

PHPUnit requiere las extensiones dom y json que normalmente están habilitadas por defecto.

Además, PHPUnit requiere las extensiones pcre, reflection, y spl. Estas extensiones estándares se habilitan por defecto y no se pueden desactivar sin parchear el sistema PHP construido y/o las fuentes en C.

La característica de reporte de cobertura de código requiere las extensiones Xdebug (2.5.0 o superior) y tokenizer. La generación de reportes en XML requiere la extensión xmlwriter.

1.2 PHP Archive (PHAR)

La manera más fácil de obtener PHPUnit es descargar un PHP Archive (PHAR) que contiene todas las dependencias requeridas (así como también algunas opcionales) de PHPUnit empaquetadas en un único archivo.

La extensión phar se necesita para usar PHP Archives (PHAR).

Si la extensión Suhosin está habilitada, necesitas permitir la ejecución de archivos PHAR en tu `php.ini`:

```
suhosin.executor.include.whitelist = phar
```

Para instalar globalmente el PHAR:

```
$  wget https://phar.phpunit.de/phpunit-7.0.phar
$  chmod +x phpunit-7.0.phar
$  sudo mv phpunit-7.0.phar /usr/local/bin/phpunit
$  phpunit --version
PHPUnit x.y.z by Sebastian Bergmann and contributors.
```

También podrías usar el archivo PHAR descargado directamente:

```
$  wget https://phar.phpunit.de/phpunit-7.0.phar
$  php phpunit-7.0.phar --version
PHPUnit x.y.z by Sebastian Bergmann and contributors.
```

1.2.1 Windows

Instalar globalmente el PHAR involucra el mismo procedimiento manual de instalar Composer en Windows:

1. Crea un directorio para binarios PHP; e.g., `C:\bin`
2. Añade ;C:bin a tu variable de entorno `PATH` (ayuda relacionada)
3. Descarga https://phar.phpunit.de/phpunit-7.0.phar y guarda el archivo como `C:\bin\phpunit.phar`
4. Abre una línea de comandos (e.g., presiona `WindowsR` » escribe cmd » `ENTER`)
5. Crea un script batch envolvente (resultando en `C:\bin\phpunit.cmd`):

```
C:\Users\username>  cd C:\bin
C:\bin>  echo @php "%~dp0phpunit.phar" %* > phpunit.cmd
C:\bin>  exit
```

6. Abre una nueva línea de comando y confirma que puedes ejecutar PHPUnit desde cualquier ruta:

```
C:\Users\username>  phpunit --version
PHPUnit x.y.z by Sebastian Bergmann and contributors.
```

Para los entornos shell Cygwin y/o MingW32 (e.g., TortoiseGit), podrías saltarte el paso 5. de arriba: simplemente guarda el archivo como `phpunit` (sin la extensión `.phar`) y hazla ejecutable usando `chmod 775 phpunit`.

1.2.2 Verificar Publicaciones de PHPUnit PHAR

Todas las publicaciones de código distribuído por el Proyecto PHPUnit están firmadas por el release manager para la versión. Firmas PGP y SHA1 están disponibles para verificación en phar.phpunit.de.

Los siguientes ejemplos detallan cómo funciona la verificación. Iniciemos por descargar el archivo `phpunit.phar` así como también su firma PGP desprendida `phpunit.phar.asc`:

```
wget https://phar.phpunit.de/phpunit.phar
wget https://phar.phpunit.de/phpunit.phar.asc
```

Queremos verificar el archivo PHAR de PHPUnit (`phpunit.phar`) contra su firma desprendida (`phpunit.phar.asc`):

```
gpg phpunit.phar.asc
gpg: Signature made Sat 19 Jul 2014 01:28:02 PM CEST using RSA key ID 6372C20A
gpg: Can't check signature: public key not found
```

No tenemos la clave pública del release manager (`6372C20A`) en nuestro sistema local. Para proceder con la verificación mecesitamos recuperar la clave pública del release manager desde un servidor de claves. Uno de esos servidores es `pgp.uni-mainz.de`. Los servidores de clave pública están enlazados juntos, así que deberías ser capaz de conectarte a cualquier servidor de claves.

```
gpg --keyserver pgp.uni-mainz.de --recv-keys 0x4AA394086372C20A
gpg: requesting key 6372C20A from hkp server pgp.uni-mainz.de
gpg: key 6372C20A: public key "Sebastian Bergmann <sb@sebastian-bergmann.de>" imported
```

```
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

Ahora hemos recibido una clave pública para una entidad conocida como «Sebastian Bergmann <sb@sebastian-bergmann.de>». Sin embargo, no tenemos una maner de verificar que esta clave fue creada por la persona conocida como Sebastian Bergmann. Pero, intentemos verificar la firma de la versión nuevamente.

```
gpg phpunit.phar.asc
gpg: Signature made Sat 19 Jul 2014 01:28:02 PM CEST using RSA key ID 6372C20A
gpg: Good signature from "Sebastian Bergmann <sb@sebastian-bergmann.de>"
gpg:                 aka "Sebastian Bergmann <sebastian@php.net>"
gpg:                 aka "Sebastian Bergmann <sebastian@thephp.cc>"
gpg:                 aka "Sebastian Bergmann <sebastian@phpunit.de>"
gpg:                 aka "Sebastian Bergmann <sebastian.bergmann@thephp.cc>"
gpg:                 aka "[jpeg image of size 40635]"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: D840 6D0D 8294 7747 2937  7831 4AA3 9408 6372 C20A
```

En este punto, la firma es buena pero no confiamos en esta clave. Una firma buena significa que el archivo no ha sido manipulado. Sin embargo, debido a la naturaleza de la criptografía de clave pública, también necesitas verificar que la clave `6372C20A` fue creada por el Sebastian Bergmann auténtico.

Cualquier atacante puede crear una clave pública y subirla a los servidores de clave pública. Ellos puede entonces crear versiones maliciosas firmadas con esta clave falsa. Entonces, si intentas verificar la firma de esta publicación corrupta, va a resultar exitosa debido a que la clave no era la clave «auténtica». Por lo tanto, necesitas validar la autenticidad de esta clave. Aún así, validar la autenticidad de una clave pública está fuera del ámbito de esta documentación.

Sería prudente crear un script de shell para manejar la instalación de PHPUnit que verifique la firma GnuPG antes de ejecutar tu test suite. Por ejemplo:

```
#!/usr/bin/env bash
clean=1 # Delete phpunit.phar after the tests are complete?
aftercmd="php phpunit.phar --bootstrap bootstrap.php src/tests"
gpg --fingerprint D8406D0D82947747293778314AA394086372C20A
if [ $? -ne 0 ]; then
    echo -e "\033[33mDownloading PGP Public Key...\033[0m"
    gpg --recv-keys D8406D0D82947747293778314AA394086372C20A
    # Sebastian Bergmann <sb@sebastian-bergmann.de>
    gpg --fingerprint D8406D0D82947747293778314AA394086372C20A
    if [ $? -ne 0 ]; then
        echo -e "\033[31mCould not download PGP public key for verification\033[0m"
        exit
    fi
fi

if [ "$clean" -eq 1 ]; then
    # Let's clean them up, if they exist
    if [ -f phpunit.phar ]; then
        rm -f phpunit.phar
    fi
    if [ -f phpunit.phar.asc ]; then
        rm -f phpunit.phar.asc
    fi
fi

# Let's grab the latest release and its signature
if [ ! -f phpunit.phar ]; then
```

```
    wget https://phar.phpunit.de/phpunit.phar
fi
if [ ! -f phpunit.phar.asc ]; then
    wget https://phar.phpunit.de/phpunit.phar.asc
fi

# Verify before running
gpg --verify phpunit.phar.asc phpunit.phar
if [ $? -eq 0 ]; then
    echo
    echo -e "\033[33mBegin Unit Testing\033[0m"
    # Run the testing suite
    `$after_cmd`
    # Cleanup
    if [ "$clean" -eq 1 ]; then
        echo -e "\033[32mCleaning Up!\033[0m"
        rm -f phpunit.phar
        rm -f phpunit.phar.asc
    fi
else
    echo
    chmod -x phpunit.phar
    mv phpunit.phar /tmp/bad-phpunit.phar
    mv phpunit.phar.asc /tmp/bad-phpunit.phar.asc
    echo -e "\033[31mSignature did not match! PHPUnit has been moved to /tmp/bad-
↪phpunit.phar\033[0m"
    exit 1
fi
```

1.3 Composer

Si usas Composer para gestionar las dependencias de tu proyecto, simplemente agrega una dependencia (en desarrollo) sobre `phpunit/phpunit` en el archivo `composer.json` del proyecto:

```
composer require --dev phpunit/phpunit ^7.0
```

1.4 Paquetes opcionales

Los siguientes paquetes opcionales están disponibles:

`PHP_Invoker`

Una clase utilitaria para invocar callables con un tiempo de espera. Este paquete se necesita para forzar tiempos de espera de prueba en el modo estricto.

Este paquete se incluye con la distribución PHAR de PHPUnit. Puede ser instalado via Composer usando el siguiente comando:

```
composer require --dev phpunit/php-invoker
```

`DbUnit`

Porte de DbUnit a PHP/PHPUnit para dar soporte a pruebas con interacción con base de datos.

Este paquete no está incluido en la distribución PHAR de PHPUnit. Puede ser instalado via Composer usando el siguiente comando

```
composer require --dev phpunit/dbunit
```

CAPÍTULO 2

Escribir pruebas con PHPUnit

El Example 2.1 muestra como podemos escribir pruebas usando PHPUnit, esta prueba ejecuta operaciones sobre un arreglo de PHP. El ejemplo presenta las convenciones básicas y los pasos para escribir pruebas con PHPUnit:

1. Las pruebas para una clase `Class` van dentro de una clase `ClassTest`.
2. `ClassTest` hereda (en la mayoría de los casos) de `PHPUnit\Framework\TestCase`.
3. Las pruebas son métodos públicos y toman como nombre `test*`.

   También, podemos usar la anotación de *docblock* `@test` en el método para marcarlo como un método de prueba.
4. Dentro de los métodos de prueba están los métodos de aserción, como `assertSame()` (ver *Aserciones*), que se usan para determinar o aseverar si el valor real coincide con el valor esperado.

Example 2.1: Probando operaciones sobre un arreglo con PHPUnit

```php
<?php
use PHPUnit\Framework\TestCase;

class StackTest extends TestCase
{
    public function testPushAndPop()
    {
        $stack = [];
        $this->assertSame(0, count($stack));

        array_push($stack, 'foo');
        $this->assertSame('foo', $stack[count($stack)-1]);
        $this->assertSame(1, count($stack));

        $this->assertSame('foo', array_pop($stack));
        $this->assertSame(0, count($stack));
    }
}
```

Martin Fowler:

> Siempre que estés intentando escribir algo en una sentencia `print` o una expresión depuradora, escribe en su lugar un prueba.

2.1 Dependencia de Pruebas

> *Adrian Kuhn et. al.*:
>
> Las pruebas unitarias son principalmente escritas como una buena práctica para ayudar a los desarrolladores a identificar y corregir errores, refactorizar el código y sirve como documentación de la parte del software bajo prueba. Para alcanzar estos beneficios, las pruebas unitarias deberían idealmente cubrir todos los posibles caminos en un programa. Una prueba unitaria cubre generalmente una ruta de acción específica de un método o función. Sin embargo un método de prueba no es necesariamente una entidad encapsulada e independiente. A veces existen dependencias implícitas entre los métodos de prueba, escondidas en el escenario de implementación de una prueba.

PHPUnit soporta la declaración de dependencias explícitas entre métodos de prueba. Estas dependencias no definen el orden en que los métodos de pruebas deben ser ejecutados pero ellas permiten retornar una instancia con los elementos necesarios para una prueba, pasándolos desde un productor hasta los consumidores.

- Un productor es un método de prueba que ofrece a la parte del software bajo prueba un valor de retorno.
- Un consumidor es un método de prueba que depende de uno o más productores y de sus valores de retorno.

El Example 2.2 muestra como usar la anotación `@depends` para expresar dependencias entre métodos de prueba.

Example 2.2: Usar la anotación `@depends` para expresar dependencias

```
<?php
use PHPUnit\Framework\TestCase;

class StackTest extends TestCase
{
    public function testEmpty()
    {
        $stack = [];
        $this->assertEmpty($stack);

        return $stack;
    }

    /**
     * @depends testEmpty
     */
    public function testPush(array $stack)
    {
        array_push($stack, 'foo');
        $this->assertSame('foo', $stack[count($stack)-1]);
        $this->assertNotEmpty($stack);

        return $stack;
    }

    /**
     * @depends testPush
     */
```

```
    public function testPop(array $stack)
    {
        $this->assertSame('foo', array_pop($stack));
        $this->assertEmpty($stack);
    }
}
```

En el ejemplo anterior, la primera prueba `testEmpty()` crea un nuevo arreglo y asevera que esta vacío. Como resultado la prueba regresa un elemento. La segunda prueba `testPush()` depende de `testEmpty()`, en este caso, se pasa como argumento el resultado de la prueba de la que depende. Finalmente, `testPop()` depende de `testPush()`.

Nota

Por defecto el valor de retorno brindado por un productor se pasa «como está» a su consumidor. Esto significa que cuando un productor regresa un objeto, se pasa una referencia del objeto al consumidor. En lugar de una referencia es posible pasar o (a) una copia (profunda) con la anotación `@depends clone` o (b) una clonación (superficial normal), basada en la palabra clave de PHP `clone`, con la anotación `@depends shallowClone`.

Para localizar defectos rápidamente, queremos concentrar nuestra atención sobre las pruebas fallidas relevantes. Por esta razón PHPUnit omite la ejecución de una prueba cuando la prueba de la que depende falla. Esto mejora la detección de los defectos, aprovechando las dependencias entre pruebas, como se muestra en Example 2.3.

Example 2.3: Aprovechar las dependencias entre pruebas

```
<?php
use PHPUnit\Framework\TestCase;

class DependencyFailureTest extends TestCase
{
    public function testOne()
    {
        $this->assertTrue(false);
    }

    /**
     * @depends testOne
     */
    public function testTwo()
    {
    }
}
```

```
$ phpunit --verbose DependencyFailureTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

FS

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) DependencyFailureTest::testOne
Failed asserting that false is true.

/home/sb/DependencyFailureTest.php:6
```

```
There was 1 skipped test:

1) DependencyFailureTest::testTwo
This test depends on "DependencyFailureTest::testOne" to pass.

FAILURES!
Tests: 1, Assertions: 1, Failures: 1, Skipped: 1.
```

Una prueba puede tener más de una anotación `@depends`. PHPUnit no cambia el orden en que las pruebas son ejecutadas, es necesario que las dependencias de una prueba puedan ser encontradas antes de que la prueba sea ejecutada.

Una prueba que tiene más de una anotación `@depends` obtendrá como primer argumento el resultado del primer productor, como segundo argumento el resultado del segundo productor y así sucesivamente. Ver Example 2.4

Example 2.4: Prueba con multiples dependencias

```php
<?php
use PHPUnit\Framework\TestCase;

class MultipleDependenciesTest extends TestCase
{
    public function testProducerFirst()
    {
        $this->assertTrue(true);
        return 'first';
    }

    public function testProducerSecond()
    {
        $this->assertTrue(true);
        return 'second';
    }

    /**
     * @depends testProducerFirst
     * @depends testProducerSecond
     */
     public function testConsumer($a, $b)
     {
         $this->assertSame('first', $a);
         $this->assertSame('second', $b);
     }
}
```

```
$ phpunit --verbose MultipleDependenciesTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

...

Time: 0 seconds, Memory: 3.25Mb

OK (3 tests, 3 assertions)
```

2.2 Proveedores de Datos

Un método de prueba puede aceptar argumentos arbitrarios. Estos argumentos son provistos por un método proveedor de datos (`additionProvider()` en el Example 2.5). El método proveedor de datos que queremos usar se especifica con la anotación `@dataProvider`.

Un método proveedor de datos debe ser `public` y retornar una arreglo de arreglos o un objeto que implementa la interfaz `Iterator` que produce un arreglo en cada paso de la iteración. Para cada arreglo que es parte de la colección se llama al método de prueba y el contenido del arreglo constituye sus argumentos.

Example 2.5: Usar un proveedor de datos que regresa un arreglo de arreglos

```
<?php
use PHPUnit\Framework\TestCase;

class DataTest extends TestCase
{
    /**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected)
    {
        $this->assertSame($expected, $a + $b);
    }

    public function additionProvider()
    {
        return [
            [0, 0, 0],
            [0, 1, 1],
            [1, 0, 1],
            [1, 1, 3]
        ];
    }
}
```

```
$ phpunit DataTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) DataTest::testAdd with data set #3 (1, 1, 3)
Failed asserting that 2 is identical to 3.

/home/sb/DataTest.php:9

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
```

Cuando se usa un gran número de datos es útil colocar una cadena de caracteres como llave en lugar de la numeración por defecto. La salida sera más verbosa y contendrá el nombre del conjunto de datos que hizo fallar la prueba.

Example 2.6: Usar un proveedor de datos con un conjunto de datos etiquetado

```
<?php
use PHPUnit\Framework\TestCase;

class DataTest extends TestCase
{
    /**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected)
    {
        $this->assertSame($expected, $a + $b);
    }

    public function additionProvider()
    {
        return [
            'adding zeros'  => [0, 0, 0],
            'zero plus one' => [0, 1, 1],
            'one plus zero' => [1, 0, 1],
            'one plus one'  => [1, 1, 3]
        ];
    }
}
```

```
$ phpunit DataTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) DataTest::testAdd with data set "one plus one" (1, 1, 3)
Failed asserting that 2 is identical to 3.

/home/sb/DataTest.php:9

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
```

Example 2.7: Usar un proveedor de datos que regresa un objeto Iterador

```
<?php
use PHPUnit\Framework\TestCase;

require 'CsvFileIterator.php';

class DataTest extends TestCase
{
    /**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected)
    {
```

```
        $this->assertSame($expected, $a + $b);
    }

    public function additionProvider()
    {
        return new CsvFileIterator('data.csv');
    }
}
```

```
$ phpunit DataTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) DataTest::testAdd with data set #3 ('1', '1', '3')
Failed asserting that 2 is identical to 3.

/home/sb/DataTest.php:11

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
```

Example 2.8: La clase CsvFileIterator

```
<?php
use PHPUnit\Framework\TestCase;

class CsvFileIterator implements Iterator
{
    protected $file;
    protected $key = 0;
    protected $current;

    public function __construct($file)
    {
        $this->file = fopen($file, 'r');
    }

    public function __destruct()
    {
        fclose($this->file);
    }

    public function rewind()
    {
        rewind($this->file);
        $this->current = fgetcsv($this->file);
        $this->key = 0;
    }

    public function valid()
    {
        return !feof($this->file);
```

```
    }

    public function key()
    {
        return $this->key;
    }

    public function current()
    {
        return $this->current;
    }

    public function next()
    {
        $this->current = fgetcsv($this->file);
        $this->key++;
    }
}
```

Cuando un prueba recibe una entrada tanto desde un método proveedor de datos `@dataProvider` como desde una o más pruebas de las que depende (`@depends`), los argumentos del proveedor de datos regresarán antes que los argumentos de las pruebas de las que depende. Los argumentos que se toman de las pruebas que se marcan como dependencias serán los mismo para cada conjunto de datos. Ver Example 2.9

Example 2.9: Combinación de @depends y @dataProvider en una misma prueba

```
<?php
use PHPUnit\Framework\TestCase;

class DependencyAndDataProviderComboTest extends TestCase
{
    public function provider()
    {
        return [['provider1'], ['provider2']];
    }

    public function testProducerFirst()
    {
        $this->assertTrue(true);
        return 'first';
    }

    public function testProducerSecond()
    {
        $this->assertTrue(true);
        return 'second';
    }

    /**
     * @depends testProducerFirst
     * @depends testProducerSecond
     * @dataProvider provider
     */
    public function testConsumer()
    {
        $this->assertSame(
            ['provider1', 'first', 'second'],
            func_get_args()
```

```
        );
    }
}
```

```
$ phpunit --verbose DependencyAndDataProviderComboTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 3.50Mb

There was 1 failure:

1) DependencyAndDataProviderComboTest::testConsumer with data set #1 ('provider2')
Failed asserting that two arrays are identical.
--- Expected
+++ Actual
@@ @@
Array &0 (
-     0 => 'provider1'
+     0 => 'provider2'
      1 => 'first'
      2 => 'second'
)
/home/sb/DependencyAndDataProviderComboTest.php:32

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
```

Example 2.10: Usar multiples proveedores de datos para una sola prueba

```
<?php
use PHPUnit\Framework\TestCase;

class DataTest extends TestCase
{
    /**
     * @dataProvider additionWithNonNegativeNumbersProvider
     * @dataProvider additionWithNegativeNumbersProvider
     */
    public function testAdd($a, $b, $expected)
    {
        $this->assertSame($expected, $a + $b);
    }

    public function additionWithNonNegativeNumbersProvider()
    {
        return [
            [0, 1, 1],
            [1, 0, 1],
            [1, 1, 3]
        ];
    }

    public function additionWithNegativeNumbersProvider()
    {
        return [
```

```
            [-1, 1, 0],
            [-1, -1, -2],
            [1, -1, 0]
        ];
    }
}
```

```
$ phpunit DataTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

..F...                                                              6 / 6 (100%)

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) DataTest::testAdd with data set #3 (1, 1, 3)
Failed asserting that 2 is identical to 3.

/home/sb/DataTest.php:12

FAILURES!
Tests: 6, Assertions: 6, Failures: 1.
```

Nota

Cuando una prueba depende de otra prueba que usa un proveedor de datos, la prueba dependiente será ejecutada solo cuando la prueba de la que depende es exitosa para al menos un elemento del conjunto de datos. El resultado de una prueba que usa proveedores de datos no puede ser inyectado dentro de una prueba dependiente.

Nota

Todos los proveedores de datos son ejecutados antes de la llamada al método estático `setUpBeforeClass()` y de la primera llamada al método `setUp()`. Por esta razón no es posible tener acceso a ninguna variable creada en estos métodos desde el proveedor de datos. Esto es necesario para que PHPUnit sea capaz de contar el numero total de pruebas.

2.3 Probar Excepciones

El Example 2.11 muestra como usar el método `expectException()` para probar si una excepción es lanzada por el código que se está probando.

Example 2.11: Usar el método expectException()

```
<?php
use PHPUnit\Framework\TestCase;

class ExceptionTest extends TestCase
{
    public function testException()
    {
        $this->expectException(InvalidArgumentException::class);
```

```
    }
}
?>
```

```
$ phpunit ExceptionTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ExceptionTest::testException
Failed asserting that exception of type "InvalidArgumentException" is thrown.

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Además del método `expectException()` existen los métodos `expectExceptionCode()`, `expectExceptionMessage()` y `expectExceptionMessageRegExp()` para establecer una predicción sobre las excepciones lanzadas por el código que se está probando.

Nota

Nótese que *expectExceptionMessage* asevera que el mensaje real (`$actual`) contiene el mensaje esperado (`$expected`) y no ejecuta una comparación exacta de cadenas de caracteres.

Alternativamente, podemos usar las anotaciones `@expectedException`, `@expectedExceptionCode`, `@expectedExceptionMessage` y `@expectedExceptionMessageRegExp` para establecer una predicción sobre las excepciones lanzadas por el código que se está probando. El Example 2.12 muestra un ejemplo.

Example 2.12: Usar la anotación @expectedException

```
<?php
use PHPUnit\Framework\TestCase;

class ExceptionTest extends TestCase
{
    /**
     * @expectedException InvalidArgumentException
     */
    public function testException()
    {
    }
}
```

```
$ phpunit ExceptionTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:
```

```
1) ExceptionTest::testException
Failed asserting that exception of type "InvalidArgumentException" is thrown.

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

2.4 Probar errores de PHP

Con la configuración por defecto de PHPUnit los errores, avisos y notificaciones de PHP que se disparan durante la ejecución de una prueba se convierten en excepciones. Usando estas excepciones se puede, por ejemplo, esperar que una prueba disparé un error de PHP, como se muestra en Example 2.13.

Nota

La configuración en tiempo de ejecución de PHP `error_reporting` puede limitar los errores que PHPUnit convertirá en excepciones. Si tenemos problemas con esta característica, debemos asegurarnos de que PHP no está configurado para eliminar los tipos de errores que estamos probando.

Example 2.13: Esperar un error de PHP usando @expectedException

```php
<?php
use PHPUnit\Framework\TestCase;

class ExpectedErrorTest extends TestCase
{
    /**
     * @expectedException PHPUnit\Framework\Error\Error
     */
    public function testFailingInclude()
    {
        include 'not_existing_file.php';
    }
}
```

```
$ phpunit -d error_reporting=2 ExpectedErrorTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

.

Time: 0 seconds, Memory: 5.25Mb

OK (1 test, 1 assertion)
```

`PHPUnit\Framework\Error\Notice` y `PHPUnit\Framework\Error\Warning` representan respectivamente notificaciones y avisos de PHP.

Nota

Cuando se prueban excepciones se debe ser tan específico como sea posible. Las pruebas de clases que son muy genéricas pueden ocasionar efectos secundarios indeseables. De la misma forma, probar la clase `Exception` con `@expectedException` o `expectException()` ya no se permite.

Cuando la prueba depende de funciones PHP que lanzan errores, como `fopen`, puede que sea útil usar la supresión de errores mientras se prueba. Esto permite revisar los valores retornados que sin la supresión de las notificaciones llevaría a un `PHPUnit\Framework\Error\Notice` de PHPUnit.

Example 2.14: Probar valores de retorno de un código que usa errores de PHP

```
<?php
use PHPUnit\Framework\TestCase;

class ErrorSuppressionTest extends TestCase
{
    public function testFileWriting()
    {
        $writer = new FileWriter;

        $this->assertFalse(@$writer->write('/is-not-writeable/file', 'stuff'));
    }
}

class FileWriter
{
    public function write($file, $content)
    {
        $file = fopen($file, 'w');

        if($file == false) {
            return false;
        }

        // ...
    }
}
```

```
$ phpunit ErrorSuppressionTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

.

Time: 1 seconds, Memory: 5.25Mb

OK (1 test, 1 assertion)
```

Sin la supresión de errores la prueba fallaría reportando `fopen(/is-not-writeable/file): failed to open stream: No such file or directory`.

2.5 Probar Salidas

A veces deseamos aseverar que la ejecución de un método genera la salida prevista (por ejemplo, con `echo` o `print`). La clase `PHPUnit\Framework\TestCase` usa la característica Funciones de Control de Salida de PHP para proporcionar la funcionalidad que se necesita para esta tarea.

El Example 2.15 muestra como usar el método `expectOutputString()` para establecer la salida prevista. Si la salida prevista no se genera, la prueba se contará como un fallo.

Example 2.15: Probar la salida de una función o método

```
<?php
use PHPUnit\Framework\TestCase;

class OutputTest extends TestCase
{
    public function testExpectFooActualFoo()
    {
        $this->expectOutputString('foo');
        print 'foo';
    }

    public function testExpectBarActualBaz()
    {
        $this->expectOutputString('bar');
        print 'baz';
    }
}
```

```
$ phpunit OutputTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

.F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) OutputTest::testExpectBarActualBaz
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'bar'
+'baz'

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

La Table 2.1 muestra los métodos que se pueden usar para probar la salida

Table 2.1: Métodos para probar la salida

Método	Propósito
`void expectOutputRegex(string $regularExpression)`	Define la salida esperada para coincidir con una `$regularExpression`.
`void expectOutputString(string $expectedString)`	Define la salida esperada igual a `$expectedString`.
`bool setOutputCallback(callable $callback)`	Define una función de retro llamada que se usa, por ejemplo, para normalizar la salida real.
`string getActualOutput()`	Trae la salida real.

Nota

Una prueba que emite una salida fallará de modo estricto.

2.6 Salida de Error

Siempre que una prueba falla PHPUnit intenta proveer la mayor cantidad de información de contexto que sea posible y que pueda ayudar a identificar el problema.

Example 2.16: Salida de error generada cuando la comparación entre arreglos falla

```
<?php
use PHPUnit\Framework\TestCase;

class ArrayDiffTest extends TestCase
{
    public function testEquality()
    {
        $this->assertSame(
            [1, 2,  3, 4, 5, 6],
            [1, 2, 33, 4, 5, 6]
        );
    }
}
```

```
$ phpunit ArrayDiffTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) ArrayDiffTest::testEquality
Failed asserting that two arrays are identical.
--- Expected
+++ Actual
@@ @@
 Array (
     0 => 1
     1 => 2
-    2 => 3
+    2 => 33
     3 => 4
     4 => 5
     5 => 6
 )

/home/sb/ArrayDiffTest.php:7

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

En este ejemplo solo uno de los valores del arreglo es diferente, los otros valores se muestran para dar el contexto y señalar donde ocurrió el error.

Cuando la salida generada es demasiado larga de leer, PHPUnit la separará y mostrará unas pocas lineas de información alrededor de cada diferencia.

Example 2.17: Salida de error cuando falla la comparación entre arreglos muy largos

```
<?php
use PHPUnit\Framework\TestCase;

class LongArrayDiffTest extends TestCase
{
    public function testEquality()
    {
        $this->assertSame(
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2,  3, 4, 5, 6],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 33, 4, 5, 6]
        );
    }
}
?>
```

```
$ phpunit LongArrayDiffTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) LongArrayDiffTest::testEquality
Failed asserting that two arrays are identical.
--- Expected
+++ Actual
@@ @@
     11 => 0
     12 => 1
     13 => 2
-    14 => 3
+    14 => 33
     15 => 4
     16 => 5
     17 => 6
 )

/home/sb/LongArrayDiffTest.php:7

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

2.6.1 Casos Límites

Cuando una comparación falla PHPUnit crea una representación textual de los valores de entrada y los compara. Debido a esta implementación un *diff* puede mostrar más problemas de los que realmente existen.

Esto solo sucede cuando se usa `assertEquals()` u otras funciones de comparación «débil» sobre arreglos u objetos.

Example 2.18: Caso límite al generar la diferencia cuando se usa comparación débil

```
<?php
use PHPUnit\Framework\TestCase;

class ArrayWeakComparisonTest extends TestCase
{
    public function testEquality()
    {
        $this->assertEquals(
            [1, 2, 3, 4, 5, 6],
            ['1', 2, 33, 4, 5, 6]
        );
    }
}
?>
```

```
$ phpunit ArrayWeakComparisonTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) ArrayWeakComparisonTest::testEquality
Failed asserting that two arrays are equal.
--- Expected
+++ Actual
@@ @@
 Array (
-    0 => 1
+    0 => '1'
     1 => 2
-    2 => 3
+    2 => 33
     3 => 4
     4 => 5
     5 => 6
 )

/home/sb/ArrayWeakComparisonTest.php:7

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

En este ejemplo la diferencia en el primer índice, entre `1` y `'1'`, se reporta aunque `assertEquals()` considera a estos valores como iguales.

CAPÍTULO 3

El Ejecutor de Pruebas desde Línea de Comandos

El ejecutor de pruebas desde línea de comandos de PHPUnit se puede invocar con el comando `phpunit`. El siguiente código muestra como ejecutar las pruebas con el ejecutor de pruebas desde línea de comandos:

```
$ phpunit ArrayTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

..

Time: 0 seconds

OK (2 tests, 2 assertions)
```

Cuando se ejecuta de la manera que se muestra arriba, el ejecutor de pruebas desde línea de comandos buscará el archivo fuente `ArrayTest.php` en la carpeta actual, lo cargará y esperará encontrar la clase de caso de prueba `ArrayTest`. Y luego se ejecutarán las pruebas de esta clase.

Para cada prueba que se ejecuta, la herramienta de línea de comandos imprime un carácter para indicar el progreso:

`.`
: Se imprime cuando la prueba es exitosa.

`F`
: Se imprime cuando una aserción falla mientras se ejecuta un método de prueba.

`E`
: Se imprime cuando ocurre un error mientras se ejecuta un método de prueba.

`R`
: Se imprime cuando una prueba se ha marcado como riesgosa (ver *Pruebas Riesgosas*).

`S`
: Se imprime cuando la prueba ha sido omitida (ver *Omisión y Pruebas Incompletas*).

`I`

Se imprime cuando la prueba se marca como incompleta o aún no implementada (ver *Omisión y Pruebas Incompletas*).

PHPUnit distingue entre *fallas* y *errores*. Una falla es una aserción de PHPUnit violada, como cuando la llamada a `assertSame()` falla. Un error es una excepción inesperada o un error de PHP. A veces esta distinción muestra ser útil porque los errores tienden a ser más fáciles de corregir que las fallas. Si tenemos una lista grande de problemas, es mejor resolver primero los errores y ver si queda alguna falla luego de que todos los errores están corregidos.

3.1 Opciones de la línea de comandos

Vamos a dar una ojeada a las opciones del ejecutor de pruebas desde línea de comandos, veamos el siguiente código:

```
$ phpunit --help
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

Usage: phpunit [options] UnitTest [UnitTest.php]
       phpunit [options] <directory>

Code Coverage Options:

  --coverage-clover <file>    Generate code coverage report in Clover XML format.
  --coverage-crap4j <file>    Generate code coverage report in Crap4J XML format.
  --coverage-html <dir>       Generate code coverage report in HTML format.
  --coverage-php <file>       Export PHP_CodeCoverage object to file.
  --coverage-text=<file>      Generate code coverage report in text format.
                              Default: Standard output.
  --coverage-xml <dir>        Generate code coverage report in PHPUnit XML format.
  --whitelist <dir>           Whitelist <dir> for code coverage analysis.
  --disable-coverage-ignore   Disable annotations for ignoring code coverage.

Logging Options:

  --log-junit <file>          Log test execution in JUnit XML format to file.
  --log-teamcity <file>       Log test execution in TeamCity format to file.
  --testdox-html <file>       Write agile documentation in HTML format to file.
  --testdox-text <file>       Write agile documentation in Text format to file.
  --testdox-xml <file>        Write agile documentation in XML format to file.
  --reverse-list              Print defects in reverse order

Test Selection Options:

  --filter <pattern>          Filter which tests to run.
  --testsuite <name,...>      Filter which testsuite to run.
  --group ...                 Only runs tests from the specified group(s).
  --exclude-group ...         Exclude tests from the specified group(s).
  --list-groups               List available test groups.
  --list-suites               List available test suites.
  --test-suffix ...           Only search for test in files with specified
                              suffix(es). Default: Test.php,.phpt

Test Execution Options:

  --dont-report-useless-tests Do not report tests that do not test anything.
  --strict-coverage           Be strict about @covers annotation usage.
  --strict-global-state       Be strict about changes to global state
  --disallow-test-output      Be strict about output during tests.
```

```
  --disallow-resource-usage   Be strict about resource usage during small tests.
  --enforce-time-limit        Enforce time limit based on test size.
  --disallow-todo-tests       Disallow @todo-annotated tests.

  --process-isolation         Run each test in a separate PHP process.
  --globals-backup            Backup and restore $GLOBALS for each test.
  --static-backup             Backup and restore static attributes for each test.

  --colors=<flag>             Use colors in output ("never", "auto" or "always").
  --columns <n>               Number of columns to use for progress output.
  --columns max               Use maximum number of columns for progress output.
  --stderr                    Write to STDERR instead of STDOUT.
  --stop-on-error             Stop execution upon first error.
  --stop-on-failure           Stop execution upon first error or failure.
  --stop-on-warning           Stop execution upon first warning.
  --stop-on-risky             Stop execution upon first risky test.
  --stop-on-skipped           Stop execution upon first skipped test.
  --stop-on-incomplete        Stop execution upon first incomplete test.
  --fail-on-warning           Treat tests with warnings as failures.
  --fail-on-risky             Treat risky tests as failures.
  -v|--verbose                Output more verbose information.
  --debug                     Display debugging information.

  --loader <loader>           TestSuiteLoader implementation to use.
  --repeat <times>            Runs the test(s) repeatedly.
  --teamcity                  Report test execution progress in TeamCity format.
  --testdox                   Report test execution progress in TestDox format.
  --testdox-group             Only include tests from the specified group(s).
  --testdox-exclude-group     Exclude tests from the specified group(s).
  --printer <printer>         TestListener implementation to use.

Configuration Options:

  --bootstrap <file>          A "bootstrap" PHP file that is run before the tests.
  -c|--configuration <file>   Read configuration from XML file.
  --no-configuration          Ignore default configuration file (phpunit.xml).
  --no-coverage               Ignore code coverage configuration.
  --no-extensions             Do not load PHPUnit extensions.
  --include-path <path(s)>    Prepend PHP's include_path with given path(s).
  -d key[=value]              Sets a php.ini value.
  --generate-configuration    Generate configuration file with suggested settings.

Miscellaneous Options:

  -h|--help                   Prints this usage information.
  --version                   Prints the version and exits.
  --atleast-version <min>     Checks that version is greater than min and exits.
```

`phpunit UnitTest`

> Ejecuta las pruebas que se encuentran en la clase `UnitTest`. Se espera que esta clase este declarada en el archivo fuente `UnitTest.php`.
>
> `UnitTest` debe ser una clase que hereda de `PHPUnit\Framework\TestCase` o una clase que provee un método `public static suite()` que regresa un objeto `PHPUnit\Framework\Test`, por ejemplo una instancia de la clase `PHPUnit_Framework_TestSuite`.

`phpunit UnitTest UnitTest.php`

Ejecuta las pruebas que están en la clase `UnitTest`. Se espera que esta clase este declarada en el archivo fuente que se especifica.

`--coverage-clover`

Genera un archivo de registro en formato XML con la información de la cobertura de código de las pruebas ejecutadas. Ver *Registro* para más detalles.

Nótese que esta funcionalidad solo está disponible cuando las extensiones tokenizer y Xdebug están instaladas.

`--coverage-crap4j`

Genera un reporte de cobertura de código en formato Crap4J. Para más detalles se puede ver *Análisis de Cobertura de Código*.

Nótese que esta funcionalidad solo está disponible cuando las extensiones tokenizer y Xdebug están instaladas.

`--coverage-html`

Genera un reporte de cobertura de código en formato HTML. Ver *Análisis de Cobertura de Código* para más detalles.

Nótese que esta funcionalidad solo está disponible cuando las extensiones tokenizer y Xdebug están instaladas.

`--coverage-php`

Genera un objeto PHP_CodeCoverage serializado con la información de cobertura de código.

Nótese que esta funcionalidad solo está disponible cuando las extensiones tokenizer y Xdebug están instaladas.

`--coverage-text`

Genera un archivo de registro o una salida en línea de comandos en un formato legible por humanos con la información de cobertura de código de las pruebas ejecutadas. Ver *Registro* para más detalles.

Nótese que esta funcionalidad solo está disponible cuando las extensiones tokenizer y Xdebug están instaladas.

`--log-junit`

Genera un archivo de sucesos en formato JUnit XML de las pruebas ejecutadas. Ver *Registro* para más detalles.

`--testdox-html` and `--testdox-text`

Genera un documento ágil en formato HTML o texto plano de las pruebas que se ejecutaron. Ver other-uses-for-tests para más detalles.

`--filter`

Solo ejecuta las pruebas cuyo nombre coincide con un patrón dado que está basado en una expresión regular. Si el patrón no se encierra entre delimitadores, PHPUnit cerrará el patrón dentro de delimitadores `/`.

El nombre de la prueba debe estar en uno de los siguientes formatos:

`TestNamespace\TestCaseClass::testMethod`

El formato de nombre para pruebas por defecto es equivalente a usar la constante mágica `__METHOD__` dentro el método de prueba.

`TestNamespace\TestCaseClass::testMethod with data set #0`

Cuando una prueba tiene un proveedor de datos, cada iteración sobre los datos trae el índice actual añadido al final del nombre por defecto de la prueba.

`TestNamespace\TestCaseClass::testMethod with data set "my named data"`

Cuando una prueba tiene un proveedor de datos que usa conjuntos etiquetados, cada iteración de los datos trae el nombre actual añadido al final del nombre por defecto de la prueba. Ver Example 3.1 para un ejemplo de conjunto de datos etiquetados.

Example 3.1: Conjunto de datos nombrados

```
<?php
use PHPUnit\Framework\TestCase;

namespace TestNamespace;

class TestCaseClass extends TestCase
{
    /**
     * @dataProvider provider
     */
    public function testMethod($data)
    {
        $this->assertTrue($data);
    }

    public function provider()
    {
        return [
            'my named data' => [true],
            'my data'       => [true]
        ];
    }
}
?>
```

`/path/to/my/test.phpt`

El nombre de la prueba para una prueba PHPT es la ruta en el sistema de archivos.

Revisar el ejemplo Example 3.2 para ver patrones de filtro validos.

Example 3.2: Ejemplos de patrones de filtro

```
--filter 'TestNamespace\\TestCaseClass::testMethod'
--filter 'TestNamespace\\TestCaseClass'
--filter TestNamespace
--filter TestCaseClase
--filter testMethod
--filter '/::testMethod .*"my named data"/'
--filter '/::testMethod .*#5$/'
--filter '/::testMethod .*#(5|6|7)$/'
```

Ver el Example 3.3 para algunos atajos adicionales que están disponibles para seleccionar proveedores de datos.

Example 3.3: Atajos para filtros

```
--filter 'testMethod#2'
--filter 'testMethod#2-4'
```

```
--filter '#2'
--filter '#2-4'
--filter 'testMethod@my named data'
--filter 'testMethod@my.*data'
--filter '@my named data'
--filter '@my.*data'
```

`--testsuite`

Solo ejecuta la suite de prueba cuyo nombre coincide con el patrón dado.

`--group`

Solo ejecuta las pruebas del o de los grupos especificados. Una prueba se puede marcar como perteneciente a un grupo usando la anotación `@group`.

Las anotaciones `@author` y `@ticket` son alias para `@group` que respectivamente permiten filtrar las pruebas con base en sus autores o en su ticket de identificación.

`--exclude-group`

Excluye las pruebas de un grupo o de los grupos especificados. Una prueba se puede marcar como perteneciente a un grupo usando la anotación `@group`.

`--list-groups`

Lista los grupos de pruebas disponibles.

`--test-suffix`

Solo busca los archivos de prueba con el o los sufijos especificados.

`--dont-report-useless-tests`

No reporta las pruebas que no prueban nada. Ver *Pruebas Riesgosas* para más detalles.

`--strict-coverage`

Es estricto con la cobertura de código involuntaria. Ver *Pruebas Riesgosas* para más detalles.

`--strict-global-state`

Es estricto con la manipulación del estado global. Ver *Pruebas Riesgosas* para más detalles.

`--disallow-test-output`

Es estricto sobre la salida durante las pruebas. Ver *Pruebas Riesgosas* para más detalles.

`--disallow-todo-tests`

No ejecuta las pruebas que tienen la anotación `@todo` es su bloque de documentación.

`--enforce-time-limit`

Impone un límite de tiempo basado en el tamaño de la prueba. Ver *Pruebas Riesgosas* para detalles.

`--process-isolation`

Ejecuta cada prueba en un proceso PHP separado.

`--no-globals-backup`

No respalda ni restaura la variable $GLOBALS. Ver *Estado Global* para más detalles.

`--static-backup`

Respalda y restaura los atributos estáticos de las clases definidas por el usuario. Ver *Estado Global* para más detalles.

`--colors`

Usa colores para la salida. En Windows, usamos ANSICON o ConEmu.

Existen tres valores posibles para esta opción:

- `never`: nunca mostrar colores en la salida. Este es el valor por defecto cuando se usa la opción `--colors`.
- `auto`: muestra los colores en la salida a menos que la terminal actual no soporte colores o si la salida se canaliza hacia un comando o si se redirige a un archivo.
- `always`: siempre muestra colores en la salida incluso cuando la terminal no soporta colores o cuando la salida se canaliza hacia un comando o se redirige a un archivo.

Cuando se usa `--colors` sin ningún valor se toma la opción `auto`.

`--columns`

Define el número de columnas que se usan para la salida que muestra el progreso. Si `max` se define con un valor, este número de columnas será el máximo de la terminal actual.

`--stderr`

Opcionalmente imprime en `STDERR` en lugar de `STDOUT`.

`--stop-on-error`

Se detiene la ejecución frente al primer error.

`--stop-on-failure`

Se detiene la ejecución frente al primer error o falla.

`--stop-on-risky`

Se detiene la ejecución frente a la primera prueba riesgosa.

`--stop-on-skipped`

Se detiene la ejecución frente a la primera prueba omitida.

`--stop-on-incomplete`

Se detiene la ejecución frente a la primera prueba incompleta.

`--verbose`

Hace a la información de salida más verbosa, por ejemplo, se muestran los nombres de las pruebas incompletas u omitidas.

`--debug`

Muestra la información de depuración en la salida, tal como el nombre de una prueba cuando comienza su ejecución.

`--loader`

Especifica la implementación de `PHPUnit\Runner\TestSuiteLoader` que se usa.

El cargador estándar de la suite de pruebas buscará el archivo fuente en la carpeta actual y en cada carpeta que se especifica en la directiva de configuración `include_path` de PHP. Un nombre de clase como `Project_Package_Class` se pone en correspondencia con un archivo fuente como `Project/Package/Class.php`.

`--repeat`

Ejecutar repetidamente la o las pruebas el número de veces especificado.

`--testdox`

Reporta el progreso de las pruebas en formato TestDox. Ver other-uses-for-tests para más detalles.

`--printer`

Especifica la impresora que se usa para generar el resultado. La clase impresora debe extender de `PHPUnit\Util\Printer` e implementar la interfaz `PHPUnit\Framework\TestListener`.

`--bootstrap`

Un archivo PHP «bootstrap» se ejecuta antes de las pruebas.

`--configuration`, `-c`

Lee la configuración desde el archivo XML. Ver *Archivo de Configuración XML* para más detalles.

Si `phpunit.xml` o `phpunit.xml.dist` (en este orden) existen en la carpeta actual de trabajo y `--configuration` *no* se usa, la configuración se leerá automáticamente de estos archivo.

Si se especifica una carpeta y si `phpunit.xml` o `phpunit.xml.dist` (en este orden) existen en la carpeta, la configuración se leerá automáticamente de estos archivos.

`--no-configuration`

Ignora los archivos `phpunit.xml` y `phpunit.xml.dist` de la capeta de trabajo actual.

`--include-path`

Añade al comienzo del `include_path` de PHP una o varias rutas dadas.

`-d`

Asigna una valor a la opción de configuración de PHP que se indica.

Nota

Nótese que desde la versión 4.8 las opciones se pueden colocar después de los argumentos.

3.2 TestDox

La característica TestDox de PHPUnit busca una clase de prueba y a todos sus métodos de prueba para los convierte desde «camel case» o «snake case» a oraciones: los métodos de prueba `testBalanceIsInitiallyZero()` or `test_balance_is_initially_zero()` son convertido en «Balance is initially zero». Si hay varios métodos de prueba cuyos nombres solo se diferencian en un sufijo de uno o más dígitos, como `testBalanceCannotBecomeNegative()` y `testBalanceCannotBecomeNegative2()`, las oraciones «Balance cannot become negative» aparecerá solo una vez, todo esto suponiendo que todas las pruebas tuvieron éxito.

Veamos que aspecto tiene un documento ágil generado para la clase `BankAccount`:

```
$ phpunit --testdox BankAccountTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

BankAccount
  Balance is initially zero
  Balance cannot become negative
```

Alternativamente, la documentación ágil se puede generar en formato HTML o texto plano y guardarlo en un archivo, esto se hace usando los argumentos `--testdox-html` y `--testdox-text`.

La Documentación Ágil se puede usar para documentar los supuestos que hacemos sobre los paquetes externos que usamos en nuestro proyecto. Cuando usamos un paquete externo estamos expuesto a que el paquete no se comporte de la forma esperada y, además, futuras versiones del paquete pueden cambiar de una manera sutil y romper nuestro código sin que nos demos cuenta de ello. Podemos prevenir estos riesgos escribiendo una prueba cada vez que hagamos una suposición. Si nuestra prueba es exitosa nuestra suposición es correcta. Si documentamos todas las suposiciones con pruebas, los lanzamientos futuros de los paquetes externos no serán causa de preocupación: si la prueba es exitosa, nuestro sistema debería seguir funcionando.

CAPÍTULO 4

Ambientes

Una de las partes que consume más tiempo cuando se hacen pruebas, es la parte en que se escribe el código que construye un mundo en un estado conocido desde donde comenzar y al que se debe regresar cuando la prueba se completa. Este estado conocido se llama *ambiente*, en ingles *fixture*, de la prueba.

En el ejemplo *Probando operaciones sobre un arreglo con PHPUnit*, el ambiente es simplemente un arreglo que se guarda en la variable `$stack`. Sin embargo, la mayoría del tiempo el ambiente será más complejo que un simple arreglo y la cantidad de código necesario para construirlo crecerá con su complejidad. El contenido real de la prueba se pierde entre el ruidoso trajinar de la construcción del ambiente. Este problema empeora aún más cuando escribimos varias pruebas con ambientes parecidos. Sin la ayuda de un framework de pruebas tendríamos que duplicar el código que construye el ambiente para cada prueba que se escribe.

PHPUnit puede compartir el código de configuración. Antes de que se ejecute un método de prueba un método modelo llamado `setUp()` se invoca. En el método `setUp()` es donde creamos el objeto contra el que probaremos. Una vez que el método de prueba terminó de ejecutarse, tanto si fue exitoso como fallido, otro método modelo llamado `tearDown()` se invoca. En el método `tearDown()` es con el que limpiamos los objetos contra los que hemos probado.

En el ejemplo *Usar la anotación @depends para expresar dependencias* usamos la relación productor-consumidor entre las pruebas para compartir un ambiente. Esto no siempre es deseable o incluso posible. En el Example 4.1 se muestra como podemos escribir las pruebas para la clase `StackTest` de una manera que no se reusa el ambiente sino el código que lo crea. Primero, declaramos una variable de instancia, `$stack`, que usaremos en lugar de la variable del método local. Luego delegamos al método `setUp()` la creación del `array` para el ambiente. Finalmente, removemos el código redundante de los métodos de prueba y usamos la variable de instancia introducida recientemente, `$this->stack`, en lugar de la variable del método local `$stack` en el método de aserción `assertSame()`.

Example 4.1: Usar setUp() para crear el ambiente para la clase StackTest

```
<?php
use PHPUnit\Framework\TestCase;

class StackTest extends TestCase
{
    protected $stack;

    protected function setUp()
```

```
    {
        $this->stack = [];
    }

    public function testEmpty()
    {
        $this->assertTrue(empty($this->stack));
    }

    public function testPush()
    {
        array_push($this->stack, 'foo');
        $this->assertSame('foo', $this->stack[count($this->stack)-1]);
        $this->assertFalse(empty($this->stack));
    }

    public function testPop()
    {
        array_push($this->stack, 'foo');
        $this->assertSame('foo', array_pop($this->stack));
        $this->assertTrue(empty($this->stack));
    }
}
```

Los métodos modelo `setUp()` y `tearDown()` se ejecutan una vez para cada método de prueba (y en instancias nuevas) de la clase de caso de prueba.

Además, los métodos modelo `setUpBeforeClass()` y `tearDownAfterClass()` se llaman respectivamente antes de que la primera prueba de la clase de caso de prueba se ejecute y después de que la última prueba de la clase de caso de prueba se haya ejecutado.

El ejemplo de abajo muestra todos los métodos modelo que están disponibles en la clase de caso de prueba.

Example 4.2: Ejemplo que muestra todos los métodos modelo disponibles

```
<?php
use PHPUnit\Framework\TestCase;

class TemplateMethodsTest extends TestCase
{
    public static function setUpBeforeClass()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    protected function setUp()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    protected function assertPreConditions()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    public function testOne()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
        $this->assertTrue(true);
```

```
    }

    public function testTwo()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
        $this->assertTrue(false);
    }

    protected function assertPostConditions()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    protected function tearDown()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    public static function tearDownAfterClass()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    protected function onNotSuccessfulTest(Exception $e)
    {
        fwrite(STDOUT, __METHOD__ . "\n");
        throw $e;
    }
}
?>
```

```
$ phpunit TemplateMethodsTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

TemplateMethodsTest::setUpBeforeClass
TemplateMethodsTest::setUp
TemplateMethodsTest::assertPreConditions
TemplateMethodsTest::testOne
TemplateMethodsTest::assertPostConditions
TemplateMethodsTest::tearDown
.TemplateMethodsTest::setUp
TemplateMethodsTest::assertPreConditions
TemplateMethodsTest::testTwo
TemplateMethodsTest::tearDown
TemplateMethodsTest::onNotSuccessfulTest
FTemplateMethodsTest::tearDownAfterClass

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) TemplateMethodsTest::testTwo
Failed asserting that <boolean:false> is true.
/home/sb/TemplateMethodsTest.php:30

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

4.1 Más setUp() que tearDown()

Los métodos `setUp()` y `tearDown()` son simétricos en la teoría pero no en la práctica. En la practica, solo necesitamos implementar `tearDown()` si asignamos recursos externos como archivos o *sockets* en el `setUp()`. Si nuestro `setUp()` solo crea objetos de PHP planos, podemos, en la mayoría de los casos, ignorar el `tearDown()`. Sin embargo, si hemos creado muchos objetos en el `setUp()`, podríamos querer usar el método `unset()` en el `tearDown()` para reiniciar las variables que están apuntando a esos objetos permitiendo que se puedan recolectar como basura. La recolección de basura de los objetos del caso de prueba no es predecible.

4.2 Variaciones

¿Que paso cuando tenemos dos pruebas con configuraciones ligeramente diferentes? Existen dos posibilidades:

- Si el código de configuración, `setUp()`, difiere solo ligeramente movemos el código que difiere del `setUp()` al método de prueba.
- Si realmente tenemos una configuración, `setUp()`, diferente necesitamos una clase de caso de prueba diferente. Creamos una nueva clase que tenga la diferencia de configuración.

4.3 Compartir el Ambiente

Existen algunas buenas razones para compartir los ambientes entre pruebas, pero en la mayoría de los casos la necesidad de compartir un ambiente entre pruebas radica en un problema de diseño no resuelto.

Un buen ejemplo de un ambiente que tiene sentido compartir a través de varias pruebas es la conexión a la base de datos: iniciamos sesión en la base de datos una vez y reusamos la conexión a la base de datos en lugar de crear una nueva conexión para cada prueba. Esto hace a las pruebas mucho más rápidas.

El Example 4.3 usa los métodos modelo `setUpBeforeClass()` y `tearDownAfterClass()` para, respectivamente, conectarse a la base de datos antes de la primera prueba del caso de prueba y para desconectarse de la base de datos después de la última prueba del caso de prueba.

Example 4.3: Compartir el ambiente de prueba entre el conjunto de pruebas

```
<?php
use PHPUnit\Framework\TestCase;

class DatabaseTest extends TestCase
{
    protected static $dbh;

    public static function setUpBeforeClass()
    {
        self::$dbh = new PDO('sqlite::memory:');
    }

    public static function tearDownAfterClass()
    {
        self::$dbh = null;
    }
}
```

Nunca es suficiente decir que compartir ambientes entre pruebas reduce el costo de las pruebas. El problema subyacente de diseño es que los objetos no están suficientemente desacoplados. Alcanzaremos mejores resultados resolviendo

el problema de diseño subyacente y luego escribiendo pruebas usando esbozos (ver *Dobles de Prueba*), en lugar de crear dependencias entre pruebas en tiempo de ejecución e ignorando la oportunidad de mejorar el diseño.

4.4 Estado Global

Es difícil probar código que usa instancias únicas de objetos (singletons). Lo mismo es verdad para el código que usa variables globales. Generalmente, el código que queremos probar está fuertemente acoplado con las variables globales y no podemos controlar su creación. Un problema adicional está en el hecho de que un cambio en la variable global para una prueba podría romper otra prueba.

En PHP las variables globales funcional de esta manera:

- Una variable global `$foo = 'bar';` se almacena como `$GLOBALS['foo'] = 'bar';`.
- La variable `$GLOBALS` es una variable *super-global*.
- Las variables super-globales son variables integradas que siempre están disponibles en todos los ámbitos.
- En el ámbito de una función o método, podemos acceder a la variable global `$foo` directamente accediendo a `$GLOBALS['foo']` o usando `global $foo;` para crear una variable local con una referencia a la variable global.

Además de las variables globales, los atributos estáticos de clases son también parte del estado global.

Antes de la versión 6, por defecto, PHPUnit ejecutaba las pruebas de una manera que los cambios de las variables globales y super-globales (`$GLOBALS`, `$_ENV`, `$_POST`, `$_GET`, `$_COOKIE`, `$_SERVER`, `$_FILES`, `$_REQUEST`) no afectaban a otras pruebas.

Desde la versión 6, PHPUnit no ejecuta por defecto esta operación de respaldo y restauración para las variables globales y super-globales. Esto se puede activar usando la opción `--globals-backup` o agregando `backupGlobals="true"` en el archivo de configuración XML.

Usando la opción `--static-backup` o agregando `backupStaticAttributes="true"` en el archivo de configuración, conseguimos que este aislamiento se puede extender a los atributos estáticos de clase.

Nota

Las operaciones de respaldo y restauración para todas las variables y atributos estáticos de clase usan `serialize()` y `unserialize()`.

Los objetos de algunas clases (por ejemplo, `PDO`) no se pueden serializar y la operación de respaldo se romperá cuando un objeto de este tipo se guarde, por ejemplo, en el arreglo `$GLOBALS`.

La anotación `@backupGlobals` sobre la que se discute en el apéndice *@backupGlobals* se puede usar para controlar las operaciones de respaldo y restauración de variables globales. Además, podemos proveer una lista negra de variables globales que deben ser excluidas de las operaciones de respaldo y recuperación de la siguiente manera:

```
class MyTest extends TestCase
{
    protected $backupGlobalsBlacklist = ['globalVariable'];

    // ...
}
```

Nota

Definir la propiedad `$backupGlobalsBlacklist` dentro del método `setUp()` no tiene efecto.

La anotación `@backupStaticAttributes` que se discute en el apéndice *@backupStaticAttributes* se puede usar para hacer un respaldo antes de cada prueba de todos los valores de las propiedades estáticas de todas las clases declaradas para restaurarlas después de la prueba.

Esta anotación procesa todas las clases, no solo la clase de prueba, que se declaran en el momento en que una prueba comienza. Esto solo aplica a las propiedades estáticas de la clase y no a las variables estáticas que están dentro de funciones.

Nota

La operación `@backupStaticAttributes` se ejecuta antes de un método de prueba, pero solo si está activada. Si el valor estático fue cambiado por la ejecución previa de una prueba que no tenía la `@backupStaticAttributes` activada, entonces el nuevo valor será respaldado y restaurado, y no el valor por defecto declarado originalmente. PHP no registra el valor por defecto declarada originalmente de ninguna variable estática.

Los mismo aplica para las propiedades estáticas de clases que fueron declaradas o cargadas recientemente dentro de una prueba. Después de su ejecución, las pruebas no pueden redefinir los valores por defecto declarados originalmente porque estos valores originales son desconocidos. Cualquier valor que sea colocado pasará a las otras pruebas.

Para las pruebas unitarias, se recomienda redefinir explícitamente los valores de la propiedades estáticas dentro de la prueba en nuestro código de `setUp()` (e idealmente también en el `tearDown()`, para no afectar las pruebas que se ejecuten posteriormente).

Podemos proveer una lista negra de los atributos estáticos que deben ser excluidos de las operaciones de respaldo y la restauración:

```
class MyTest extends TestCase
{
    protected $backupStaticAttributesBlacklist = [
        'className' => ['attributeName']
    ];

    // ...
}
```

Nota

Colocar la propiedad `$backupStaticAttributesBlacklist` dentro del método `setUp()` no tiene efecto.

CAPÍTULO 5

Organizar las Pruebas

Uno de los objetivos de PHPUnit es que las pruebas se puedan combinar: queremos ser capaces de ejecutar cualquier cantidad o combinación de pruebas, por ejemplo, todas las pruebas de un proyecto entero o las pruebas para todas las clases de un componente que es parte de un proyecto o solamente probar una clase.

PHPUnit soporta diferentes maneras de organizar y componer las pruebas dentro de una suite de prueba. Este capítulo muestra los enfoques que se usan comúnmente.

5.1 Componer una Suite de Prueba Usando el Sistema de Archivos

Probablemente la manera más fácil de componer una suite de prueba es tener todos los archivos fuente con los casos de prueba en una carpeta de prueba. PHPUnit puede descubrir automáticamente y ejecutar las pruebas recorriendo recursivamente el directorio de prueba.

Vamos a dar un vistazo a la suite de prueba de la biblioteca sebastianbergmann/money. Al ver la estructura de la carpeta del proyecto nos damos cuenta que las clases de casos de prueba en la carpeta `tests` son un espejo del paquete y de la estructura de clases del Sistema Bajo Prueba, en ingles *System Under Test (SUT)*, en la carpeta `src`:

```
src                                 tests
`-- Currency.php                    `-- CurrencyTest.php
`-- IntlFormatter.php               `-- IntlFormatterTest.php
`-- Money.php                       `-- MoneyTest.php
`-- autoload.php
```

Para ejecutar todas las pruebas de la biblioteca solo necesitamos apuntar el ejecutor de pruebas en línea de comandos a la carpeta donde están las pruebas:

```
$ phpunit --bootstrap src/autoload.php tests
PHPUnit |version|.0 by Sebastian Bergmann.

..............................

Time: 636 ms, Memory: 3.50Mb
```

```
OK (33 tests, 52 assertions)
```

Nota

Si apuntamos el ejecutor de pruebas en línea de comandos a la carpeta, él buscará los archivos `*Test.php`.

Para ejecutar solamente las pruebas que están en la clase de prueba `CurrencyTest` del archivo `tests/CurrencyTest.php`, podemos usar el siguiente comando:

```
$ phpunit --bootstrap src/autoload.php tests/CurrencyTest
PHPUnit |version|.0 by Sebastian Bergmann.

........

Time: 280 ms, Memory: 2.75Mb

OK (8 tests, 8 assertions)
```

Para un control de tipo granular sobre que pruebas ejecutar podemos usar la opción `--filter`:

```
$ phpunit --bootstrap src/autoload.php --filter␣
↪testObjectCanBeConstructedForValidConstructorArgument tests
PHPUnit |version|.0 by Sebastian Bergmann.

..

Time: 167 ms, Memory: 3.00Mb

OK (2 test, 2 assertions)
```

Nota

Una desventaja de este abordaje es que no tenemos control sobre el orden en que las pruebas se ejecutan. Esto puede causar problemas con las pruebas que tienen dependencias, ver *Dependencia de Pruebas*. En la siguiente sección veremos como podemos dar un orden de ejecución explicito usando un archivo de configuración XML.

5.2 Componer una Suite de Prueba con una Configuración XML

El archivo de configuración XML de PHPUnit (*Archivo de Configuración XML*) se puede usar, además, para componer una suite de prueba. El Example 5.1 muestra un archivo `phpunit.xml` que agregará todas las clases que se encuentran en los archivos `*Test.php` luego de recorrer recursivamente la carpeta `tests`.

Example 5.1: Componer una Suite de Prueba con Configuración XML

```
<phpunit bootstrap="src/autoload.php">
  <testsuites>
    <testsuite name="money">
      <directory>tests</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

Si `phpunit.xml` o `phpunit.xml.dist` (en este orden) existen en la carpeta de trabajo actual y la opción `--configuration` *no* se usa, la configuración será leída automáticamente del aquellos archivos.

Se puede hacer explícito el orden en que las pruebas se ejecutan:

Example 5.2: Componer una Suite de Prueba con Configuración XML

```
<phpunit bootstrap="src/autoload.php">
  <testsuites>
    <testsuite name="money">
      <file>tests/IntlFormatterTest.php</file>
      <file>tests/MoneyTest.php</file>
      <file>tests/CurrencyTest.php</file>
    </testsuite>
  </testsuites>
</phpunit>
```

CAPÍTULO 6

Pruebas Riesgosas

PHPUnit puede ejecutar revisiones adicionales, que documentamos más abajo, mientras se ejecutan las pruebas.

6.1 Pruebas Inútiles

PHPUnit es por defecto estricto con las pruebas que no prueban nada. Esta revisión se puede desactivar con la opción de línea de comandos `--dont-report-useless-tests` o colocando `beStrictAboutTestsThatDoNotTestAnything="false"` en el archivo de configuración XML de PHPUnit.

Una prueba que no ejecuta una aserción se marcará como riesgosa cuando esta revisión está habilitada. Expectativas sobre objetos falsos o anotaciones como @expectedException se cuentan como aserciones.

6.2 Cobertura Involuntaria de Código

PHPUnit puede ser estricto con el código cubierto involuntariamente. Esta revisión se puede activar usando la opción `--strict-coverage` en la línea de comandos o colocando `beStrictAboutCoversAnnotation="true"` en el archivo de configuración XML de PHPUnit.

Una prueba que tiene la anotación @covers y ejecuta código que no está marcado con las anotaciones @covers o @uses será marcada como riesgosa si esta revisión está activada.

6.3 Salida Durante la Ejecución de la Prueba

PHPUnit puede ser estricto con las salidas durante las pruebas. Esta revisión se puede habilitar desde la línea de comandos con la opción `--disallow-test-output` o colocando `beStrictAboutOutputDuringTests="true"` en el archivo de configuración XML de PHPUnit.

Una prueba que emite una salida, por ejemplo, invocando *print* en el código de prueba o en el código probado será marcada como riesgosa si esta revisión esta habilitada.

6.4 Tiempo de Espera de Ejecución de la Prueba

Se puede colocar un tiempo límite de ejecución para una prueba si el paquete `PHP_Invoker` está instalado y la extensión `pcntl` está disponible. La obligación de tener un tiempo límite se puede activar desde la línea de comandos con la opción `--enforce-time-limit` o colocando `enforceTimeLimit="true"` en el archivo de configuración XML de PHPUnit.

Una prueba con la anotación `@large` fallará si toma más de 60 segundos en ejecutarse. El tiempo de espera se puede configurar con el atributo `timeoutForLargeTests` en el archivo de configuración XML de PHPUnit.

Una prueba con la anotación `@medium` fallará si demora más de 10 segundos en ejecutarse. Este tiempo de espera se puede configurar con el atributo `timeoutForMediumTests` en el archivo de configuración XML de PHPUnit.

Una prueba con la anotación `@small` fallará si toma más de 1 segundo en ejecutarse. Este tiempo límite se puede configurar con el atributo `timeoutForSmallTests` en el archivo de configuración XML.

Nota

Las pruebas se deben marcar explícitamente con la anotación `@small`, `@medium` o `@large` para habilitar los límites de ejecución.

6.5 Manipulación del Estado Global

PHPUnit puede ser estricto con las pruebas que manipulan el estado global. Esta revisión se puede habilitar usando `--strict-global-state` desde la línea de comandos o colocando `beStrictAboutChangesToGlobalState="true"` en el archivo de configuración XML de PHPUnit.

CAPÍTULO 7

Omisión y Pruebas Incompletas

7.1 Pruebas Incompletas

Cuando estamos trabajando con una nueva clase para un caso de prueba, podríamos querer escribir un método de prueba vacío y de esta manera mantener una lista con las pruebas que debemos escribir, en este caso podríamos comenzar de la siguiente manera:

```
public function testSomething()
{
}
```

El problema con los métodos de prueba vacíos es que son interpretados como exitosos por el framework PHPUnit. Este error en la interpretación lleva a que el reporte de la prueba sea inútil, porque no podemos ver si la prueba fue realmente exitosa o solo no está implementada. Llamar a `$this->fail()` en un método de prueba no implementado no nos ayudará, pues, la prueba será interpretada como una falla. Lo que sería tan incorrecto como interpretar una prueba no implementada como exitosa.

Si pensamos en una luz verde para una prueba exitosa y una luz roja para una prueba fallida, necesitamos una luz adicional de color amarillo para marcar una prueba como incompleta o aún no implementada. `PHPUnit\Framework\IncompleteTest` es una interfaz de marcado para indicar un excepción que lanza el método de prueba cuando la prueba está incompleta o actualmente no está implementada. La clase `PHPUnit\Framework\IncompleteTestError` es la implementación estándar de esta interfaz.

El Example 7.1 muestra una clase para un caso de prueba, `SampleTest`, que contiene el método de prueba `testSomething()`. Mediante el llamado al método `markTestIncomplete()` (que lanza automáticamente una excepción `PHPUnit\Framework\IncompleteTestError`) en el método de prueba, conseguimos marcar la prueba como incompleta.

Example 7.1: Marcar una prueba como incompleta

```
<?php
use PHPUnit\Framework\TestCase;

class SampleTest extends TestCase
```

```
{
    public function testSomething()
    {
        // Optional: Test anything here, if you want.
        $this->assertTrue(true, 'This should already work.');

        // Stop here and mark this test as incomplete.
        $this->markTestIncomplete(
          'This test has not been implemented yet.'
        );
    }
}
```

Una prueba incompleta se denota con una `I` en la salida del ejecutor de pruebas en línea de comandos de PHPUnit, como se muestra en el siguiente ejemplo:

```
$ phpunit --verbose SampleTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

I

Time: 0 seconds, Memory: 3.95Mb

There was 1 incomplete test:

1) SampleTest::testSomething
This test has not been implemented yet.

/home/sb/SampleTest.php:12
OK, but incomplete or skipped tests!
Tests: 1, Assertions: 1, Incomplete: 1.
```

En la Table 7.1 se muestra la API para marcar las pruebas como incompletas.

Table 7.1: API para Pruebas Incompletas

Método	Propósito
`void markTestIncomplete()`	Marca la prueba actual como incompleta.
`void markTestIncomplete(string $message)`	Marca la prueba actual como incompleta usando un mensaje, `$message`, explicativo.

7.2 Omitir las Pruebas

No todas la pruebas se pueden ejecutar en cualquier entorno. Consideremos por ejemplo, una capa de abstracción de base de datos que tiene varios controladores para los diferentes sistemas de base de datos soportados. Las pruebas para un controlador MySQL se pueden ejecutar solamente, por supuesto, si un servidor MySQL está disponible.

El Example 7.2 muestra una clase para un caso de prueba, `DatabaseTest`, que contiene el método de prueba, `testConnection()`. En el método modelo `setUp()` de la clase de caso de prueba revisamos si la extensión MySQLi está disponible y usamos el método `markTestSkipped()` para saltar la prueba si no está disponible la extensión.

Example 7.2: Omitir la prueba

```
<?php
use PHPUnit\Framework\TestCase;

class DatabaseTest extends TestCase
{
    protected function setUp()
    {
        if (!extension_loaded('mysqli')) {
            $this->markTestSkipped(
              'The MySQLi extension is not available.'
            );
        }
    }

    public function testConnection()
    {
        // ...
    }
}
```

Una prueba que se omite se denota con una `S` en la salida del ejecutor de pruebas en línea de comandos de PHPUnit, como se muestra en el siguiente ejemplo:

```
$ phpunit --verbose DatabaseTest
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

S

Time: 0 seconds, Memory: 3.95Mb

There was 1 skipped test:

1) DatabaseTest::testConnection
The MySQLi extension is not available.

/home/sb/DatabaseTest.php:9
OK, but incomplete or skipped tests!
Tests: 1, Assertions: 0, Skipped: 1.
```

En la Table 7.2 se muestra la API para omitir pruebas.

Table 7.2: API para omitir pruebas

Método	Propósito
`void markTestSkipped()`	Marca la prueba actual como omitida.
`void markTestSkipped(string $message)`	Marca la prueba actual como omitida usando un mensaje, `$message`, de explicación.

7.3 Omitir pruebas usando @requires

A parte de los métodos de arriba es posible usar la anotación `@requires` para expresar precondiciones comunes para un caso de prueba.

Table 7.3: Posibles usos para @requires

Tipo	Posibles Valores	Ejemplos	Otros Ejemplos
`PHP`	Cualquier identificador de versión de PHP	@requires PHP 5.3.3	@requires PHP 7.1-dev
`PHPUnit`	Cualquier identificador de versión de PHPUnit	@requires PHPUnit 3.6.3	@requires PHPUnit 4.6
`OS`	Una expresión regular que coincida con PHP_OS	@requires OS Linux	@requires OS WIN32|WINNT
`OSFAMILY`	Cualquier valor de Familia de SO	@requires OSFAMILY Solaris	@requires OSFAMILY Windows
`function`	Cualquier parámetro valido para function_exists	@requires function imap_open	@requires function ReflectionMethod::setAccessible
`extension`	Cualquier nombre de extensión junto con un identificador de versión opcional	@requires extension mysqli	@requires extension redis 2.2.0

Example 7.3: Omitir casos de pruebas usando @requires

```
<?php
use PHPUnit\Framework\TestCase;

/**
 * @requires extension mysqli
 */
class DatabaseTest extends TestCase
{
    /**
     * @requires PHP 5.3
     */
    public function testConnection()
    {
        // Test requires the mysqli extension and PHP >= 5.3
    }

    // ... All other tests require the mysqli extension
}
```

Si estamos usando una sintaxis que no copila con una determinada versión de PHP es conveniente revisar la dependencia de versiones en la configuración xml que se incluye en *Suites de Pruebas*.

CAPÍTULO 8

Probar Bases de Datos

Muchos ejemplos de pruebas unitarias para principiantes y usuarios intermedios de cualquier lenguaje de programación sugieren que es absolutamente fácil probar la lógica de la aplicación usando pruebas simples. Pero para aplicaciones centradas en base de datos esto está muy lejos de la realidad. Basta comenzar a usar WordPress, TYPO3 o Symfony con Doctrine o Propel, por ejemplo, para experimentar rápidamente considerables problemas con PHPUnit porque la base de datos está estrechamente vinculada con estas bibliotecas.

Nota

Debemos asegurarnos de tener instalada la extensión de PHP `pdo` y la extensión específica para la base de datos, como `pdo_mysql`. De lo contrario los ejemplos que se muestran abajo no van a funcionar.

Probablemente conoces el escenario, por tu trabajo diario y por tus proyectos, en el que queremos colocar en acción nuestras habilidades (nuevas o no) con PHPUnit y quedamos atrapados con uno de los siguientes problemas:

1. El método que queremos probar ejecuta una operación JOIN muy grande y usa los datos para calcular algunos resultados importantes.
2. Nuestra lógica de negocio ejecuta una mezcla de sentencias SELECT, INSERT, UPDATE y DELETE
3. Necesitamos asignar datos de prueba a (muy probablemente) más de dos tablas con el objeto de tener datos iniciales razonables para los métodos que queremos probar.

La extensión DbUnit simplifica considerablemente la configuración de la base de datos para las pruebas y permite verificar el contenido de una base de datos después de ejecutar una serie de operaciones. La instalación de la extensión DbUnit es fácil y está documentada en *Paquetes opcionales*.

8.1 Proveedores Soportados para las Pruebas de la Base de Datos

Actualmente DbUnit soporta MySQL, PostgreSQL, Oracle y SQLite. Si se integra Zend Framework o Doctrine 2 DbUnit puede acceder a otros sistemas de base de datos como IBM DB2 o Microsoft SQL Server.

8.2 Dificultades al Probar Bases de Datos

Existe una buena razón para que todos los ejemplos de pruebas unitarias no incluyan interacciones con la base de datos: esos tipos de pruebas son tanto complejas de configurar como de mantener. Cuando se prueba una base de datos necesitamos tener cuidado con las siguientes variables:

- El esquema y las tablas de la base de datos.
- Insertar las filas necesarias para las pruebas en esas tablas.
- Verificar el estado de la base de datos después de ejecutar las pruebas.
- Limpiar la base de datos para cada nueva prueba.

Como muchas APIs de base de datos, como PDO, MySQLi o ICI8, son incomodas de usar y de escritura verbosa, hacer estos pasos manualmente es definitivamente una pesadilla.

Probar código debe ser lo más corto y preciso posible por varias razones:

- No queremos modificar una considerable cantidad de código de prueba por pequeños cambios en el código de producción.
- Queremos ser capaces de leer y entender el código de prueba fácilmente, incluso meses después de escribirlo.

Además, no debemos olvidar que la base de datos es esencialmente una variable global de entrada para nuestro código. Dos pruebas de nuestro conjunto de pruebas se pueden ejecutar contra la misma base de datos, posiblemente usando los datos varias veces. Fallas en la primera prueba puede fácilmente afectar el resultado de las siguientes pruebas, haciendo muy difícil nuestra experiencia con las pruebas. La limpieza, uno de los pasos mencionados anteriormente, tiene mucha importancia para resolver el problema de: «una base de datos como una entrada global».

Cuando se prueban bases de datos DbUnit ayuda a simplificar todos estos problemas de una manera elegante.

En lo que PHPUnit no puede ayudar es con el hecho de que las pruebas a bases de datos son mucho más lentas comparadas con las pruebas que no usan bases de datos. Dependiendo del cantidad de las interacciones con la base de datos las pruebas podrían demorar una cantidad de tiempo considerable. Sin embargo, si la cantidad de datos usados para cada prueba se mantiene pequeña y probamos tanto código como sea posible con pruebas que no interactuaran con la base de datos, podemos fácilmente terminar en menos de un minuto un conjunto grande de pruebas.

El conjunto de pruebas del proyecto Doctrine 2", por ejemplo, tiene actualmente cerca de 1000 pruebas donde casi la mitad de ellas acceden a la base de datos, su ejecución contra una base de datos MySQL en una computadora de escritorio estándar es de 15 segundos.

8.3 Las cuatro etapas de las pruebas con base de datos

En su libro sobre Patrones de Prueba xUnit Gerard Meszaros señala las cuatro etapas de una prueba unitaria:

1. Configurar el ambiente (fixture).
2. Ejercitar el Sistema Bajo Prueba.
3. Verificar los resultados.
4. Desmontar (Teardown).

   ¿Que es un ambiente (Fixture)?

   Un ambiente describe el estado inicial en que está nuestra aplicación y su base de datos cuando se ejecuta la prueba.

Probar la base de datos exige que se utilice al menos una configuración (setup) y un desmontaje (teardown) para limpiar y escribir los datos iniciales dentro de las tablas. La extensión de base de datos tiene una buena razón para revertir los cuatro etapas de una prueba de base de datos, el siguiente flujo de trabajo se ejecuta en cada prueba:

8.3.1 1. Limpiar la Base de Datos

Como siempre existe una primera prueba que se ejecuta contra la base de datos y no sabemos exactamente si ya existen datos en las tablas. PHPUnit va a ejecutar un TRUNCATE contra todas las tablas especificadas para redefinir sus estados a vacío.

8.3.2 2. Configurar el ambiente

PHPUnit va a iterar sobre todas las filas de ambientación especificadas y las insertará en sus respectivas tablas.

8.3.3 3–5. Ejecutar la Prueba, Verificar el resultado y Desmontar

Después de redefinir la base de datos y cargarla con su estado inicial la verdadera prueba es ejecutada por PHPUnit. Esta parte de la prueba no necesita de ningún conocimiento sobre la Extensión de Base de Datos por lo que podemos seguir y probar cualquier cosa que queramos con nuestro código.

Nuestras pruebas usan una aserción especial llamada `assertDataSetsEqual()` para fines de verificación, sin embargo es totalmente opcional. Esta característica se explica en la sección «Aserciones en Bases de Datos».

8.4 Configuración de un Caso de Prueba de una Base de Datos

Generalmente cuando se usa PHPUnit nuestros casos de prueba extenderán de la clase `PHPUnit\Framework\TestCase` de la siguiente manera:

```
<?php
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    public function testCalculate()
    {
        $this->assertSame(2, 1 + 1);
    }
}
```

Si queremos probar código que trabaja con la Extensión de Base de Datos la configuración es un poco más compleja y debemos extender de una *TestCase* abstracta diferente e implementando los métodos abstractos `getConnection()` y `getDataSet()`:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class MyGuestbookTest extends TestCase
{
    use TestCaseTrait;

    /**
```

```
     * @return PHPUnit\DbUnit\Database\Connection
     */
    public function getConnection()
    {
        $pdo = new PDO('sqlite::memory:');
        return $this->createDefaultDBConnection($pdo, ':memory:');
    }

    /**
     * @return PHPUnit\DbUnit\DataSet\IDataSet
     */
    public function getDataSet()
    {
        return $this->createFlatXMLDataSet(dirname(__FILE__).'/_files/guestbook-seed.
↪xml');
    }
}
```

8.4.1 Implementando getConnection()

Para permitir que las funcionalidades de limpieza y carga de datos funcione, la Extensión de Base de Datos de PHPUnit necesita acceder a una conexión de base de datos abstraída del proveedor a través de la biblioteca PDO. Es importante notar que nuestra aplicación no necesita estar basada en PDO para usar la extensión de base de datos de PHPUnit, la conexión solo se usa para limpiar y aplicar la configuración inicial o ambiente.

En el ejemplo anterior creamos una conexión Sqlite en memoria que pasamos al método `createDefaultDBConnection` y que, además, envuelve la instancia PDO, como segundo parámetro pasamos el nombre de la base de datos. Todo esto se hace usando una capa de abstracción muy simple para conexiones de base de datos del tipo `PHPUnit\DbUnit\Database\Connection`.

La sección «Usar la API de Conexión de Base de Datos» explica la API de esta interfaz y como podemos hacer el mejor uso de ella.

8.4.2 Implementando getDataSet()

El método `getDataSet()` define como debe ser el estado inicial de la base de datos antes de que cada prueba sea ejecutada. El estado de la base de datos es abstraído a través de los concepto *DataSet* (Conjunto de Datos) y *DataTable* (Tabla de Datos) que son representados por las interfaces `PHPUnit\DbUnit\DataSet\IDataSet` y `PHPUnit\DbUnit\DataSet\IDataTable`. La siguiente sección describe en detalles como estos conceptos trabajan y que beneficios trae su uso en las pruebas de base de datos.

Para la implementación solo necesitamos saber que el método `getDataSet()` se llama una vez durante el `setUp()` para traer el conjunto de datos de la ambientación para luego insertarlos en la base de datos. En el ejemplo estamos usando el método de fábrica `createFlatXMLDataSet($filename)` que representa un conjunto de datos por medio de una representación XML.

8.4.3 ¿Y que pasa con el Esquema de Base de Datos (DDL)?

PHPUnit asume que el esquema de base de datos con todas sus tablas, lanzadores, secuencias y vistas está creado antes de que la prueba sea ejecutada. Esto significa que como desarrolladores debemos asegurar que la base de datos está correctamente configurada antes de ejecutar el paquete de pruebas.

Existen varias maneras para alcanzar esta pre-condición de las pruebas con bases de datos.

1. Si estamos usando una base de datos persistente (no SQLite en memoria) podemos, con facilidad, configurar la base de datos una solo vez con herramientas como phpMyAdmin para MySQL y usar la misma base de datos en cada ejecución de una prueba.
2. Si usamos bibliotecas como Doctrine 2 o Propel podemos usar sus APIs para crear una sola vez el esquema de base de datos que necesitamos antes de ejecutar las pruebas. Podemos usar las capacidades de Configuración y Bootstrap de PHPUnit's para ejecutar ese código cada vez que nuestras pruebas sean ejecutadas.

8.4.4 Consejo: Usemos nuestro propio Caso Abstracto de Prueba de Base de Datos

Del ejemplo previo de implementación podemos ver fácilmente que el método `getConnection()` es muy estático y podría usarse en diferentes casos de prueba de base de datos. Además, para mantener el buen rendimiento de nuestras pruebas y la carga sobre la base de datos baja podemos refactorizar un poco el código para obtener un caso de prueba abstracto genérico para nuestra aplicación, y que aún nos permita especificar datos de ambientación diferentes para cada caso de prueba:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

abstract class MyApp_Tests_DatabaseTestCase extends TestCase
{
    use TestCaseTrait;

    // only instantiate pdo once for test clean-up/fixture load
    static private $pdo = null;

    // only instantiate PHPUnit\DbUnit\Database\Connection once per test
    private $conn = null;

    final public function getConnection()
    {
        if ($this->conn === null) {
            if (self::$pdo == null) {
                self::$pdo = new PDO('sqlite::memory:');
            }
            $this->conn = $this->createDefaultDBConnection(self::$pdo, ':memory:');
        }

        return $this->conn;
    }
}
?>
```

Este código tiene la conexión a la base de datos incrustada en la conexión PDO. PHPUnit tiene otra característica importante que podría hacer este caso de prueba incluso más genérico. Si usamos la Configuración XML podemos hacer la conexión a base de datos configurable para cada ejecución de una prueba. Primero vamos a crear el archivo «phpunit.xml» en nuestra carpeta tests/ de la aplicación para que se vea de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8" ?>
<phpunit>
    <php>
        <var name="DB_DSN" value="mysql:dbname=myguestbook;host=localhost" />
        <var name="DB_USER" value="user" />
        <var name="DB_PASSWD" value="passwd" />
        <var name="DB_DBNAME" value="myguestbook" />
```

```
    </php>
</phpunit>
```

Ahora podemos modificar nuestro caso de prueba de la siguiente manera:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

abstract class Generic_Tests_DatabaseTestCase extends TestCase
{
    use TestCaseTrait;

    // only instantiate pdo once for test clean-up/fixture load
    static private $pdo = null;

    // only instantiate PHPUnit\DbUnit\Database\Connection once per test
    private $conn = null;

    final public function getConnection()
    {
        if ($this->conn === null) {
            if (self::$pdo == null) {
                self::$pdo = new PDO( $GLOBALS['DB_DSN'], $GLOBALS['DB_USER'],
↪$GLOBALS['DB_PASSWD'] );
            }
            $this->conn = $this->createDefaultDBConnection(self::$pdo, $GLOBALS['DB_
↪DBNAME']);
        }

        return $this->conn;
    }
}
?>
```

Ahora podemos ejecutar el paquete de pruebas de base de datos usando diferentes configuraciones desde la interfaz de línea de comandos:

```
$ user@desktop> phpunit --configuration developer-a.xml MyTests/
$ user@desktop> phpunit --configuration developer-b.xml MyTests/
```

La posibilidad de ejecutar las pruebas de base de datos contra diferentes base de datos es muy importante si estamos programando en una computadora de desarrollo. Si varios desarrolladores ejecutan las pruebas de base de datos contra la misma conexión de base de datos podemos fácilmente experimentar fallas en las pruebas a causa de una condición de carrera.

8.5 Entendiendo los Conjuntos de Datos y las Tablas de Datos

Los Conjuntos de Datos (DataSets) y las Tablas de Datos (DataTables) son conceptos centrales de la Extensión de Base de Datos de PHPUnit. Deberías intentar entender estos conceptos simples para dominar las pruebas de base de datos con PHPUnit. Los DataSet y los DataTable son una capa de abstracción en torno a las tablas, filas y columnas de nuestra base de datos. Una API simple que oculta el contenido subyacente de la base de datos en una estructura de tipo objeto y que además permite implementar otras fuentes de datos que no son una base de datos.

Esta abstracción es necesaria para comparar el contenido actual de la base de datos con el contenido esperado. Por

ejemplo, las expectativas se pueden representar con archivos XML, YAML o CSV; o con un arreglo PHP. Las interfaces DataSet (Conjunto de Datos) y DataTable (Tablas de Datos) permiten comparar estas fuentes conceptualmente diferentes, emulando el almacenamiento en una base de datos relacional y con un abordaje sistemáticamente similar.

El flujo de trabajo para las aserciones de base de datos en nuestras pruebas consiste en tres simples pasos:

- Especificar una o más tablas de nuestra base de datos por el nombre de la tabla (conjunto de datos real).
- Especificar el conjunto de datos esperado en nuestro formato preferido (YAML, XML, . . .).
- Afirmar que ambas representaciones de datos son iguales.

Las aserciones no son los únicos casos de uso para los Conjuntos de Datos y las Tablas de Datos de la Extensión para Bases de Datos de PHPUnit. Como se muestra en la sección anterior ellos además describen el contenido inicial de una base de datos. Estamos obligados a definir un conjunto de datos para la ambientación del Caso de Prueba de la Base de Datos, que luego se usa para:

- Borrar todas las filas de las tablas especificadas en el conjunto de datos.
- Escribir todas las columnas en las tablas de datos de la base de datos.

8.5.1 Implementaciones Disponibles

Existen tres diferentes tipos de Conjunto de Datos y Tablas de Datos:

- Conjuntos de Datos y Tablas de Datos basadas en archivo.
- Conjuntos de Datos y Tablas de Datos basadas en consulta.
- Filtros y Composición de Conjuntos de Datos y Tablas de Datos.

Los Conjuntos de Datos y Tablas de Datos basados en archivo se usan generalmente para la ambientación inicial y para describir los estados esperados de la base de datos.

Conjunto de Datos con XML Plano

El conjunto de datos más común es el llamado XML Plano. Es un formato XML muy simple donde una etiqueta dentro del nodo raíz `<dataset>` representa exactamente una fila en la base de datos. El nombre de la etiqueta es igual a la tabla en la que se inserta la fila y un atributo representa una columna. Un ejemplo para una aplicación simple de libro de visitas podría ser el siguiente:

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" created="2010-04-24 17:15:23"␣
↪/>
    <guestbook id="2" content="I like it!" user="nancy" created="2010-04-26 12:14:20"␣
↪/>
</dataset>
```

Es obviamente fácil de escribir. Aquí `<guestbook>` es el nombre de la tabla donde dos filas se insertan cada una con cuatro columnas «id», «content», «user» y «created» con sus respectivos valores.

Sin embargo, esta simplicidad tiene un costo.

En el ejemplo anterior no es obvio como podríamos especificar una tabla vacía. Podemos insertar una etiqueta sin atributos con el nombre de la tabla vacía. Un archivo XML plano para una tabla que representa un libro de visitas vacío podría ser el siguiente:

```
<?xml version="1.0" ?>
<dataset>
    <guestbook />
</dataset>
```

La manipulación de valores NULL con XML plano es tedioso. En casi todas las bases de datos, excepto Oracle, un valor NULL es diferente a una cadena de caracteres con valor vacío, además, es algo que resulta difícil de describir en un formato XML plano. Podemos representar valores NULL omitiendo el atributo que especifica la fila. Si nuestro libro de visitas permite entradas anónimas representadas por el valor NULL en la columna de usuario, un estado hipotético de la tabla que representa el libro de visitas podría ser el siguiente:

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" created="2010-04-24 17:15:23"␣
↪/>
    <guestbook id="2" content="I like it!" created="2010-04-26 12:14:20" />
</dataset>
```

En este caso la segunda entrada es publicada anónimamente. Sin embargo, esto deja un problema serio para el reconocimiento de columnas. Durante las aserciones de igualdad de conjunto de datos, cada conjunto de datos debe especificar que columnas pertenecen a una tabla. Si un atributo es NULL para todas las filas de la tabla de datos, ¿como podría saber la Extensión de Base de Datos que la columna debe ser parte de la tabla?

El conjunto de datos en XML plano hace una suposición crucial, a partir del atributo de la primera fila definida de una tabla, se definen las columnas de esta tabla. En el ejemplo anterior esto significaría que «id», «content», «user» y «created» son columnas de la tabla del libro de visitas. Para la segunda fila donde «user» no se define, un NULL se insertaría en la base de datos.

Cuando la primera entrada del libro de visitas se borra solo las columnas «id», «content» y «created» serán columnas de la tabla del libro de visitas porque «user» no está especificado.

Para usar eficazmente el conjunto de datos con un XML Plano cuando los valores NULL son importantes la primera columna de cada tabla no debe contener ningún valor nulo y solo las filas siguientes pueden omitir atributos. Esto puede ser molesto ya que el orden de las filas es un importante factor para las aserciones de base de datos.

Además, si especificamos solo un subconjunto de columnas de la tabla en el conjunto de datos XML Plano todos los valores omitidos se colocarán en su valor por defecto. Esto puede traer errores si una de las columnas omitidas se define como «NOT NULL DEFAULT NULL».

En conclusión podemos decir que los conjuntos de datos en XML Plano solo se pueden usar si no necesitamos valores NULL.

Podemos crear una instancia del conjunto de datos XML plano dentro de nuestro Caso de Prueba de Base de Datos llamando al método `createFlatXmlDataSet($filename)`:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class MyTestCase extends TestCase
{
    use TestCaseTrait;

    public function getDataSet()
    {
        return $this->createFlatXmlDataSet('myFlatXmlFixture.xml');
    }
}
?>
```

Conjunto de Datos XML

Existe otro conjunto de datos XML más estructurado, que es un poco más detallado de escribir pero evita los problemas con los valores NULL del conjunto de datos XML Plano. Dentro de la raíz `<dataset>` podemos especificar etiquetas `<table>`, `<column>`, `<row>`, `<value>` y `<null />`. Un conjunto de datos equivalente al libro de visitas con XML plano definido anteriormente se ve de la siguiente manera:

```
<?xml version="1.0" ?>
<dataset>
    <table name="guestbook">
        <column>id</column>
        <column>content</column>
        <column>user</column>
        <column>created</column>
        <row>
            <value>1</value>
            <value>Hello buddy!</value>
            <value>joe</value>
            <value>2010-04-24 17:15:23</value>
        </row>
        <row>
            <value>2</value>
            <value>I like it!</value>
            <null />
            <value>2010-04-26 12:14:20</value>
        </row>
    </table>
</dataset>
```

Toda `<table>` definida tiene un nombre y requiere una definición de todas las columnas con sus nombres. Esta puede contener cero o cualquier número de elementos `<row>` anidados. No definir un elemento `<row>` significa que la tabla está vacía. Las etiquetas `<value>` y `<null />` se deben especificar en el orden en que se especificaron los elementos `<column>`. La etiqueta `<null />` obviamente significa que el valor es NULL.

Podemos crear una instancia del conjunto de datos XML desde dentro de nuestro Caso de Prueba de Base de Datos llamando al método `createXmlDataSet($filename)`:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class MyTestCase extends TestCase
{
    use TestCaseTrait;

    public function getDataSet()
    {
        return $this->createXMLDataSet('myXmlFixture.xml');
    }
}
?>
```

Conjunto de Datos XML MySQL

Este nuevo formato es específico para servidor de base de datos MySQL. El soporte para este formato se añadió en PHPUnit 3.5. Los archivos en este formato se pueden generar usando la herramiento mysqldump. A diferencia de los conjuntos de datos CSV, que `mysqldump` también soporta, un solo archivo en este formato pueden contener datos de varias tablas. Podemos crear un archivo en este formato invocando el comando `mysqldump` de la siguiente forma:

```
$ mysqldump --xml -t -u [username] --password=[password] [database] > /path/to/file.
↪xml
```

Este archivo se puede usar en nuestro Caso de Prueba de Base de Datos llamando al método `createMySQLXMLDataSet($filename)`:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class MyTestCase extends TestCase
{
    use TestCaseTrait;

    public function getDataSet()
    {
        return $this->createMySQLXMLDataSet('/path/to/file.xml');
    }
}
?>
```

Conjunto de Datos YAML

También podemos usar el conjunto de datos YAML para el ejemplo del libro de visitas:

```
guestbook:
  -
    id: 1
    content: "Hello buddy!"
    user: "joe"
    created: 2010-04-24 17:15:23
  -
    id: 2
    content: "I like it!"
    user:
    created: 2010-04-26 12:14:20
```

Este formato es simple, conveniente Y soluciona el problema de los valores NULL que tendría un conjunto de datos similar representado con un XML Plano. Un valor NULL en YAML es solamente el nombre de la columna sin ningún valor especificado. Una cadena de caracteres vacía se especifica como `column1: ""`.

El conjunto de datos YAML no tiene actualmente un método fábrica para el Caso de Prueba de Base de Datos, por lo que debemos instanciarlo manualmente:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;
use PHPUnit\DbUnit\DataSet\YamlDataSet;

class YamlGuestbookTest extends TestCase
```

```
{
    use TestCaseTrait;

    protected function getDataSet()
    {
        return new YamlDataSet(dirname(__FILE__)."/_files/guestbook.yml");
    }
}
?>
```

Conjunto de Datos CSV

Otro conjunto de datos basado en archivo se basa en archivos CSV. Cada tabla del conjunto de datos se representa con un archivo CSV. Para nuestro ejemplo de libro de visitas podemos definir el archivo guestbook-table.csv:

```
id,content,user,created
1,"Hello buddy!","joe","2010-04-24 17:15:23"
2,"I like it!","nancy","2010-04-26 12:14:20"
```

A pesar de ser muy conveniente para editar con Excel o LibreOffice, no podemos especificar valores NULL con el conjunto de datos CSV. Una columna vacía llevará a que el valor vacío por defecto de la base de datos se inserte en la columna.

Podemos crear un Conjunto de Datos CSV llamando:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;
use PHPUnit\DbUnit\DataSet\CsvDataSet;

class CsvGuestbookTest extends TestCase
{
    use TestCaseTrait;

    protected function getDataSet()
    {
        $dataSet = new CsvDataSet();
        $dataSet->addTable('guestbook', dirname(__FILE__)."/_files/guestbook.csv");
        return $dataSet;
    }
}
?>
```

Conjunto de Datos en Arreglo

No existe (aún) un Conjunto de Datos basado en Arreglos en la Extensión de Base de Datos de PHPUnit, pero podemos implementar con facilidad uno propio. Nuestro ejemplo de libro de visitas se vera de la siguiente manera:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class ArrayGuestbookTest extends TestCase
{
    use TestCaseTrait;
```

```
    protected function getDataSet()
    {
        return new MyApp_DbUnit_ArrayDataSet(
            [
                'guestbook' => [
                    [
                        'id' => 1,
                        'content' => 'Hello buddy!',
                        'user' => 'joe',
                        'created' => '2010-04-24 17:15:23'
                    ],
                    [
                        'id' => 2,
                        'content' => 'I like it!',
                        'user' => null,
                        'created' => '2010-04-26 12:14:20'
                    ],
                ],
            ]
        );
    }
}
?>
```

Un Conjunto de Datos con PHP tiene obvias ventajas sobre todo los otros conjuntos de datos basados en archivos:

- Los Arreglos de PHP pueden obviamente manejar valores `NULL`.
- No necesitaremos agregar archivos para las aserciones y podemos especificarlas directamente en el Caso de Prueba.

Para este conjunto de datos; como para los anteriores Conjuntos de Datos XML Plano, CSV y YAML; las llaves de la primera fila especificada definen los nombres de las columnas de la tabla, en el caso anterior estas serán ser «id», «content», «user» y «created».

La implementación para este Conjunto de Datos basado en Arreglos es simple y directo:

```
<?php

use PHPUnit\DbUnit\DataSet\AbstractDataSet;
use PHPUnit\DbUnit\DataSet\DefaultTableMetaData;
use PHPUnit\DbUnit\DataSet\DefaultTable;
use PHPUnit\DbUnit\DataSet\DefaultTableIterator;

class MyApp_DbUnit_ArrayDataSet extends AbstractDataSet
{
    /**
     * @var array
     */
    protected $tables = [];

    /**
     * @param array $data
     */
    public function __construct(array $data)
    {
        foreach ($data as $tableName => $rows) {
            $columns = [];
```

```
            if (isset($rows[0])) {
                $columns = array_keys($rows[0]);
            }

            $metaData = new DefaultTableMetaData($tableName, $columns);
            $table = new DefaultTable($metaData);

            foreach ($rows as $row) {
                $table->addRow($row);
            }
            $this->tables[$tableName] = $table;
        }
    }

    protected function createIterator($reverse = false)
    {
        return new DefaultTableIterator($this->tables, $reverse);
    }

    public function getTable($tableName)
    {
        if (!isset($this->tables[$tableName])) {
            throw new InvalidArgumentException("$tableName is not a table in the␣
↪current database.");
        }

        return $this->tables[$tableName];
    }
}
?>
```

Conjunto de Datos basados en Consultas SQL

Para las aserciones de base de datos no solo necesitamos conjuntos de datos basados en archivo sino también conjuntos de datos basados en Consultas SQL que contengan el contenido real de la base de datos. En este caso es que los Conjuntos de Datos basados en Consultas toman protagonismo:

```
<?php
$ds = new PHPUnit\DbUnit\DataSet\QueryDataSet($this->getConnection());
$ds->addTable('guestbook');
?>
```

Agregar una tabla usando su nombre es una manera implícita de definir los datos de un tabla, que es equivalente a la siguiente consulta:

```
<?php
$ds = new PHPUnit\DbUnit\DataSet\QueryDataSet($this->getConnection());
$ds->addTable('guestbook', 'SELECT * FROM guestbook');
?>
```

Podemos usar este método para especificar consultas arbitrarias sobre nuestras tablas y, por ejemplo, restringir las filas, las columnas o agregar clausulas `ORDER BY`:

```
<?php
$ds = new PHPUnit\DbUnit\DataSet\QueryDataSet($this->getConnection());
```

```
$ds->addTable('guestbook', 'SELECT id, content FROM guestbook ORDER BY created DESC');
?>
```

En la sección sobre Aserciones de Base de Datos mostraremos algunos detalles más sobre como hacer uso del Conjunto de Datos en base a Consultas.

Conjunto de Datos basados en Base de Datos

Luego de acceder a una Conexión de Prueba podemos automáticamente crear un Conjunto de Datos que consiste en todas las tablas, con su contenido, de la base de datos que se especifica como segundo parámetro del método Fábrica de Conexiones.

Podemos crear un conjunto de datos para toda la base de datos como se muestra en `testGuestbook()` o restringirlo a un conjunto de tablas, especificándolas por su nombre con una lista blanca como se muestra en el método `testFilteredGuestbook()`.

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class MySqlGuestbookTest extends TestCase
{
    use TestCaseTrait;

    /**
     * @return PHPUnit\DbUnit\Database\Connection
     */
    public function getConnection()
    {
        $database = 'my_database';
        $user = 'my_user';
        $password = 'my_password';
        $pdo = new PDO('mysql:...', $user, $password);
        return $this->createDefaultDBConnection($pdo, $database);
    }

    public function testGuestbook()
    {
        $dataSet = $this->getConnection()->createDataSet();
        // ...
    }

    public function testFilteredGuestbook()
    {
        $tableNames = ['guestbook'];
        $dataSet = $this->getConnection()->createDataSet($tableNames);
        // ...
    }
}
?>
```

Replacement DataSet

Hemos hablado sobre los problemas con los valores nulos cuando usamos Conjuntos de Datos como XML Plano y CSV, pero hay una solución algo complicada para ambos casos que nos permite ponerlos a trabajar con valores NULL.

El Conjunto de Datos de Reemplazo es un decorador para un conjunto de datos existente que permite remplazar el valor de cualquier columna del conjunto de datos por otro valor. Para tener nuestro ejemplo de libro de visitas trabajando con valores NULL modificamos el archivo de la siguiente manera:

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" created="2010-04-24 17:15:23"␣
↪/>
    <guestbook id="2" content="I like it!" user="##NULL##" created="2010-04-26␣
↪12:14:20" />
</dataset>
```

Luego, envolvemos el Conjunto de Datos XML Plano con el Conjunto de Datos de Remplazo:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class ReplacementTest extends TestCase
{
    use TestCaseTrait;

    public function getDataSet()
    {
        $ds = $this->createFlatXmlDataSet('myFlatXmlFixture.xml');
        $rds = new PHPUnit\DbUnit\DataSet\ReplacementDataSet($ds);
        $rds->addFullReplacement('##NULL##', null);
        return $rds;
    }
}
?>
```

Filtro de Conjunto de Datos

Si tenemos un archivo de ambientación muy grande podemos usar el Filtro de Conjunto de Datos para crear una lista blanco o negra de tablas y columnas que contendrá un subconjunto de datos. Esto es especialmente útil, cuando se combina con el Conjunto de Datos basado en Base de Datos, para filtrar las columnas del conjunto de datos.

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class DataSetFilterTest extends TestCase
{
    use TestCaseTrait;

    public function testIncludeFilteredGuestbook()
    {
        $tableNames = ['guestbook'];
        $dataSet = $this->getConnection()->createDataSet();

        $filterDataSet = new PHPUnit\DbUnit\DataSet\DataSetFilter($dataSet);
        $filterDataSet->addIncludeTables(['guestbook']);
        $filterDataSet->setIncludeColumnsForTable('guestbook', ['id', 'content']);
        // ..
    }
```

```
    public function testExcludeFilteredGuestbook()
    {
        $tableNames = ['guestbook'];
        $dataSet = $this->getConnection()->createDataSet();

        $filterDataSet = new PHPUnit\DbUnit\DataSet\DataSetFilter($dataSet);
        $filterDataSet->addExcludeTables(['foo', 'bar', 'baz']); // only keep the␣
↪guestbook table!
        $filterDataSet->setExcludeColumnsForTable('guestbook', ['user', 'created']);
        // ..
    }
}
?>
```

Nota

No podemos usar ambos filtros de columnas, exclusión e inclusión, sobre la misma tabla, solo es posible sobre diferentes tablas. Ademas, solo es posible la lista blanca de tablas o la negra, no ambas.

Conjunto de Datos Compuesto

El Conjunto de Datos compuesto es muy útil para agregar varios conjuntos de datos que ya existen dentro de un solo conjunto de datos. Cuando varios conjuntos de datos pertenecen a la misma tabla, las filas se añaden en el orden especificado. Por ejemplo si tenemos dos conjuntos de datos *fixture1.xml*:

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" created="2010-04-24 17:15:23"␣
↪/>
</dataset>
```

y *fixture2.xml*:

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="2" content="I like it!" user="##NULL##" created="2010-04-26␣
↪12:14:20" />
</dataset>
```

Usando el Conjunto de Datos Compuesto podemos agregar ambos archivos a la ambientación:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class CompositeTest extends TestCase
{
    use TestCaseTrait;

    public function getDataSet()
    {
        $ds1 = $this->createFlatXmlDataSet('fixture1.xml');
        $ds2 = $this->createFlatXmlDataSet('fixture2.xml');

        $compositeDs = new PHPUnit\DbUnit\DataSet\CompositeDataSet();
```

```
        $compositeDs->addDataSet($ds1);
        $compositeDs->addDataSet($ds2);

        return $compositeDs;
    }
}
?>
```

8.5.2 Precauciones con las Llaves Foráneas

La Extensión de Base de Datos de PHPUnit inserta las columnas en la base de datos en el orden en que ellas se especifican en nuestra ambientación. Si nuestro esquema de base de datos usa llaves foráneas debemos especificar las tablas en un orden que no cause que las restricciones de llave foránea fallen.

8.5.3 Implementar nuestro propio Conjunto de Datos/Tablas de Datos

Para entender el interior de los Conjuntos de Datos y Tablas de Datos, vamos a dar un vistazo a la interfaz de un Conjunto de Datos. Se puede saltar esta parte si no planeamos implementar nuestro propio Conjunto de Datos o Tabla de Datos.

```
<?php
namespace PHPUnit\DbUnit\DataSet;

interface IDataSet extends IteratorAggregate
{
    public function getTableNames();
    public function getTableMetaData($tableName);
    public function getTable($tableName);
    public function assertEquals(IDataSet $other);

    public function getReverseIterator();
}
?>
```

La interfaz pública es usada internamente por la aserción `assertDataSetsEqual()` en el Caso de Prueba de Base de Datos para revisar la calidad del conjunto de datos. De la interfaz `IteratorAggregate` la clase IDataSet hereda el método `getIterator()` para iterar sobre todas las tablas del conjunto de datos. El iterador reverso permite a PHPUnit truncar tablas en el orden opuesto al que ellas fueron creadas para satisfacer las restricciones de llave foránea.

Dependiendo de la implementación, se toman diferentes enfoques para agregar instancias de tabla a un conjunto de datos. Por ejemplo, las tablas se agregan internamente durante la construcción del archivo fuente en todos los Conjuntos de Datos basados en archivo, tales como `YamlDataSet`, `XmlDataSet` o `FlatXmlDataSet`.

Una tabla también se representa con la siguiente interfaz:

```
<?php
interface ITable
{
    public function getTableMetaData();
    public function getRowCount();
    public function getValue($row, $column);
    public function getRow($row);
    public function assertEquals(ITable $other);
}
?>
```

Con la excepción del método `getTableMetaData()`, el código anterior es bastante autoexplicativo. Los métodos usados son todos requeridos para las diferentes aserciones de la Extensión de la Base de Datos que se explican en el siguiente capítulo. El método `getTableMetaData()` debe regresar una implementación de la interfaz `PHPUnit\DbUnit\DataSet\ITableMetaData`, que describe la estructura de la tabla. La información que posee es:

- El nombre de la tabla.
- Un arreglo con los nombres de las columnas de la tabla, ordenada por su aparición en el conjunto de resultados.
- Un arreglo de las columnas que son llave primaria.

Esta interfaz tiene además una aserción que revisa si dos instancias de los Metadatos de la Tabla son iguales entre si, que es usado por la aserción de igualdad de conjunto de datos.

8.6 Usar la API de Conexión de Base de Datos

Existen tres métodos interesantes en la interfaz de Conexión que debe regresar el método `getConnection()` en el Caso de Prueba de Base de Datos:

```
<?php
namespace PHPUnit\DbUnit\Database;

interface Connection
{
    public function createDataSet(array $tableNames = null);
    public function createQueryTable($resultName, $sql);
    public function getRowCount($tableName, $whereClause = null);

    // ...
}
?>
```

1. El método `createDataSet()` crea un Conjunto de Datos basado en Base de Datos (DB) como se describe en la sección de implementaciones de Conjunto de Datos.

   ```
   <?php
   use PHPUnit\Framework\TestCase;
   use PHPUnit\DbUnit\TestCaseTrait;

   class ConnectionTest extends TestCase
   {
       use TestCaseTrait;

       public function testCreateDataSet()
       {
           $tableNames = ['guestbook'];
           $dataSet = $this->getConnection()->createDataSet();
       }
   }
   ?>
   ```

2. El método `createQueryTable()` se puede usar para crear instancias de una «Consulta a Tabla» (QueryTable), dado el nombre de una tabla y una consulta SQL.

   ```
   <?php
   use PHPUnit\Framework\TestCase;
   ```

```
use PHPUnit\DbUnit\TestCaseTrait;

class ConnectionTest extends TestCase
{
    use TestCaseTrait;

    public function testCreateQueryTable()
    {
        $tableNames = ['guestbook'];
        $queryTable = $this->getConnection()->createQueryTable('guestbook',
↪'SELECT * FROM guestbook');
    }
}
?>
```

3. El método `getRowCount()` es una manera conveniente de acceder al número de filas de una tabla, opcionalmente filtrado por una clausula «where». Este método se puede usar con una aserción simple de igualdad:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class ConnectionTest extends TestCase
{
    use TestCaseTrait;

    public function testGetRowCount()
    {
        $this->assertSame(2, $this->getConnection()->getRowCount('guestbook'));
    }
}
?>
```

8.7 API de Aserciones de Base de Datos

Una herramienta de pruebas como la Extensión de Base de Datos debe proveer algunas aserciones que podemos usar para revisar el estado actual de la base de datos, las tablas y la cantidad de filas de una tabla. En esta sección se describe estas funcionalidades detalladamente:

8.7.1 Aseverar el número de filas de una Tabla

A menudo es útil revisar si una tabla contiene una cantidad específica de filas. Podemos fácilmente conseguir esto sin código de enlace adicional usando la API de conexión. Supongamos que queremos revisar si después de insertar una fila en nuestro libro de visitas no solo tenemos las dos entradas iniciales, las que nos han acompañado en todos los ejemplos anteriores, sino tres:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class GuestbookTest extends TestCase
{
    use TestCaseTrait;
```

```
    public function testAddEntry()
    {
        $this->assertSame(2, $this->getConnection()->getRowCount('guestbook'), "Pre-
↪Condition");

        $guestbook = new Guestbook();
        $guestbook->addEntry("suzy", "Hello world!");

        $this->assertSame(3, $this->getConnection()->getRowCount('guestbook'),
↪"Inserting failed");
    }
}
?>
```

8.7.2 Aseverar el Estado de una Tabla

La aseveración anterior es útil pero seguramente queremos revisar el contenido real de una tabla para verificar que todos los valores se escribieron correctamente en las columnas. Esto se puede lograr con una aserción de tabla.

Para esto definiremos una instancia de Consulta de Tabla que obtiene su contenido del nombre de una tabla y una consulta SQL, y luego la compara con un Conjunto de Datos basado en Archivos o Arreglos:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class GuestbookTest extends TestCase
{
    use TestCaseTrait;

    public function testAddEntry()
    {
        $guestbook = new Guestbook();
        $guestbook->addEntry("suzy", "Hello world!");

        $queryTable = $this->getConnection()->createQueryTable(
            'guestbook', 'SELECT * FROM guestbook'
        );
        $expectedTable = $this->createFlatXmlDataSet("expectedBook.xml")
                              ->getTable("guestbook");
        $this->assertTablesEqual($expectedTable, $queryTable);
    }
}
?>
```

Ahora debemos escribir el archivo XML Plano *expectedBook.xml* para esta aserción:

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" created="2010-04-24 17:15:23"␣
↪/>
    <guestbook id="2" content="I like it!" user="nancy" created="2010-04-26 12:14:20"␣
↪/>
    <guestbook id="3" content="Hello world!" user="suzy" created="2010-05-01 21:47:08
↪" />
</dataset>
```

Sin embargo esta aserción solo se cumple durante un segundo, *2010–05–01 21:47:08*. Las fechas representan un problema especial para las pruebas de base de datos, podemos evitar estas fallas omitiendo la columna «created» en la aserción.

El archivo XML Plano *expectedBook.xml* ajustado debe verse de la siguiente manera para que la aserción pase la prueba:

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" />
    <guestbook id="2" content="I like it!" user="nancy" />
    <guestbook id="3" content="Hello world!" user="suzy" />
</dataset>
```

Debemos arreglar la llamada a la Consulta de Tabla:

```
<?php
$queryTable = $this->getConnection()->createQueryTable(
    'guestbook', 'SELECT id, content, user FROM guestbook'
);
?>
```

8.7.3 Aseverar el Resultado de una Consulta

Además podemos aseverar el resultado de una consulta compleja con el enfoque de Consulta de Tabla, solamente especificando el resultado de la consulta y comparándola con un conjunto de datos:

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class ComplexQueryTest extends TestCase
{
    use TestCaseTrait;

    public function testComplexQuery()
    {
        $queryTable = $this->getConnection()->createQueryTable(
            'myComplexQuery', 'SELECT complexQuery...'
        );
        $expectedTable = $this->createFlatXmlDataSet("complexQueryAssertion.xml")
                              ->getTable("myComplexQuery");
        $this->assertTablesEqual($expectedTable, $queryTable);
    }
}
?>
```

8.7.4 Aseverar el Estado de Varias Tablas

Sin dudas podemos aseverar el estado de multiples tablas de una sola vez, comparando un conjunto de datos basados en una consulta contra un conjunto de datos basados en archivos. Existen dos maneras diferentes para las aserciones de Conjunto de Datos.

1. Podemos usar el Conjunto de Datos de Base de Datos (DB) y compararlo con un Conjunto de Datos basado en Archivos.

```php
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;

class DataSetAssertionsTest extends TestCase
{
    use TestCaseTrait;

    public function testCreateDataSetAssertion()
    {
        $dataSet = $this->getConnection()->createDataSet(['guestbook']);
        $expectedDataSet = $this->createFlatXmlDataSet('guestbook.xml');
        $this->assertDataSetsEqual($expectedDataSet, $dataSet);
    }
}
?>
```

2. Podemos construir nuestro propio Conjunto de Datos:

```php
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;
use PHPUnit\DbUnit\DataSet\QueryDataSet;

class DataSetAssertionsTest extends TestCase
{
    use TestCaseTrait;

    public function testManualDataSetAssertion()
    {
        $dataSet = new QueryDataSet();
        $dataSet->addTable('guestbook', 'SELECT id, content, user FROM guestbook
↪'); // additional tables
        $expectedDataSet = $this->createFlatXmlDataSet('guestbook.xml');

        $this->assertDataSetsEqual($expectedDataSet, $dataSet);
    }
}
?>
```

8.8 Preguntas y Respuestas Comunes

8.8.1 ¿PHPUnit (re)creará el esquema de base de datos para cada prueba?

No, PHPUnit necesita que todos los objetos de la base de datos estén disponibles cuando las pruebas comiencen. La base de datos, tablas, secuencias, lanzadores y vistas se deben crear antes de ejecutar el conjunto de pruebas.

Doctrine 2 o eZ Components tienen poderosas herramientas que permiten crear el esquema de base de datos desde una estructura de datos predefinida. Sin embargo, ellas deben estar enlazadas a la extensión de PHPUnit para permitir la recreación automática de la base de datos antes de ejecutar el paquete de pruebas completo.

Como cada prueba limpia completamente la base de datos no necesitamos recrear la base de datos para ejecutar cada prueba. Una base de datos disponible permanentemente funciona perfectamente.

8.8.2 ¿Estoy obligado a usar PDO en mi aplicación para que la Extensión de Base de Datos funcione?

No, PDO solo es obligatorio para limpiar y configurar el ambiente y para las aserciones. Podemos usar cualquier abstracción de base de datos dentro de nuestro código.

8.8.3 ¿Que puedo hacer cuando recibo un Error «Too much Connections»?

Si no guardamos la instancia PDO que se crea con el método `getConnection()` del Caso de Prueba el número de conexiones a la base de datos se incrementará una o más veces por cada prueba de base de datos. La configuración por defecto de MySQL solo permite 100 conexiones concurrentes y otros proveedores también tienen un límite máximo de conexiones.

La subsección «Usemos nuestro propio Caso Abstracto de Prueba de Base de Datos» muestra como podemos prevenir que este error suceda usando una sola instancia PDO, guardada en la caché, para todos nuestras pruebas.

8.8.4 ¿Como lidiar con valores NULL en los Conjuntos de Datos XML Plano y CSV?

No lo hagas. En su lugar, deberíamos usar el Conjunto de Datos XML o el YAML.

CAPÍTULO 9

Dobles de Prueba

Gerard Meszaros introduce el concepto de Dobles de Prueba, en ingles *Test Doubles*, de la siguiente manera en Meszaros2007:

> *Gerard Meszaros*:
>
> A veces es lisa y llanamente difícil probar el sistema bajo prueba (SUT) porque el sistema depende de otros componentes que no se pueden usar en el entorno de pruebas. Esto podría ser porque ellos no están disponibles, no regresarán los resultados necesarios para la prueba o porque la ejecución de esos componentes tendría efectos secundarios indeseables. En otros casos, nuestra estrategia de prueba necesita que nosotros tengamos más control o visibilidad del comportamiento interno del SUT.
>
> Cuando estamos escribiendo una prueba en que no podemos (o elegimos no) usar un componente real del que se depende (DOC), podemos reemplazarlo por una Doble Prueba. El Doble de Prueba no debe comportarse exactamente como el DOC real; él solamente debe proveer la misma API que la dependencia real para que así el SUT piense que es una dependencia real.

Los métodos `createMock($type)` y `getMockBuilder($type)` que provee PHPUnit se pueden usar en una prueba para generar automáticamente un objeto que actúa como un doble de pruebas del objeto original especificado, de tipo interfaz o nombre de clase. Este objeto doble de prueba se puede usar en cada contexto donde un objeto del tipo original se espera o necesita.

El método `createMock($type)` regresa un doble de pruebas del objeto del tipo especificado (interfaz o clase). Por defecto, la creación de este doble de prueba se realiza usando las mejores prácticas. Los métodos `__construct()` y `__clone()` de la clase original no se ejecutan y los argumentos pasados a un método del doble de pruebas no se clonan. Si estas configuraciones predeterminadas no son las que necesitamos podemos usar el método `getMockBuilder($type)` para personalizar la generación del doble de pruebas usando una interfaz fluida.

Por defecto, todos los métodos de la clases original se reemplazan por una implementación simulada que solo regresa `null` (sin llamar al método original). Usando el método `will($this->returnValue())`, por ejemplo, podemos configurar esas implementaciones simuladas para que regresen un valor cuando sean llamadas.

Limitaciones: métodos final, private y static

Nótese que los métodos `final`, `private` y `static` no se pueden esbozar, *stubbed*, o simular *mocked*. La funcionalidad de dobles de prueba de PHPUnit ignora a estos métodos que mantienen su compor-

tamiento original excepto los métodos `static` que se reemplazan por el método lanzador de excepciones `\PHPUnit\Framework\MockObject\BadMethodCallException`.

9.1 Esbozos

La practica de remplazar un objeto con un doble de pruebas, que (opcionalmente) regresa valores de retorno configurados, se llama *stubbing*, quizás bosquejar. Podemos usar un *stub*, quizás bosquejo o esbozo, para «reemplazar un componente real del que el SUT depende y así la prueba tiene un punto de control para las entradas indirectas del SUT. Esto permite probar de manera forzada al SUT por rutas que de otra manera no se ejecutarían».

El Example 9.2 muestra como esbozar llamadas a métodos y configurar valores de retorno. Primero usamos el método `createMock()`, que provee la clase `PHPUnit\Framework\TestCase`, para configurar el esbozo del objeto que se verá como un objeto de la clase `SomeClass` (Example 9.1). Luego usamos una Interfaz Fluida que PHPUnit provee para especificar el comportamiento del esbozo. En esencia, esto significa que no necesitamos crear varios objetos temporales y después unirlos. En su lugar, encadenamos las llamadas a los métodos como se muestra en el ejemplo. Esto lleva a tener un código más legible y «fluido».

Example 9.1: La clase que queremos esbozar

```
<?php
class SomeClass
{
    public function doSomething()
    {
        // Do something.
    }
}
?>
```

Example 9.2: Esbozo de una llamada a un método que regresa un valor asignado

```
<?php
use PHPUnit\Framework\TestCase;

class StubTest extends TestCase
{
    public function testStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

        // Configure the stub.
        $stub->method('doSomething')
             ->willReturn('foo');

        // Calling $stub->doSomething() will now return
        // 'foo'.
        $this->assertSame('foo', $stub->doSomething());
    }
}
?>
```

Limitación: Métodos llamados «method»

El ejemplo de arriba solo funciona cuando en la clase original no se declara un método llamado «método».

Si la clase original declara un método llamado «method» entonces debemos usar `$stub->expects($this->any())->method('doSomething')->willReturn('foo');`.

«Detrás de bastidores» PHPUnit automáticamente genera una nueva clase PHP que implementa el comportamiento deseado cuando se usa el método `createMock()`.

El Example 9.3 muestra un ejemplo de como usar la interfaz fluida del *Mock Builder* para configurar la creación de un doble de pruebas. La configuración de este doble de pruebas usa las mismas buenas practicas que por defecto usa el método `createMock()`.

Example 9.3: La *Mock Builder API* se puede usar para configurar la generación del doble de pruebas de clase

```
<?php
use PHPUnit\Framework\TestCase;

class StubTest extends TestCase
{
    public function testStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->getMockBuilder(SomeClass::class)
                     ->disableOriginalConstructor()
                     ->disableOriginalClone()
                     ->disableArgumentCloning()
                     ->disallowMockingUnknownTypes()
                     ->getMock();

        // Configure the stub.
        $stub->method('doSomething')
             ->willReturn('foo');

        // Calling $stub->doSomething() will now return
        // 'foo'.
        $this->assertSame('foo', $stub->doSomething());
    }
}
?>
```

Hasta ahora con los ejemplos anteriores regresamos valores simples usando el método `willReturn($value)`. Esa sintaxis corta es equivalente a `will($this->returnValue($value))`. Podemos usar variaciones de esta sintaxis más larga y obtener un comportamiento más complejo para el esbozo.

A veces queremos regresar como resultado de la llamada al método esbozado uno de los argumentos del método llamado (sin cambios). El Example 9.4 muestra como podemos hacer esto usando el método `returnArgument()` en lugar de `returnValue()`.

Example 9.4: Llamada a un método esbozado que regresa uno de sus argumentos

```
<?php
use PHPUnit\Framework\TestCase;

class StubTest extends TestCase
{
    public function testReturnArgumentStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);
```

```
        // Configure the stub.
        $stub->method('doSomething')
             ->will($this->returnArgument(0));

        // $stub->doSomething('foo') returns 'foo'
        $this->assertSame('foo', $stub->doSomething('foo'));

        // $stub->doSomething('bar') returns 'bar'
        $this->assertSame('bar', $stub->doSomething('bar'));
    }
}
?>
```

Cuando se prueba una interfaz fluida, a veces es útil tener un método esbozado que regresa una referencia al objeto esbozado. El Example 9.5 muestra como alcanzar este objetivo con el método `returnSelf()`.

Example 9.5: Esbozar la llamada a un método que regresa un referencia al objeto esbozado

```
<?php
use PHPUnit\Framework\TestCase;

class StubTest extends TestCase
{
    public function testReturnSelf()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

        // Configure the stub.
        $stub->method('doSomething')
             ->will($this->returnSelf());

        // $stub->doSomething() returns $stub
        $this->assertSame($stub, $stub->doSomething());
    }
}
?>
```

Algunos de los métodos esbozados deberían regresar diferentes valores dependiendo de una lista predefinida de argumentos. Podemos usar el método `returnValueMap()` para crear un mapa que asocia argumentos con valores de retorno. Veamos el ejemplo Example 9.6 para un ejemplo.

Example 9.6: Esbozar la llamada a un método para regresar un valor desde un mapa

```
<?php
use PHPUnit\Framework\TestCase;

class StubTest extends TestCase
{
    public function testReturnValueMapStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

        // Create a map of arguments to return values.
        $map = [
            ['a', 'b', 'c', 'd'],
```

```
            ['e', 'f', 'g', 'h']
        ];

        // Configure the stub.
        $stub->method('doSomething')
             ->will($this->returnValueMap($map));

        // $stub->doSomething() returns different values depending on
        // the provided arguments.
        $this->assertSame('d', $stub->doSomething('a', 'b', 'c'));
        $this->assertSame('h', $stub->doSomething('e', 'f', 'g'));
    }
}
?>
```

Cuando la llamada a un esbozo de método debe regresar un valor calculado en lugar de un valor fijado (ver `returnValue()`) o un argumento sin cambios (ver `returnArgument()`), podemos usar el método `returnCallback()` para tener un esbozo de método que regresa el resultado de una función o método de retro llamada. Ver el Example 9.7:

Example 9.7: Esbozar la llamada a un método que regresar un valor desde una retro llamada

```
<?php
use PHPUnit\Framework\TestCase;

class StubTest extends TestCase
{
    public function testReturnCallbackStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

        // Configure the stub.
        $stub->method('doSomething')
             ->will($this->returnCallback('str_rot13'));

        // $stub->doSomething($argument) returns str_rot13($argument)
        $this->assertSame('fbzrguvat', $stub->doSomething('something'));
    }
}
?>
```

Una alternativa simple para configurar un método de retro llamada puede ser especificando una lista de valores deseables de retorno. Podemos hacer esto con el método `onConsecutiveCalls()`. Ver el Example 9.8.

Example 9.8: Esbozar la llamada a un método que regresar una lista de valores en el orden especificado

```
<?php
use PHPUnit\Framework\TestCase;

class StubTest extends TestCase
{
    public function testOnConsecutiveCallsStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

        // Configure the stub.
```

```
        $stub->method('doSomething')
             ->will($this->onConsecutiveCalls(2, 3, 5, 7));

        // $stub->doSomething() returns a different value each time
        $this->assertSame(2, $stub->doSomething());
        $this->assertSame(3, $stub->doSomething());
        $this->assertSame(5, $stub->doSomething());
    }
}
?>
```

En lugar de regresar un valor, un método esbozado puede además lanzar una excepción. El Example 9.9 el método muestra como usar el método `throwException()` para hacer esto.

Example 9.9: Esbozar la llama a un método para lanzar un excepción

```
<?php
use PHPUnit\Framework\TestCase;

class StubTest extends TestCase
{
    public function testThrowExceptionStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

        // Configure the stub.
        $stub->method('doSomething')
             ->will($this->throwException(new Exception));

        // $stub->doSomething() throws Exception
        $stub->doSomething();
    }
}
?>
```

Alternativamente, nosotros mismos podemos escribir un esbozo y mejorar su diseño a lo largo del camino. Los recursos usados ampliamente se acceden a través de una sola fachada, *single façade*, por lo que podemos fácilmente reemplazar el recurso con un esbozo. Por ejemplo, en lugar de tener llamadas directamente a la base de datos esparcidas a lo largo del código, podemos tener un solo objeto `Database` que implementación de la interfaz `IDatabase`. Luego, podemos crear un esbozo de la implementación de `IDatabase` y usarla para nuestras pruebas. Incluso podemos crear una opción para ejecutar las pruebas con el esbozo de base de datos o una base de datos real, así podemos usar nuestras pruebas tanto para pruebas locales durante el desarrollo como para la integración de las pruebas con una base de datos real.

Las funcionalidades que se necesitan esbozar tienden a ser agrupadas en el mismo objeto con lo que se mejora su cohesion. Al presenta la funcionalidad en una sola y coherente interfaz podemos reducir el acoplamiento con el resto del sistema.

9.2 Objetos Falsos

La práctica de reemplazar un objeto con un doble de pruebas que verifica las expectativas; por ejemplo, al aseverar que un método se ha llamado; tiene el nombre de *mocking*, quizás simulación o falsificación.

Podemos usar un *objeto falso* «como un punto de observación que se usa para verificar las salidas indirectas del SUT cuando se está *ejercitando*. Generalmente el objeto falso incluye además las funcionalidades de la prueba esbozada

puesto que él debe retornar valores al SUT, siempre que el sistema no ha fallado las pruebas, pero el énfasis está en la verificación de las salidas indirectas. Por eso, un objeto falso es mucho más que un esbozo de prueba más algunas aserciones; este se usa de una manera fundamentalmente diferente» (Gerard Meszaros).

Limitación: Verificación automática de las expectativas

Solo los objetos falsos generados dentro del ámbito de una prueba serán verificados automáticamente por PHPUnit. Los objetos falsos generados por los proveedores de datos, por ejemplo, o inyectados dentro de la prueba usando la anotación `@depends` no serán verificados automáticamente por PHPUnit.

Aquí tenemos un ejemplo: supongamos que queremos probar si el método correcto, `update()` en nuestro ejemplo, es llamado por un objeto que observa a otro objeto. El Example 9.10 muestra el código para las clases `Subject` y `Observer` que son parte del sistema que se está probando (SUT).

Example 9.10: Las clases *Subject* y *Observer* que son parte del sistema sometido a prueba (SUT)

```
<?php
use PHPUnit\Framework\TestCase;

class Subject
{
    protected $observers = [];
    protected $name;

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

    public function getName()
    {
        return $this->name;
    }

    public function attach(Observer $observer)
    {
        $this->observers[] = $observer;
    }

    public function doSomething()
    {
        // Do something.
        // ...

        // Notify observers that we did something.
        $this->notify('something');
    }

    public function doSomethingBad()
    {
        foreach ($this->observers as $observer) {
            $observer->reportError(42, 'Something bad happened', $this);
        }
    }

    protected function notify($argument)
    {
```

```
        foreach ($this->observers as $observer) {
            $observer->update($argument);
        }
    }

    // Other methods.
}

class Observer
{
    public function update($argument)
    {
        // Do something.
    }

    public function reportError($errorCode, $errorMessage, Subject $subject)
    {
        // Do something
    }

    // Other methods.
}
?>
```

El Example 9.11 muestra como usar un objeto falso para probar la interacción entre los objetos `Subject` y `Observer`.

Primero usamos el método `getMockBuilder()` que es provisto por la clase `PHPUnit\Framework\TestCase` para configurar un objeto falso para el `Observer`. Como damos un arreglo como segundo parámetro (opcional) para el método `getMock()`, solo el método `update()` de la clase `Observer` es reemplazada por la implementación falsa.

Como estamos interesados en revisar si se llama a un método y con que argumentos, introducimos los métodos `expects()` y `with()` para especificar como esta interacción debería darse.

Example 9.11: Probar si un método es llamado y con que argumentos

```
<?php
use PHPUnit\Framework\TestCase;

class SubjectTest extends TestCase
{
    public function testObserversAreUpdated()
    {
        // Create a mock for the Observer class,
        // only mock the update() method.
        $observer = $this->getMockBuilder(Observer::class)
                         ->setMethods(['update'])
                         ->getMock();

        // Set up the expectation for the update() method
        // to be called only once and with the string 'something'
        // as its parameter.
        $observer->expects($this->once())
                 ->method('update')
                 ->with($this->equalTo('something'));

        // Create a Subject object and attach the mocked
```

```
        // Observer object to it.
        $subject = new Subject('My subject');
        $subject->attach($observer);

        // Call the doSomething() method on the $subject object
        // which we expect to call the mocked Observer object's
        // update() method with the string 'something'.
        $subject->doSomething();
    }
}
?>
```

El método `with()` puede tomar cualquier número de argumentos mientras que correspondan con el número de argumentos que tienen el método que está siendo simulado (falsificado). Podemos especificar restricciones más avanzadas que una simple comparación en los argumentos del método.

Example 9.12: Probar que un método regresa con un número de argumentos restringidos de diferentes maneras

```
<?php
use PHPUnit\Framework\TestCase;

class SubjectTest extends TestCase
{
    public function testErrorReported()
    {
        // Create a mock for the Observer class, mocking the
        // reportError() method
        $observer = $this->getMockBuilder(Observer::class)
                         ->setMethods(['reportError'])
                         ->getMock();

        $observer->expects($this->once())
                 ->method('reportError')
                 ->with(
                       $this->greaterThan(0),
                       $this->stringContains('Something'),
                       $this->anything()
                   );

        $subject = new Subject('My subject');
        $subject->attach($observer);

        // The doSomethingBad() method should report an error to the observer
        // via the reportError() method
        $subject->doSomethingBad();
    }
}
?>
```

El método `withConsecutive()` puede tomar cualquier número de arreglos como argumentos dependiendo de las llamadas que deseamos probar. Cada arreglo es una lista de restricciones correspondientes a los argumentos del método que se está simulando, como en `with()`.

Example 9.13: Prueba que un método fue llamado dos veces con argumentos específicos.

```
<?php
use PHPUnit\Framework\TestCase;
```

```
class FooTest extends TestCase
{
    public function testFunctionCalledTwoTimesWithSpecificArguments()
    {
        $mock = $this->getMockBuilder(stdClass::class)
                     ->setMethods(['set'])
                     ->getMock();

        $mock->expects($this->exactly(2))
             ->method('set')
             ->withConsecutive(
                 [$this->equalTo('foo'), $this->greaterThan(0)],
                 [$this->equalTo('bar'), $this->greaterThan(0)]
             );

        $mock->set('foo', 21);
        $mock->set('bar', 48);
    }
}
?>
```

La restricción `callback()` se puede usar para la verificación de argumentos más complejos. Esta restricción recibe una retro llamada de PHP como único argumento. La retro llamada de PHP recibirá el argumento que será verificado como único argumento y debería retornar `true` si el argumento pasa la verificación y de lo contrario `false`.

Example 9.14: Verificación de argumentos más complejos

```
<?php
use PHPUnit\Framework\TestCase;

class SubjectTest extends TestCase
{
    public function testErrorReported()
    {
        // Create a mock for the Observer class, mocking the
        // reportError() method
        $observer = $this->getMockBuilder(Observer::class)
                         ->setMethods(['reportError'])
                         ->getMock();

        $observer->expects($this->once())
                 ->method('reportError')
                 ->with($this->greaterThan(0),
                        $this->stringContains('Something'),
                        $this->callback(function($subject){
                          return is_callable([$subject, 'getName']) &&
                                 $subject->getName() == 'My subject';
                        }));

        $subject = new Subject('My subject');
        $subject->attach($observer);

        // The doSomethingBad() method should report an error to the observer
        // via the reportError() method
        $subject->doSomethingBad();
    }
}
?>
```

Example 9.15: Prueba que el método fue llamado una vez y con un objeto idéntico al que fue llamado

```
<?php
use PHPUnit\Framework\TestCase;

class FooTest extends TestCase
{
    public function testIdenticalObjectPassed()
    {
        $expectedObject = new stdClass;

        $mock = $this->getMockBuilder(stdClass::class)
                     ->setMethods(['foo'])
                     ->getMock();

        $mock->expects($this->once())
             ->method('foo')
             ->with($this->identicalTo($expectedObject));

        $mock->foo($expectedObject);
    }
}
?>
```

Example 9.16: Crear un objeto falso con la clonación de parámetros habilitada

```
<?php
use PHPUnit\Framework\TestCase;

class FooTest extends TestCase
{
    public function testIdenticalObjectPassed()
    {
        $cloneArguments = true;

        $mock = $this->getMockBuilder(stdClass::class)
                     ->enableArgumentCloning()
                     ->getMock();

        // now your mock clones parameters so the identicalTo constraint
        // will fail.
    }
}
?>
```

Las restricciones, conocidas en ingles como *Constraints*, muestran las limitaciones que se pueden aplicar a los argumentos del método y en la Table 9.1 se muestran las comparaciones que están disponibles para especificar el número de invocaciones.

Table 9.1: Comparadores

Comparador	Significado
`PHPUnit\Framework\MockObject\Matcher\AnyInvokedCount any()`	Regresa la coincidencia que resulta cuando el método que se evalúa se ejecuta cero o más veces.
`PHPUnit\Framework\MockObject\Matcher\InvokedCount never()`	Regresa la coincidencia que resulta cuando el método que se evalúa nunca se ejecuta.
`PHPUnit\Framework\MockObject\Matcher\InvokedAtLeastOnce atLeastOnce()`	Regresa la coincidencia que resulta cuando el método que se evalúa se ejecuta al menos una vez.
`PHPUnit\Framework\MockObject\Matcher\InvokedCount once()`	Regresa la coincidencia que resulta cuando el método que se evalúa se ejecuta exactamente una vez.
`PHPUnit\Framework\MockObject\Matcher\InvokedCount exactly(int $count)`	Regresa la coincidencia que resulta cuando el método que se evalúa se ejecuta exactamente `$count` veces.
`PHPUnit\Framework\MockObject\Matcher\InvokedAtIndex at(int $index)`	Regresa la coincidencia que resulta cuando el método que se evalúa se invoca dada una variable `$index`.

Nota

El parámetro `$index` para el comparador `at()` se refiere al índice, comenzando en cero, de *todas las invocaciones de métodos* dado un objeto simulado. Tenga cuidado cuando ejecute este comparador pues nos puede llevar a pruebas frágiles cuando ellas están muy atadas a detalles específicos de la implementación.

Como mencionamos al comienzo, cuando los valores predeterminados usados por el método `createMock()` para generar los dobles de pruebas no satisfacen nuestras necesidades podemos usar el método `getMockBuilder($type)` para personalizar la generación de los dobles de prueba usando una interfaz fluida. Aquí hay una lista con los métodos que provee el *Mock Builder*:

- Se puede llamar al método `setMethods(array $methods)` sobre el objeto *Mock Builder* para especificar los métodos que serán reemplazados con un doble de prueba configurable. El comportamiento de los otros métodos no se carga. Si llamamos al método `setMethods(null)` ningún método será reemplazado.
- Se puede llamar al método `setMethodsExcept(array $methods)` sobre el objeto *Mock Builder* para especificar los métodos que no serán reemplazados con un doble de prueba configurable mientras que se reemplazan todos los otros métodos. Este método trabaja de forma inversa a `setMethods()`.
- Se puede llamar al método `setConstructorArgs(array $args)` para proveer un arreglo de parámetros que se pasa al constructor original de la clase (que por defecto no se reemplaza con una implementación falsa).
- Se puede llamar al método `setMockClassName($name)` para especificar un nombre de clase para la clase de dobles de prueba generada.

  `setMockClassName($name)` can be used to specify a class name for the generated test double class.
- Se puede usar el método `disableOriginalConstructor()` para inhabilitar la llamada al constructor de la clase original.
- El método `disableOriginalClone()` se puede usar para inhabilitar la llamada al constructor clone de la clase original.
- El método `disableAutoload()` se puede usar para inhabilitar el `__autoload()` durante la generación de la clase para el doble de pruebas.

9.3 Profecía

Prophecy es un «extremadamente dogmático pero muy poderoso y flexible framework de simulación de objetos PHP.

Aunque inicialmente fue creado para satisfacer las necesidades de phpspec2 es lo suficientemente flexible para usarse dentro de cualquier framework de pruebas con un mínimo esfuerzo».

PHPUnit tiene soporte incluido para usar *Prophecy* y crear dobles de prueba. El Example 9.17 muestra como la misma prueba del ejemplo Example 9.11 se puede expresar usando la filosofía de *Prophecy* de profecías y revelaciones:

Example 9.17: Probar que un método es llamado una vez y con un argumento específico

```
<?php
use PHPUnit\Framework\TestCase;

class SubjectTest extends TestCase
{
    public function testObserversAreUpdated()
    {
        $subject = new Subject('My subject');

        // Create a prophecy for the Observer class.
        $observer = $this->prophesize(Observer::class);

        // Set up the expectation for the update() method
        // to be called only once and with the string 'something'
        // as its parameter.
        $observer->update('something')->shouldBeCalled();

        // Reveal the prophecy and attach the mock object
        // to the Subject.
        $subject->attach($observer->reveal());

        // Call the doSomething() method on the $subject object
        // which we expect to call the mocked Observer object's
        // update() method with the string 'something'.
        $subject->doSomething();
    }
}
?>
```

Es necesario revisar la documentación de *Prophecy* para mayores detalles de como crear, configurar y usar esbozos, espías y simulaciones con este framework alternativo para dobles de pruebas.

9.4 Simular *Traits* y Clases Abstractas

El método `getMockForTrait()` regresa un objeto falso que usa un *trait* específico. Todos los métodos abstractos del *trait* dado se simulan. Esto permite probar métodos concretos de un *trait*.

Example 9.18: Probar los métodos concretos de un *trait*

```
<?php
use PHPUnit\Framework\TestCase;

trait AbstractTrait
{
    public function concreteMethod()
    {
        return $this->abstractMethod();
    }
```

```
    public abstract function abstractMethod();
}

class TraitClassTest extends TestCase
{
    public function testConcreteMethod()
    {
        $mock = $this->getMockForTrait(AbstractTrait::class);

        $mock->expects($this->any())
             ->method('abstractMethod')
             ->will($this->returnValue(true));

        $this->assertTrue($mock->concreteMethod());
    }
}
?>
```

El método `getMockForAbstractClass()` regresa un objeto simulado para una clase abstracta. Todos los métodos de una clase abstracta se simulan. Esto permite probar los métodos concretos de una clase abstracta.

Example 9.19: Probar los métodos concretos de una clase abstracta

```
<?php
use PHPUnit\Framework\TestCase;

abstract class AbstractClass
{
    public function concreteMethod()
    {
        return $this->abstractMethod();
    }

    public abstract function abstractMethod();
}

class AbstractClassTest extends TestCase
{
    public function testConcreteMethod()
    {
        $stub = $this->getMockForAbstractClass(AbstractClass::class);

        $stub->expects($this->any())
             ->method('abstractMethod')
             ->will($this->returnValue(true));

        $this->assertTrue($stub->concreteMethod());
    }
}
?>
```

9.5 Esbozar y Simular Servicios Web

Cuando nuestra aplicación interactúa con servicios web quisiéramos probarlos sin interactuar realmente con el servicio web. Para hacer el esbozo o la simulación de un servicio web, se puede usar el método `getMockFromWsdl()`

exactamente como `getMock()` (ver arriba). La única diferencia es que `getMockFromWsdl()` regresa un esbozo o simulación basado en una descripción de servicio web WSDL y `getMock()` regresa un esbozo o simulación basado en una clase o interfaz PHP.

El Example 9.20 muestra como `getMockFromWsdl()` se puede usar para esbozar, por ejemplo, el servicio web descrito en `GoogleSearch.wsdl`.

Example 9.20: Esbozar un servicio web

```
<?php
use PHPUnit\Framework\TestCase;

class GoogleTest extends TestCase
{
    public function testSearch()
    {
        $googleSearch = $this->getMockFromWsdl(
          'GoogleSearch.wsdl', 'GoogleSearch'
        );

        $directoryCategory = new stdClass;
        $directoryCategory->fullViewableName = '';
        $directoryCategory->specialEncoding = '';

        $element = new stdClass;
        $element->summary = '';
        $element->URL = 'https://phpunit.de/';
        $element->snippet = '...';
        $element->title = '<b>PHPUnit</b>';
        $element->cachedSize = '11k';
        $element->relatedInformationPresent = true;
        $element->hostName = 'phpunit.de';
        $element->directoryCategory = $directoryCategory;
        $element->directoryTitle = '';

        $result = new stdClass;
        $result->documentFiltering = false;
        $result->searchComments = '';
        $result->estimatedTotalResultsCount = 3.9000;
        $result->estimateIsExact = false;
        $result->resultElements = [$element];
        $result->searchQuery = 'PHPUnit';
        $result->startIndex = 1;
        $result->endIndex = 1;
        $result->searchTips = '';
        $result->directoryCategories = [];
        $result->searchTime = 0.248822;

        $googleSearch->expects($this->any())
                     ->method('doGoogleSearch')
                     ->will($this->returnValue($result));

        /**
         * $googleSearch->doGoogleSearch() will now return a stubbed result and
         * the web service's doGoogleSearch() method will not be invoked.
         */
        $this->assertEquals(
          $result,
          $googleSearch->doGoogleSearch(
```

```
            '00000000000000000000000000000000',
            'PHPUnit',
            0,
            1,
            false,
            '',
            false,
            '',
            '',
            ''
        )
    );
  }
}
?>
```

9.6 Simular el Sistema de Archivos

vfsStream es un envoltorio para flujos, *stream wrapper*, para un sistema de archivos virtual que puede ser útil en pruebas unitarias para simular un sistema de archivos real.

Si usamos Composer como administración de dependencias en nuestro proyecto, simplemente agregamos el paquete `mikey179/vfsStream` como dependencia en nuestro archivo `composer.json` del proyecto. Abajo hay un ejemplo de un archivo `composer.json` simplificado que define las dependencias: PHPUnit 4.6 y vfsStream, en *tiempo de desarrollo*.

```
{
    "require-dev": {
        "phpunit/phpunit": "~4.6",
        "mikey179/vfsStream": "~1"
    }
}
```

El Example 9.21 muestra una clase que interactúa con el sistema de archivos.

Example 9.21: Una clase que interactúa con el sistema de archivos

```
<?php
use PHPUnit\Framework\TestCase;

class Example
{
    protected $id;
    protected $directory;

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

    public function setDirectory($directory)
    {
        $this->directory = $directory . DIRECTORY_SEPARATOR . $this->id;

        if (!file_exists($this->directory)) {
            mkdir($this->directory, 0700, true);
```

```
        }
    }
}?>
```

Sin un sistema de archivos virtual como vfsStream no podemos probar el método `setDirectory()` aislado de influencias externas (ver Example 9.22).

Example 9.22: Probar una clase que interactúa con el sistema de archivos

```
<?php
use PHPUnit\Framework\TestCase;

class ExampleTest extends TestCase
{
    protected function setUp()
    {
        if (file_exists(dirname(__FILE__) . '/id')) {
            rmdir(dirname(__FILE__) . '/id');
        }
    }

    public function testDirectoryIsCreated()
    {
        $example = new Example('id');
        $this->assertFalse(file_exists(dirname(__FILE__) . '/id'));

        $example->setDirectory(dirname(__FILE__));
        $this->assertTrue(file_exists(dirname(__FILE__) . '/id'));
    }

    protected function tearDown()
    {
        if (file_exists(dirname(__FILE__) . '/id')) {
            rmdir(dirname(__FILE__) . '/id');
        }
    }
}
?>
```

La estrategia de arriba tiene varias desventajas:

- Como con cualquier recurso externo, podría haber problemas de intermitencia con el sistema de archivos. Esto hace fragiles a las pruebas que interactuaran con el sistema de archivos.
- En los métodos `setUp()` y `tearDown()` debemos asegurarnos que la carpeta no existe ni antes ni después de la prueba.
- Cuando la ejecución de la prueba termina antes de que el método `tearDown()` es invocado la carpeta permanecerá en el sistema de archivos.

El Example 9.23 muestra como vfsStream se puede usar para simular el sistema de archivos en una prueba para una clase que interactúa con el sistema de archivos.

Example 9.23: Simular el sistema de archivos para una prueba que interactúa con el sistema de archivos

```
<?php
use PHPUnit\Framework\TestCase;

class ExampleTest extends TestCase
```

```
{
    public function setUp()
    {
        vfsStreamWrapper::register();
        vfsStreamWrapper::setRoot(new vfsStreamDirectory('exampleDir'));
    }

    public function testDirectoryIsCreated()
    {
        $example = new Example('id');
        $this->assertFalse(vfsStreamWrapper::getRoot()->hasChild('id'));

        $example->setDirectory(vfsStream::url('exampleDir'));
        $this->assertTrue(vfsStreamWrapper::getRoot()->hasChild('id'));
    }
}
?>
```

Esto tiene algunas ventajas:

- La prueba misma es más concisa.
- vfsStream otorga al desarrollador control total sobre la configuración del sistema de archivos para el código que se prueba.
- Como las operaciones sobre el sistema de archivos no se ejecutan sobre un sistema de archivos real, las operaciones de limpieza que se colocan en el método `tearDown()` no son necesarias.

CAPÍTULO 10

Análisis de Cobertura de Código

> *Wikipedia*:
>
> En ciencias de la computación la cobertura de código es una medida usada para describir el grado en que el código fuente de un programa se ha probado por medio de un conjunto de pruebas. Un programa con una alta cobertura de código ha sido probado más profundamente y tiene una baja probabilidad de contener errores en comparación con un programa con una baja cobertura de código.

En este capítulo aprenderemos todo sobre la cobertura de código de PHPUnit, funcionalidad que provee una idea de que parte del código de producción se ejecuta cuando se hacen las pruebas. La funcionalidad se vale del componente php-code-coverage, que a su vez aprovecha la funcionalidad de cobertura de código que provee la extensión de PHP Xdebug.

Nota

Xdebug no se distribuye como parte de PHPUnit. Si recibimos una notificación mientras ejecutamos las pruebas indicando que el controlador de cobertura de código no esta disponible, es posible que Xdebug no este instalado o no este configurado apropiadamente. Antes de poder usar la característica de cobertura de código en PHPUnit deberíamos leer la guía de instalación de Xdebug.

php-code-coverage también soporta phpdbg como fuente alternativa de datos para la cobertura de código.

PHPUnit puede generar un reporte de cobertura de código basado en HTML, como también en archivos de registro de sucesos basados en XML, con la información de cobertura de código en varios formatos (Clover, Crap4J, PHPUnit). La información de cobertura de código se puede reportar como texto (e impresa por STDOUT) y exportada como código PHP para ser procesado posteriormente.

Podemos revisar *El Ejecutor de Pruebas desde Línea de Comandos* para ver una lista de comandos que acompañados con los interruptores adecuados permite controlar la funcionalidad de cobertura de código, también debemos revisar *Logging* para las configuraciones relevantes.

10.1 Métricas de Software para la Cobertura de Código

Existen varias métricas de software para medir la cobertura de código:

Line Coverage

La métrica de software *Line Coverage* mide si cada línea ejecutable fue ejecutada.

Function and Method Coverage

La métrica de software *Function and Method Coverage* mide si cada función o método se ha invocado. php-code-coverage solo considera una función o método como cubierto cuando todas sus líneas ejecutables fueron cubiertas.

Class and Trait Coverage

La métrica de software *Class and Trait Coverage* mide si cada método de una clase o *trait* fue cubierto. php-code-coverage solo considera una clase o *trait* como cubierto cuando todos sus métodos fueron cubiertos.

Opcode Coverage

La métrica de software *Opcode Coverage* mide si cada *código de operación*, «opcode», de una función o método se ha ejecutado mientras se corre la suite de prueba. Una línea de código se copila usualmente dentro de más de una linea de *código de operación*. La cobertura de línea considera a una línea de código cubierta tan pronto como sus *códigos de operación* se ejecutan.

Branch Coverage

La métrica de software *Branch Coverage* mide si la expresión booleana de cada estructura de control evalúa `true` y `false` mientras se ejecuta la suite de prueba.

Path Coverage

La métrica de software *Path Coverage* mide si cada una de las rutas posibles de ejecución de una función o método fue seguida durante la ejecución de la suite de prueba. Una ruta de ejecución es una secuencia única entre un conjunto de ramas desde el comienzo de la función o método hasta su salida.

Change Risk Anti-Patterns (CRAP) Index

El *Change Risk Anti-Patterns (CRAP) Index* se calcula en base a la complejidad ciclomática y cobertura de código de una unidad de código. El código que no es muy complejo y tiene una cobertura de código adecuada tendrá un índice CRAP bajo. El índice CRAP se puede bajar escribiendo pruebas y refactorizando el código para disminuir su complejidad.

Nota

Las métricas de software *Opcode Coverage*, *Branch Coverage* y *Path Coverage* aún no están soportadas por php-code-coverage.

10.2 Lista Blanca de Archivos

Es obligatorio configurar una *whitelist* para indicarle a PHPUnit que archivos de código fuente incluir en el reporte de cobertura de código. Esto se puede hacer o usando la opción de línea de comandos `--whitelist` o desde el archivo de configuración (ver *Lista Blanca de Archivos para la Cobertura de Código*).

Los valores de configuración `addUncoveredFilesFromWhitelist` y `processUncoveredFilesFromWhitelist` se usan para configurar la manera como se usa la lista blanca:

- `addUncoveredFilesFromWhitelist="false"` significa que solo los archivos en la lista blanca que por lo menos tienen una línea de código ejecutado se incluyen en el reporte de cobertura de código.
- `addUncoveredFilesFromWhitelist="true"` (por defecto) significa que todos los archivos en la lista blanca se incluyen en el reporte de cobertura de código incluso si no fue ejecutada ni una sola línea del archivo.
- `processUncoveredFilesFromWhitelist="false"` (por defecto) significa que un archivo listado en la lista blanca que no tiene líneas de código ejecutadas será agregado al reporte de cobertura de código (si se indica `addUncoveredFilesFromWhitelist="true"`) pero no será cargado por PHPUnit y por lo tanto no se analizará la corrección de las líneas de código de información.
- `processUncoveredFilesFromWhitelist="true"` significa que un archivo listado en la lista blanca que no tiene líneas de código ejecutadas será cargado por PHPUnit así que se puede analizar la corrección de las líneas de información del código ejecutables.

Nota

Por favor observemos que la carga de los archivos de código fuente que se ejecuta cuando se configura `processUncoveredFilesFromWhitelist="true"`, por ejemplo, puede causar problemas cuando un archivo de código fuente contiene código fuera del alcance de una clase o función.

10.3 Ignorar Bloques de Código

En ocasiones tenemos bloques de código que no podemos probar y que podríamos querer ignorar durante el análisis de cobertura de código. PHPUnit permite hacer esto usando las anotaciones `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` y `@codeCoverageIgnoreEnd` como se muestra en Example 10.1.

Example 10.1: Uso de las anotaciones `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` y `@codeCoverageIgnoreEnd`

```
<?php
use PHPUnit\Framework\TestCase;

/**
 * @codeCoverageIgnore
 */
class Foo
{
    public function bar()
    {
    }
}

class Bar
{
    /**
     * @codeCoverageIgnore
     */
    public function foo()
    {
    }
}

if (false) {
    // @codeCoverageIgnoreStart
```

```
    print '*';
    // @codeCoverageIgnoreEnd
}

exit; // @codeCoverageIgnore
?>
```

Las líneas de código ignoradas (marcadas como ignoradas usando las anotaciones) se cuentan como ejecutadas (si ellas son ejecutables) y no serán señaladas.

10.4 Especificar los Métodos de Cobertura

La anotación `@covers` (ver *Anotaciones que permite especificar que métodos son cubiertos por una prueba*) se puede usar en el código de prueba para especificar que método(s) un método de prueba quiere ejecutar. Si se provee, solo la información de cobertura de código para el o los métodos especificados serán considerados. El Example 10.2 muestra un ejemplo.

Example 10.2: Prueba que especifica que métodos se quieren cubrir

```
<?php
use PHPUnit\Framework\TestCase;

class BankAccountTest extends TestCase
{
    protected $ba;

    protected function setUp()
    {
        $this->ba = new BankAccount;
    }

    /**
     * @covers BankAccount::getBalance
     */
    public function testBalanceIsInitiallyZero()
    {
        $this->assertSame(0, $this->ba->getBalance());
    }

    /**
     * @covers BankAccount::withdrawMoney
     */
    public function testBalanceCannotBecomeNegative()
    {
        try {
            $this->ba->withdrawMoney(1);
        }

        catch (BankAccountException $e) {
            $this->assertSame(0, $this->ba->getBalance());

            return;
        }

        $this->fail();
    }
```

```
    /**
     * @covers BankAccount::depositMoney
     */
    public function testBalanceCannotBecomeNegative2()
    {
        try {
            $this->ba->depositMoney(-1);
        }

        catch (BankAccountException $e) {
            $this->assertSame(0, $this->ba->getBalance());

            return;
        }

        $this->fail();
    }

    /**
     * @covers BankAccount::getBalance
     * @covers BankAccount::depositMoney
     * @covers BankAccount::withdrawMoney
     */
    public function testDepositWithdrawMoney()
    {
        $this->assertSame(0, $this->ba->getBalance());
        $this->ba->depositMoney(1);
        $this->assertSame(1, $this->ba->getBalance());
        $this->ba->withdrawMoney(1);
        $this->assertSame(0, $this->ba->getBalance());
    }
}
?>
```

Además, es posible especificar que una prueba no debe cubrir *ningún* method usando la anotación `@coversNothing` (ver *@coversNothing*). Esto puede ser útil cuando escribimos pruebas de integración para asegurar que solo se genera cobertura de código para pruebas unitarias.

Example 10.3: Una prueba que especifica que ningún método será cubierto

```
<?php
use PHPUnit\DbUnit\TestCase

class GuestbookIntegrationTest extends TestCase
{
    /**
     * @coversNothing
     */
    public function testAddEntry()
    {
        $guestbook = new Guestbook();
        $guestbook->addEntry("suzy", "Hello world!");

        $queryTable = $this->getConnection()->createQueryTable(
            'guestbook', 'SELECT * FROM guestbook'
        );
```

```
        $expectedTable = $this->createFlatXmlDataSet("expectedBook.xml")
                              ->getTable("guestbook");

        $this->assertTablesEqual($expectedTable, $queryTable);
    }
}
?>
```

10.5 Casos Límite

Esta sección muestra interesantes casos límite que producen información de cobertura de código confusa.

```
<?php
use PHPUnit\Framework\TestCase;

// Because it is "line based" and not statement base coverage
// one line will always have one coverage status
if (false) this_function_call_shows_up_as_covered();

// Due to how code coverage works internally these two lines are special.
// This line will show up as non executable
if (false)
    // This line will show up as covered because it is actually the
    // coverage of the if statement in the line above that gets shown here!
    will_also_show_up_as_covered();

// To avoid this it is necessary that braces are used
if (false) {
    this_call_will_never_show_up_as_covered();
}
?>
```

CAPÍTULO 11

Registro

PHPUnit puede producir una serie de archivos de registro.

11.1 Resultados de Pruebas (XML)

El archivo de registro de resultados de pruebas en XML de PHPUnit está basado en él utilizado por la tarea JUnit para Apache Ant. El siguiente ejemplo muestra el archivo XML generado por las pruebas en `ArrayTest`:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="ArrayTest"
             file="/home/sb/ArrayTest.php"
             tests="2"
             assertions="2"
             failures="0"
             errors="0"
             time="0.016030">
    <testcase name="testNewArrayIsEmpty"
              class="ArrayTest"
              file="/home/sb/ArrayTest.php"
              line="6"
              assertions="1"
              time="0.008044"/>
    <testcase name="testArrayContainsAnElement"
              class="ArrayTest"
              file="/home/sb/ArrayTest.php"
              line="15"
              assertions="1"
              time="0.007986"/>
  </testsuite>
</testsuites>
```

El siguiente archivo XML fue generado por dos pruebas, `testFailure` y `testError`, de una clase de pruebas llamada `FailureErrorTest` y muestra cómo se guardan los fallos y errores.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="FailureErrorTest"
             file="/home/sb/FailureErrorTest.php"
             tests="2"
             assertions="1"
             failures="1"
             errors="1"
             time="0.019744">
    <testcase name="testFailure"
              class="FailureErrorTest"
              file="/home/sb/FailureErrorTest.php"
              line="6"
              assertions="1"
              time="0.011456">
      <failure type="PHPUnit_Framework_ExpectationFailedException">
testFailure(FailureErrorTest)
Failed asserting that &lt;integer:2&gt; matches expected value &lt;integer:1&gt;.

/home/sb/FailureErrorTest.php:8
</failure>
    </testcase>
    <testcase name="testError"
              class="FailureErrorTest"
              file="/home/sb/FailureErrorTest.php"
              line="11"
              assertions="0"
              time="0.008288">
      <error type="Exception">testError(FailureErrorTest)
Exception:

/home/sb/FailureErrorTest.php:13
</error>
    </testcase>
  </testsuite>
</testsuites>
```

11.2 Cobertura de Código (XML)

El formato XML para la información sobre cobertura de código producido por PHPUnit está ligeramente basado en el utilizado por Clover. El siguiente ejemplo muestra el registro XML generado por las pruebas en `BankAccountTest`:

```
<?xml version="1.0" encoding="UTF-8"?>
<coverage generated="1184835473" phpunit="3.6.0">
  <project name="BankAccountTest" timestamp="1184835473">
    <file name="/home/sb/BankAccount.php">
      <class name="BankAccountException">
        <metrics methods="0" coveredmethods="0" statements="0"
                 coveredstatements="0" elements="0" coveredelements="0"/>
      </class>
      <class name="BankAccount">
        <metrics methods="4" coveredmethods="4" statements="13"
                 coveredstatements="5" elements="17" coveredelements="9"/>
      </class>
      <line num="77" type="method" count="3"/>
```

```
      <line num="79" type="stmt" count="3"/>
      <line num="89" type="method" count="2"/>
      <line num="91" type="stmt" count="2"/>
      <line num="92" type="stmt" count="0"/>
      <line num="93" type="stmt" count="0"/>
      <line num="94" type="stmt" count="2"/>
      <line num="96" type="stmt" count="0"/>
      <line num="105" type="method" count="1"/>
      <line num="107" type="stmt" count="1"/>
      <line num="109" type="stmt" count="0"/>
      <line num="119" type="method" count="1"/>
      <line num="121" type="stmt" count="1"/>
      <line num="123" type="stmt" count="0"/>
      <metrics loc="126" ncloc="37" classes="2" methods="4" coveredmethods="4"
               statements="13" coveredstatements="5" elements="17"
               coveredelements="9"/>
    </file>
    <metrics files="1" loc="126" ncloc="37" classes="2" methods="4"
             coveredmethods="4" statements="13" coveredstatements="5"
             elements="17" coveredelements="9"/>
  </project>
</coverage>
```

11.3 Cobertura de Código (TEXT)

La cobertura de código en texto plano puede salir por la consola o a un archivo de texto.

El objetivo de este formato es proveer de un visual de la cobertura de código mientras se prueba un grupo pequeño de clases. Para proyectos más grandes esta salida puede ser útil para obtener una visión rápida de la cobertura del proyecto o simplemente se puede usar con la funcionalidad `--filter`. Cuando lo usamos desde la consola será escrito en `php://stdout`; respetando la configuración de `--colors`. Cuando se llama al comando la opción por defecto es escribir la salida en la consola. Por defecto se mostrará solo los archivos que tengan al menos una línea cubierta. Esto solo puede cambiarse con la opción `showUncoveredFiles` en la configuración para xml. Ver *Logging*. Por defecto todos los archivos y sus estados de cobertura se muestran en el informe detallado. Esto se puede cambiar con la opción `showOnlySummary` en la configuración para xml.

CAPÍTULO 12

Extender PHPUnit

PHPUnit se puede extender de varias maneras para hacer la escritura de las pruebas una tarea más fácil y para personalizar los mensajes que se obtienen a partir de la ejecución de las pruebas. Aquí presentamos los puntos de partida comunes para extender PHPUnit.

12.1 La Subclase PHPUnit\Framework\TestCase

Una de las maneras más fáciles de extender PHPUnit es escribir aserciones personalizadas y métodos útiles en una subclase abstracta de `PHPUnit\Framework\TestCase` y derivar nuestros clases de casos de prueba desde esta clase.

12.2 Escribir aserciones personalizadas

Cuando escribimos aserciones personalizadas es bueno seguir la manera como PHPUnit implementa sus propias aserciones. Como podemos ver en el Example 12.1, el método `assertTrue()` solo es un envoltorio alrededor de los métodos `isTrue()` y `assertThat()`. `isTrue()` crea un objeto con el que hacer la comparación que se pasa al método `assertThat()` para ser evaluado.

Example 12.1: Los métodos assertTrue() y isTrue() de la clase PHPUnit\Framework\Assert

```
<?php
namespace PHPUnit\Framework;

use PHPUnit\Framework\TestCase;

abstract class Assert
{
    // ...

    /**
     * Asserts that a condition is true.
```

```
     *
     * @param  boolean $condition
     * @param  string  $message
     * @throws PHPUnit\Framework\AssertionFailedError
     */
    public static function assertTrue($condition, $message = '')
    {
        self::assertThat($condition, self::isTrue(), $message);
    }

    // ...

    /**
     * Returns a PHPUnit\Framework\Constraint\IsTrue matcher object.
     *
     * @return PHPUnit\Framework\Constraint\IsTrue
     * @since  Method available since Release 3.3.0
     */
    public static function isTrue()
    {
        return new PHPUnit\Framework\Constraint\IsTrue;
    }

    // ...
}
```

El Example 12.2 muestra como la clase `PHPUnit\Framework\Constraint\IsTrue` extiende a la clase base abstracta `PHPUnit\Framework\Constraint` para construir los objetos de comparación (o restricciones).

Example 12.2: La clase PHPUnit\Framework\Constraint\IsTrue

```
<?php
namespace PHPUnit\Framework\Constraint;

use PHPUnit\Framework\Constraint;

class IsTrue extends Constraint
{
    /**
     * Evaluates the constraint for parameter $other. Returns true if the
     * constraint is met, false otherwise.
     *
     * @param mixed $other Value or object to evaluate.
     * @return bool
     */
    public function matches($other)
    {
        return $other === true;
    }

    /**
     * Returns a string representation of the constraint.
     *
     * @return string
     */
    public function toString()
    {
        return 'is true';
```

```
    }
}
```

La ventaja de implementar los métodos `assertTrue()` y `isTrue()` de la misma manera que la clase `PHPUnit\Framework\Constraint\IsTrue` está en que `assertThat()` automáticamente contabiliza la evaluación de la aserción para incluirla en las estadísticas. Además, el método `isTrue()` se puede usar como un objeto para la comparación cuando se configuran objetos falsos.

12.3 Implementar PHPUnit\Framework\TestListener

El Example 12.3 muestra una implementación simple de la interfaz `PHPUnit\Framework\TestListener`.

Example 12.3: Un escucha, «listener», de pruebas simple

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\Framework\TestListener;

class SimpleTestListener implements TestListener
{
    public function addError(PHPUnit\Framework\Test $test, Exception $e, $time): void
    {
        printf("Error while running test '%s'.\n", $test->getName());
    }

    public function addWarning(PHPUnit\Framework\Test $test,␣
↪PHPUnit\Framework\Warning $e, float $time): void
    {
        printf("Warning while running test '%s'.\n", $test->getName());
    }

    public function addFailure(PHPUnit\Framework\Test $test,␣
↪PHPUnit\Framework\AssertionFailedError $e, $time): void
    {
        printf("Test '%s' failed.\n", $test->getName());
    }

    public function addIncompleteTest(PHPUnit\Framework\Test $test, Exception $e,
↪$time): void
    {
        printf("Test '%s' is incomplete.\n", $test->getName());
    }

    public function addRiskyTest(PHPUnit\Framework\Test $test, Exception $e, $time):␣
↪void
    {
        printf("Test '%s' is deemed risky.\n", $test->getName());
    }

    public function addSkippedTest(PHPUnit\Framework\Test $test, Exception $e,
↪$time): void
    {
        printf("Test '%s' has been skipped.\n", $test->getName());
    }

    public function startTest(PHPUnit\Framework\Test $test): void
```

```
    {
        printf("Test '%s' started.\n", $test->getName());
    }

    public function endTest(PHPUnit\Framework\Test $test, $time): void
    {
        printf("Test '%s' ended.\n", $test->getName());
    }

    public function startTestSuite(PHPUnit\Framework\TestSuite $suite): void
    {
        printf("TestSuite '%s' started.\n", $suite->getName());
    }

    public function endTestSuite(PHPUnit\Framework\TestSuite $suite): void
    {
        printf("TestSuite '%s' ended.\n", $suite->getName());
    }
}
```

El Example 12.4 muestra como usar un el «trait» `PHPUnit\Framework\TestListenerDefaultImplementation`, que nos permite especificar sola los métodos de interfaz que son interesantes para nuestro caso de uso, mientras que se proveen implementaciones vacías para todos los otros.

Example 12.4: Usar la implementación «trait» por defecto en un escucha de pruebas

```
<?php
use PHPUnit\Framework\TestListener;
use PHPUnit\Framework\TestListenerDefaultImplementation;

class ShortTestListener implements TestListener
{
    use TestListenerDefaultImplementation;

    public function endTest(PHPUnit\Framework\Test $test, $time): void
    {
        printf("Test '%s' ended.\n", $test->getName());
    }
}
```

En el *«Listeners» de Prueba* podemos ver como configurar PHPUnit para adjuntar nuestro escucha de pruebas a la ejecución de una prueba.

12.4 Implementar PHPUnit\Framework\Test

La interfaz `PHPUnit\Framework\Test` es pequeña y fácil de implementar. Por ejemplo, podemos escribir una implementación de `PHPUnit\Framework\Test` que sea más simple que `PHPUnit\Framework\TestCase` y que ejecuten las *pruebas dirigidas por datos*.

El Example 12.5 muestra una clase de casos de pruebas dirigida por datos que usa un archivo con valores separados por comas (CSV). Cada línea del archivo es de la forma `foo;bar`, donde el primer valor es el valor esperado y el segundo es el valor real.

Example 12.5: Una prueba dirigida por datos

```
<?php
use PHPUnit\Framework\TestCase;

class DataDrivenTest implements PHPUnit\Framework\Test
{
    private $lines;

    public function __construct($dataFile)
    {
        $this->lines = file($dataFile);
    }

    public function count()
    {
        return 1;
    }

    public function run(PHPUnit\Framework\TestResult $result = null)
    {
        if ($result === null) {
            $result = new PHPUnit\Framework\TestResult;
        }

        foreach ($this->lines as $line) {
            $result->startTest($this);
            PHP_Timer::start();
            $stopTime = null;

            list($expected, $actual) = explode(';', $line);

            try {
                PHPUnit\Framework\Assert::assertEquals(
                  trim($expected), trim($actual)
                );
            }

            catch (PHPUnit\Framework\AssertionFailedError $e) {
                $stopTime = PHP_Timer::stop();
                $result->addFailure($this, $e, $stopTime);
            }

            catch (Exception $e) {
                $stopTime = PHP_Timer::stop();
                $result->addError($this, $e, $stopTime);
            }

            if ($stopTime === null) {
                $stopTime = PHP_Timer::stop();
            }

            $result->endTest($this, $stopTime);
        }

        return $result;
    }
}
```

```
$test = new DataDrivenTest('data_file.csv');
$result = PHPUnit\TextUI\TestRunner::run($test);
```

```
PHPUnit |version|.0 by Sebastian Bergmann and contributors.

.F

Time: 0 seconds

There was 1 failure:

1) DataDrivenTest
Failed asserting that two strings are equal.
expected string <bar>
difference      <  x>
got string      <baz>
/home/sb/DataDrivenTest.php:32
/home/sb/DataDrivenTest.php:53

FAILURES!
Tests: 2, Failures: 1.
```

12.5 Extender TestRunner

PHPUnit latest soporta extensiones para TestRunner que se pueden enganchar a varios eventos durante la ejecución de la prueba. Para más detalles sobre como registrar extensiones en la configuración XML de PHPUnit podemos ver *Registrar Extensiones de TestRunner*.

Cada evento disponible al que la extensión se puede enganchar se representa con una interfaz que la extensión debe implementar. La lista de eventos disponibles en PHPUnit latest se puede ver *Interfaces de Enganche Disponibles*.

12.5.1 Interfaces de Enganche Disponibles

- `AfterIncompleteTestHook`
- `AfterLastTestHook`
- `AfterRiskyTestHook`
- `AfterSkippedTestHook`
- `AfterSuccessfulTestHook`
- `AfterTestErrorHook`
- `AfterTestFailureHook`
- `AfterTestWarningHook`
- `BeforeFirstTestHook`
- `BeforeTestHook`

El Example 12.6 muestra un ejemplo para una extensión que implementa las interfaces `BeforeFirstTestHook` y `AfterLastTestHook`.

Example 12.6: Ejemplo de Extensión para el TestRunner

```php
<?php

namespace Vendor;

use PHPUnit\Runner\AfterLastTestHook;
use PHPUnit\Runner\BeforeFirstTestHook;

final class MyExtension implements BeforeFirstTestHook, AfterLastTestHook
{
    public function executeAfterLastTest(): void
    {
        // called after the last test has been run
    }

    public function executeBeforeFirstTest(): void
    {
        // called before the first test is being run
    }
}
```

CAPÍTULO 13

Aserciones

En este apéndice listamos varios métodos para hacer aserciones que están disponibles.

13.1 Static vs. Non-Static Usage of Assertion Methods

Las aserciones de PHPUnit están implementadas en `PHPUnit\Framework\Assert`. `PHPUnit\Framework\TestCase` hereda de `PHPUnit\Framework\Assert`.

Los métodos para hacer aserciones son declarados estáticos y se pueden invocar desde cualquier contexto usando `PHPUnit\Framework\Assert::assertTrue()`, `$this->assertTrue()` o `self::assertTrue()` en una clase que extiende de PHPUnit\Framework\TestCase.

De hecho, incluso podemos usar envoltorios de funciones globales como `assertTrue()` en cualquier contexto (incluyendo clases que extienden de `PHPUnit\Framework\TestCase`) cuando hemos incluido (manualmente) el archivo con código fuente `src/Framework/Assert/Functions.php` que viene con PHPUnit.

Una pregunta común, especialmente de los nuevos desarrolladores de PHPUnit, es cual es «la manera correcta» de invocar una aserción si usar `$this->assertTrue()` o `self::assertTrue()`. La respuesta corta es: no hay una manera correcta. Y tampoco hay una manera incorrecta. Es un asunto de preferencias personales.

La mayoría de las personas «se siente segura» usando `$this->assertTrue()` porque el método de prueba se invoca en un objeto de prueba. El hecho de que los métodos para hacer aserciones se declaran estáticos permite (re)usarlos fuera del ámbito de un objeto de prueba. Por último, la envoltura de funciones globales permite a los desarrolladores escribir menos caracteres (`assertTrue()` en lugar de `$this->assertTrue()` o `self::assertTrue()`).

13.2 assertArrayHasKey()

```
assertArrayHasKey(mixed $key, array $array[, string $message = ''])
```

Reporta un error identificado por el `$message` si el `$array` no tiene la llave `$key`.

`assertArrayNotHasKey()` es el inverso de este método y toma los mismos argumentos.

Example 13.1: Uso del método assertArrayHasKey()

```
<?php
use PHPUnit\Framework\TestCase;

class ArrayHasKeyTest extends TestCase
{
    public function testFailure()
    {
        $this->assertArrayHasKey('foo', ['bar' => 'baz']);
    }
}
```

```
$ phpunit ArrayHasKeyTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ArrayHasKeyTest::testFailure
Failed asserting that an array has the key 'foo'.

/home/sb/ArrayHasKeyTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.3 assertClassHasAttribute()

`assertClassHasAttribute(string $attributeName, string $className[, string $message = ''])`

Reporta un error identificado por el `$message` si `$className::attributeName` no existe.

`assertClassNotHasAttribute()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.2: Uso del método assertClassHasAttribute()

```
<?php
use PHPUnit\Framework\TestCase;

class ClassHasAttributeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertClassHasAttribute('foo', stdClass::class);
    }
}
```

```
$ phpunit ClassHasAttributeTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.
```

```
F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ClassHasAttributeTest::testFailure
Failed asserting that class "stdClass" has attribute "foo".

/home/sb/ClassHasAttributeTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.4 assertArraySubset()

`assertArraySubset(array $subset, array $array[, bool $strict = false, string $message = ''])`

Reporta un error identificado por el `$message` si el `$array` no contiene el `$subset`.

`$strict` es una bandera usada para comparar la identidad de objetos dentro de arreglos.

Example 13.3: Uso del método assertArraySubset()

```
<?php
use PHPUnit\Framework\TestCase;

class ArraySubsetTest extends TestCase
{
    public function testFailure()
    {
        $this->assertArraySubset(['config' => ['key-a', 'key-b']], ['config' => ['key-
↪a']]);
    }
}
```

```
$ phpunit ArraySubsetTest
PHPUnit 7.0.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) Epilog\EpilogTest::testNoFollowOption
Failed asserting that an array has the subset Array &0 (
    'config' => Array &1 (
        0 => 'key-a'
        1 => 'key-b'
    )
).

/home/sb/ArraySubsetTest.php:6
```

```
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.5 assertClassHasStaticAttribute()

`assertClassHasStaticAttribute(string $attributeName, string $className[, string $message = ''])`

Reporta un error identificado por el `$message` si `$className::attributeName` no existe.

`assertClassNotHasStaticAttribute()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.4: Uso del método assertClassHasStaticAttribute()

```php
<?php
use PHPUnit\Framework\TestCase;

class ClassHasStaticAttributeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertClassHasStaticAttribute('foo', stdClass::class);
    }
}
```

```
$ phpunit ClassHasStaticAttributeTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ClassHasStaticAttributeTest::testFailure
Failed asserting that class "stdClass" has static attribute "foo".

/home/sb/ClassHasStaticAttributeTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.6 assertContains()

`assertContains(mixed $needle, Iterator|array $haystack[, string $message = ''])`

Reporta un error identificado por el `$message` si `$needle` no es un elemento de `$haystack`.

`assertNotContains()` es el inverso de esta aserción y toma los mismos argumentos.

`assertAttributeContains()` y `assertAttributeNotContains()` son envoltorios convenientes que usan un atributo `public`, `protected` o `private` de una clase u objeto como el lugar donde buscar, *haystack*.

Example 13.5: Uso del método assertContains()

```
<?php
use PHPUnit\Framework\TestCase;

class ContainsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContains(4, [1, 2, 3]);
    }
}
```

```
$ phpunit ContainsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsTest::testFailure
Failed asserting that an array contains 4.

/home/sb/ContainsTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

`assertContains(string $needle, string $haystack[, string $message = '', boolean $ignoreCase = false])`

Reporta un error identificado por el `$message` si `$needle` no es una subcadena de caracteres de `$haystack`.

Si `$ignoreCase` es `true`, la prueba será insensible a mayúsculas y minúsculas.

Example 13.6: Uso del método assertContains()

```
<?php
use PHPUnit\Framework\TestCase;

class ContainsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContains('baz', 'foobar');
    }
}
```

```
$ phpunit ContainsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:
```

```
1) ContainsTest::testFailure
Failed asserting that 'foobar' contains "baz".

/home/sb/ContainsTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Example 13.7: Uso del método assertContains() con $ignoreCase

```
<?php
use PHPUnit\Framework\TestCase;

class ContainsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContains('foo', 'FooBar');
    }

    public function testOK()
    {
        $this->assertContains('foo', 'FooBar', '', true);
    }
}
```

```
$ phpunit ContainsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F.

Time: 0 seconds, Memory: 2.75Mb

There was 1 failure:

1) ContainsTest::testFailure
Failed asserting that 'FooBar' contains "foo".

/home/sb/ContainsTest.php:6

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

13.7 assertContainsOnly()

`assertContainsOnly(string $type, Iterator|array $haystack[, boolean $isNativeType = null, string $message = ''])`

Reporta un error identificado por el `$message` si `$haystack` no contiene solamente variables del tipo `$type`.

`$isNativeType` es una bandera usada para indicar si `$type` es un tipo nativo de PHP o no.

`assertNotContainsOnly()` es el inverso de esta aserción y toma los mismos argumentos.

`assertAttributeContainsOnly()` y `assertAttributeNotContainsOnly()` son envoltorios conve-

nientes que usan un atributo `public`, `protected` o `private` de una clases u objeto como el lugar donde buscar, *haystack*.

Example 13.8: Uso del método assertContainsOnly()

```
<?php
use PHPUnit\Framework\TestCase;

class ContainsOnlyTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContainsOnly('string', ['1', '2', 3]);
    }
}
```

```
$ phpunit ContainsOnlyTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsOnlyTest::testFailure
Failed asserting that Array (
    0 => '1'
    1 => '2'
    2 => 3
) contains only values of type "string".

/home/sb/ContainsOnlyTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.8 assertContainsOnlyInstancesOf()

`assertContainsOnlyInstancesOf(string $classname, Traversable|array $haystack[, string $message = ''])`

Reporta un error identificado por el `$message` si `$haystack` no contiene solamente instancias de la clase `$classname`.

Example 13.9: Uso del método assertContainsOnlyInstancesOf()

```
<?php
use PHPUnit\Framework\TestCase;

class ContainsOnlyInstancesOfTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContainsOnlyInstancesOf(
            Foo::class,
```

```
            [new Foo, new Bar, new Foo]
        );
    }
}
```

```
$ phpunit ContainsOnlyInstancesOfTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsOnlyInstancesOfTest::testFailure
Failed asserting that Array ([0]=> Bar Object(...)) is an instance of class "Foo".

/home/sb/ContainsOnlyInstancesOfTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.9 assertCount()

`assertCount($expectedCount, $haystack[, string $message = ''])`

Reporta un error identificado por el `$message` si el número de elementos en `$haystack` no es `$expectedCount`.

`assertNotCount()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.10: Uso del método assertCount()

```
<?php
use PHPUnit\Framework\TestCase;

class CountTest extends TestCase
{
    public function testFailure()
    {
        $this->assertCount(0, ['foo']);
    }
}
```

```
$ phpunit CountTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) CountTest::testFailure
Failed asserting that actual size 1 matches expected size 0.
```

```
/home/sb/CountTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.10 assertDirectoryExists()

`assertDirectoryExists(string $directory[, string $message = ''])`

Reporta un error identificado por el `$message` si la carpeta especificada por `$directory` no existe.

`assertDirectoryNotExists()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.11: Uso del método assertDirectoryExists()

```php
<?php
use PHPUnit\Framework\TestCase;

class DirectoryExistsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertDirectoryExists('/path/to/directory');
    }
}
```

```
$ phpunit DirectoryExistsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) DirectoryExistsTest::testFailure
Failed asserting that directory "/path/to/directory" exists.

/home/sb/DirectoryExistsTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.11 assertDirectoryIsReadable()

`assertDirectoryIsReadable(string $directory[, string $message = ''])`

Reporta un error identificado por el `$message` si la carpeta especificada en `$directory` no es una carpeta o no es legible.

`assertDirectoryNotIsReadable()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.12: Uso del método assertDirectoryIsReadable()

```
<?php
use PHPUnit\Framework\TestCase;

class DirectoryIsReadableTest extends TestCase
{
    public function testFailure()
    {
        $this->assertDirectoryIsReadable('/path/to/directory');
    }
}
```

```
$ phpunit DirectoryIsReadableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) DirectoryIsReadableTest::testFailure
Failed asserting that "/path/to/directory" is readable.

/home/sb/DirectoryIsReadableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.12 assertDirectoryIsWritable()

`assertDirectoryIsWritable(string $directory[, string $message = ''])`

Reporta un error identificado por el `$message` si la carpeta especificada en `$directory` no es una carpeta o no se puede escribir en ella.

`assertDirectoryNotIsWritable()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.13: Uso del método assertDirectoryIsWritable()

```
<?php
use PHPUnit\Framework\TestCase;

class DirectoryIsWritableTest extends TestCase
{
    public function testFailure()
    {
        $this->assertDirectoryIsWritable('/path/to/directory');
    }
}
```

```
$ phpunit DirectoryIsWritableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.
```

```
F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) DirectoryIsWritableTest::testFailure
Failed asserting that "/path/to/directory" is writable.

/home/sb/DirectoryIsWritableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.13 assertEmpty()

`assertEmpty(mixed $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si `$actual` no está vacío.

`assertNotEmpty()` es el inverso de esta aserción y toma los mismos argumentos.

`assertAttributeEmpty()` y `assertAttributeNotEmpty()` son envoltorios apropiados para ser usados con atributos de tipo `public`, `protected` o `private` de un objeto o una clase.

Example 13.14: Uso del método assertEmpty()

```php
<?php
use PHPUnit\Framework\TestCase;

class EmptyTest extends TestCase
{
    public function testFailure()
    {
        $this->assertEmpty(['foo']);
    }
}
```

```
$ phpunit EmptyTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) EmptyTest::testFailure
Failed asserting that an array is empty.

/home/sb/EmptyTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.14 assertEqualXMLStructure()

```
assertEqualXMLStructure(DOMElement $expectedElement, DOMElement
$actualElement[, boolean $checkAttributes = false, string $message = ''])
```

Reporta un error identificado por el `$message` si la estructura del DOMElement en `$actualElement` no es igual a la estructura XML del DOMElement en `$expectedElement`.

Example 13.15: Uso del método assertEqualXMLStructure()

```
<?php
use PHPUnit\Framework\TestCase;

class EqualXMLStructureTest extends TestCase
{
    public function testFailureWithDifferentNodeNames()
    {
        $expected = new DOMElement('foo');
        $actual = new DOMElement('bar');

        $this->assertEqualXMLStructure($expected, $actual);
    }

    public function testFailureWithDifferentNodeAttributes()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo bar="true" />');

        $actual = new DOMDocument;
        $actual->loadXML('<foo/>');

        $this->assertEqualXMLStructure(
          $expected->firstChild, $actual->firstChild, true
        );
    }

    public function testFailureWithDifferentChildrenCount()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/><bar/><bar/></foo>');

        $actual = new DOMDocument;
        $actual->loadXML('<foo><bar/></foo>');

        $this->assertEqualXMLStructure(
          $expected->firstChild, $actual->firstChild
        );
    }

    public function testFailureWithDifferentChildren()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/><bar/><bar/></foo>');

        $actual = new DOMDocument;
        $actual->loadXML('<foo><baz/><baz/><baz/></foo>');

        $this->assertEqualXMLStructure(
```

```
            $expected->firstChild, $actual->firstChild
        );
    }
}
```

```
$ phpunit EqualXMLStructureTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

FFFF

Time: 0 seconds, Memory: 5.75Mb

There were 4 failures:

1) EqualXMLStructureTest::testFailureWithDifferentNodeNames
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'foo'
+'bar'

/home/sb/EqualXMLStructureTest.php:9

2) EqualXMLStructureTest::testFailureWithDifferentNodeAttributes
Number of attributes on node "foo" does not match
Failed asserting that 0 matches expected 1.

/home/sb/EqualXMLStructureTest.php:22

3) EqualXMLStructureTest::testFailureWithDifferentChildrenCount
Number of child nodes of "foo" differs
Failed asserting that 1 matches expected 3.

/home/sb/EqualXMLStructureTest.php:35

4) EqualXMLStructureTest::testFailureWithDifferentChildren
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'bar'
+'baz'

/home/sb/EqualXMLStructureTest.php:48

FAILURES!
Tests: 4, Assertions: 8, Failures: 4.
```

13.15 assertEquals()

`assertEquals(mixed $expected, mixed $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si las dos variables `$expected` y `$actual` no son iguales.

`assertNotEquals()` es el inverso de esta aserción y toma los mismos argumentos.

`assertAttributeEquals()` y `assertAttributeNotEquals()` son los envoltorios apropiados cuando se usa atributos `public`, `protected` o `private` de una clase u objeto para el valor real

Example 13.16: Uso del método assertEquals()

```
<?php
use PHPUnit\Framework\TestCase;

class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertEquals(1, 0);
    }

    public function testFailure2()
    {
        $this->assertEquals('bar', 'baz');
    }

    public function testFailure3()
    {
        $this->assertEquals("foo\nbar\nbaz\n", "foo\nbah\nbaz\n");
    }
}
```

```
$ phpunit EqualsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

FFF

Time: 0 seconds, Memory: 5.25Mb

There were 3 failures:

1) EqualsTest::testFailure
Failed asserting that 0 matches expected 1.

/home/sb/EqualsTest.php:6

2) EqualsTest::testFailure2
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'bar'
+'baz'

/home/sb/EqualsTest.php:11

3) EqualsTest::testFailure3
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
 'foo
-bar
+bah
```

```
 baz
 '

/home/sb/EqualsTest.php:16

FAILURES!
Tests: 3, Assertions: 3, Failures: 3.
```

Comparaciones más especializadas se usan para especificar el tipo de argumentos para `$expected` y `$actual`, ver abajo.

`assertEquals(float $expected, float $actual[, string $message = '', float $delta = 0])`

Reporta un error identificado por el `$message` si la diferencia absoluta entre los números de tipo flotante `$expected` y `$actual` es mayor que el `$delta`. Si la diferencia absoluta entre los número de tipo flotante `$expected` y `$actual` es menor o *igual a* `$delta` entonces la aserción pasa.

Por favor lea «Todo lo que un Científico de la Computación Debe Saber sobre la Aritmética de Punto Flotante» para entender porqué `$delta` es necesario.

Example 13.17: Uso del método assertEquals() con número de punto flotante

```php
<?php
use PHPUnit\Framework\TestCase;

class EqualsTest extends TestCase
{
    public function testSuccess()
    {
        $this->assertEquals(1.0, 1.1, '', 0.1);
    }

    public function testFailure()
    {
        $this->assertEquals(1.0, 1.1);
    }
}
```

```
$ phpunit EqualsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

.F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that 1.1 matches expected 1.0.

/home/sb/EqualsTest.php:11

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

`assertEquals(DOMDocument $expected, DOMDocument $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si la forma canónica no comentada del documento XML representado por los dos objectos DOMDocument `$expected` y `$actual` no son iguales.

Example 13.18: Uso del método assertEquals() con objetos DOMDocument

```
<?php
use PHPUnit\Framework\TestCase;

class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/></foo>');

        $actual = new DOMDocument;
        $actual->loadXML('<bar><foo/></bar>');

        $this->assertEquals($expected, $actual);
    }
}
```

```
$ phpunit EqualsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
-<foo>
-  <bar/>
-</foo>
+<bar>
+  <foo/>
+</bar>

/home/sb/EqualsTest.php:12

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

`assertEquals(object $expected, object $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si los dos objetos `$expected` y `$actual` no tienen valores de atributos iguales.

Example 13.19: Uso del método assertEquals() con objetos

```
<?php
use PHPUnit\Framework\TestCase;
```

```
class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $expected = new stdClass;
        $expected->foo = 'foo';
        $expected->bar = 'bar';

        $actual = new stdClass;
        $actual->foo = 'bar';
        $actual->baz = 'bar';

        $this->assertEquals($expected, $actual);
    }
}
```

```
$ phpunit EqualsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that two objects are equal.
--- Expected
+++ Actual
@@ @@
 stdClass Object (
-    'foo' => 'foo'
-    'bar' => 'bar'
+    'foo' => 'bar'
+    'baz' => 'bar'
 )

/home/sb/EqualsTest.php:14

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

`assertEquals(array $expected, array $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si los dos arreglos `$expected` y `$actual` no son iguales.

Example 13.20: Uso del método assertEquals() con arreglos

```
<?php
use PHPUnit\Framework\TestCase;

class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertEquals(['a', 'b', 'c'], ['a', 'c', 'd']);
    }
}
```

```
$ phpunit EqualsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that two arrays are equal.
--- Expected
+++ Actual
@@ @@
 Array (
     0 => 'a'
-    1 => 'b'
-    2 => 'c'
+    1 => 'c'
+    2 => 'd'
 )

/home/sb/EqualsTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.16 assertFalse()

`assertFalse(bool $condition[, string $message = ''])`

Reporta un error identificado por el `$message` si `$condition` es `true`.

`assertNotFalse()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.21: Uso del método assertFalse()

```
<?php
use PHPUnit\Framework\TestCase;

class FalseTest extends TestCase
{
    public function testFailure()
    {
        $this->assertFalse(true);
    }
}
```

```
$ phpunit FalseTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:
```

```
1) FalseTest::testFailure
Failed asserting that true is false.

/home/sb/FalseTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.17 assertFileEquals()

`assertFileEquals(string $expected, string $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si the archivo especificado en `$expected` no tiene el mismo contenido que el archivo especificado en `$actual`.

`assertFileNotEquals()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.22: Uso del método assertFileEquals()

```php
<?php
use PHPUnit\Framework\TestCase;

class FileEqualsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertFileEquals('/home/sb/expected', '/home/sb/actual');
    }
}
```

```
$ phpunit FileEqualsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) FileEqualsTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'expected
+'actual
 '

/home/sb/FileEqualsTest.php:6

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

13.18 assertFileExists()

`assertFileExists(string $filename[, string $message = ''])`

Reporta un error identificado por el `$message` si the archivo especificado en `$filename` no existe.

`assertFileNotExists()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.23: Uso del método assertFileExists()

```
<?php
use PHPUnit\Framework\TestCase;

class FileExistsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertFileExists('/path/to/file');
    }
}
```

```
$ phpunit FileExistsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) FileExistsTest::testFailure
Failed asserting that file "/path/to/file" exists.

/home/sb/FileExistsTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.19 assertFileIsReadable()

`assertFileIsReadable(string $filename[, string $message = ''])`

Reporta un error identificado por el `$message` si el archivo especificado en `$filename` no es un archivo o no es legible.

`assertFileNotIsReadable()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.24: Uso del método assertFileIsReadable()

```
<?php
use PHPUnit\Framework\TestCase;

class FileIsReadableTest extends TestCase
{
    public function testFailure()
    {
```

```
        $this->assertFileIsReadable('/path/to/file');
    }
}
```

```
$ phpunit FileIsReadableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) FileIsReadableTest::testFailure
Failed asserting that "/path/to/file" is readable.

/home/sb/FileIsReadableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.20 assertFileIsWritable()

`assertFileIsWritable(string $filename[, string $message = ''])`

Reporta un error identificado por el `$message` si el archivo especificado en `$filename` no es un archivo o no se puede escribir en él.

`assertFileNotIsWritable()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.25: Uso del método assertFileIsWritable()

```
<?php
use PHPUnit\Framework\TestCase;

class FileIsWritableTest extends TestCase
{
    public function testFailure()
    {
        $this->assertFileIsWritable('/path/to/file');
    }
}
```

```
$ phpunit FileIsWritableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) FileIsWritableTest::testFailure
Failed asserting that "/path/to/file" is writable.
```

```
/home/sb/FileIsWritableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.21 assertGreaterThan()

`assertGreaterThan(mixed $expected, mixed $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si el valor de `$actual` no es mayor que el valor de `$expected`.

`assertAttributeGreaterThan()` es un envoltorio conveniente que usa el atributo `public`, `protected` o `private` de una clase u objeto como el valor real..

Example 13.26: Uso del método assertGreaterThan()

```
<?php
use PHPUnit\Framework\TestCase;

class GreaterThanTest extends TestCase
{
    public function testFailure()
    {
        $this->assertGreaterThan(2, 1);
    }
}
```

```
$ phpunit GreaterThanTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) GreaterThanTest::testFailure
Failed asserting that 1 is greater than 2.

/home/sb/GreaterThanTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.22 assertGreaterThanOrEqual()

`assertGreaterThanOrEqual(mixed $expected, mixed $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si el valor de `$actual` no es mayor o igual al valor de `$expected`.

`assertAttributeGreaterThanOrEqual()` es un envoltorio conveniente que usa el atributo `public`, `protected` o `private` de una clase u objeto como el valor real.

Example 13.27: Uso del método assertGreaterThanOrEqual()

```
<?php
use PHPUnit\Framework\TestCase;

class GreatThanOrEqualTest extends TestCase
{
    public function testFailure()
    {
        $this->assertGreaterThanOrEqual(2, 1);
    }
}
```

```
$ phpunit GreaterThanOrEqualTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) GreatThanOrEqualTest::testFailure
Failed asserting that 1 is equal to 2 or is greater than 2.

/home/sb/GreaterThanOrEqualTest.php:6

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

13.23 assertInfinite()

`assertInfinite(mixed $variable[, string $message = ''])`

Reporta un error identificado por el `$message` si `$variable` es not `INF`.

`assertFinite()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.28: Uso del método assertInfinite()

```
<?php
use PHPUnit\Framework\TestCase;

class InfiniteTest extends TestCase
{
    public function testFailure()
    {
        $this->assertInfinite(1);
    }
}
```

```
$ phpunit InfiniteTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F
```

```
Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) InfiniteTest::testFailure
Failed asserting that 1 is infinite.

/home/sb/InfiniteTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.24 assertInstanceOf()

`assertInstanceOf($expected, $actual[, $message = ''])`

Reporta un error identificado por el `$message` si `$actual` no es una instancia de `$expected`.

`assertNotInstanceOf()` es el inverso de esta aserción y toma los mismos argumentos.

`assertAttributeInstanceOf()` son `assertAttributeNotInstanceOf()` son envoltorios convenientes que se pueden aplicar a un atributo `public`, `protected` o `private` de una clase u objeto.

Example 13.29: Uso del método assertInstanceOf()

```
<?php
use PHPUnit\Framework\TestCase;

class InstanceOfTest extends TestCase
{
    public function testFailure()
    {
        $this->assertInstanceOf(RuntimeException::class, new Exception);
    }
}
```

```
$ phpunit InstanceOfTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) InstanceOfTest::testFailure
Failed asserting that Exception Object (...) is an instance of class "RuntimeException
↪".

/home/sb/InstanceOfTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.25 assertInternalType()

`assertInternalType($expected, $actual[, $message = ''])`

Reporta un error identificado por el `$message` si `$actual` no es del tipo `$expected`.

`assertNotInternalType()` es el inverso de esta aserción y toma los mismos argumentos.

`assertAttributeInternalType()` y `assertAttributeNotInternalType()` son envoltorios convenientes que se pueden aplicar a un atributo `public`, `protected` o `private` de una clase u objeto.

Example 13.30: Uso del método assertInternalType()

```
<?php
use PHPUnit\Framework\TestCase;

class InternalTypeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertInternalType('string', 42);
    }
}
```

```
$ phpunit InternalTypeTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) InternalTypeTest::testFailure
Failed asserting that 42 is of type "string".

/home/sb/InternalTypeTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.26 assertIsReadable()

`assertIsReadable(string $filename[, string $message = ''])`

Reporta un error identificado por el `$message` si el archivo o carpeta especificada en `$filename` no se puede leer.

`assertNotIsReadable()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.31: Uso del método assertIsReadable()

```
<?php
use PHPUnit\Framework\TestCase;

class IsReadableTest extends TestCase
{
    public function testFailure()
```

```
    {
        $this->assertIsReadable('/path/to/unreadable');
    }
}
```

```
$ phpunit IsReadableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) IsReadableTest::testFailure
Failed asserting that "/path/to/unreadable" is readable.

/home/sb/IsReadableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.27 assertIsWritable()

`assertIsWritable(string $filename[, string $message = ''])`

Reporta un error identificado por el `$message` si el archivo o carpeta especificada en `$filename` no se puede escribir.

`assertNotIsWritable()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.32: Uso del método assertIsWritable()

```
<?php
use PHPUnit\Framework\TestCase;

class IsWritableTest extends TestCase
{
    public function testFailure()
    {
        $this->assertIsWritable('/path/to/unwritable');
    }
}
```

```
$ phpunit IsWritableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) IsWritableTest::testFailure
Failed asserting that "/path/to/unwritable" is writable.
```

```
/home/sb/IsWritableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.28 assertJsonFileEqualsJsonFile()

`assertJsonFileEqualsJsonFile(mixed $expectedFile, mixed $actualFile[, string $message = ''])`

Reporta un error identificado por el `$message` si el valor de `$actualFile` no coincide con el valor de `$expectedFile`.

Example 13.33: Uso del método assertJsonFileEqualsJsonFile()

```
<?php
use PHPUnit\Framework\TestCase;

class JsonFileEqualsJsonFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertJsonFileEqualsJsonFile(
          'path/to/fixture/file', 'path/to/actual/file');
    }
}
```

```
$ phpunit JsonFileEqualsJsonFileTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) JsonFileEqualsJsonFile::testFailure
Failed asserting that '{"Mascot":"Tux"}' matches JSON string "["Mascott", "Tux", "OS",
↪ "Linux"]".

/home/sb/JsonFileEqualsJsonFileTest.php:5

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

13.29 assertJsonStringEqualsJsonFile()

`assertJsonStringEqualsJsonFile(mixed $expectedFile, mixed $actualJson[, string $message = ''])`

Reporta un error identificado por el `$message` si el valor de `$actualJson` no coincide con el valor de `$expectedFile`.

Example 13.34: Uso del método assertJsonStringEqualsJsonFile()

```
<?php
use PHPUnit\Framework\TestCase;

class JsonStringEqualsJsonFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertJsonStringEqualsJsonFile(
            'path/to/fixture/file', json_encode(['Mascot' => 'ux'])
        );
    }
}
```

```
$ phpunit JsonStringEqualsJsonFileTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) JsonStringEqualsJsonFile::testFailure
Failed asserting that '{"Mascot":"ux"}' matches JSON string "{"Mascott":"Tux"}".

/home/sb/JsonStringEqualsJsonFileTest.php:5

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

13.30 assertJsonStringEqualsJsonString()

`assertJsonStringEqualsJsonString(mixed $expectedJson, mixed $actualJson[, string $message = ''])`

Reporta un error identificado por el `$message` si el valor de `$actualJson` no coincide con el valor de `$expectedJson`.

Example 13.35: Uso del método assertJsonStringEqualsJsonString()

```
<?php
use PHPUnit\Framework\TestCase;

class JsonStringEqualsJsonStringTest extends TestCase
{
    public function testFailure()
    {
        $this->assertJsonStringEqualsJsonString(
            json_encode(['Mascot' => 'Tux']),
            json_encode(['Mascot' => 'ux'])
        );
    }
}
```

```
$ phpunit JsonStringEqualsJsonStringTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) JsonStringEqualsJsonStringTest::testFailure
Failed asserting that two objects are equal.
--- Expected
+++ Actual
@@ @@
 stdClass Object (
 -    'Mascot' => 'Tux'
 +    'Mascot' => 'ux'
)

/home/sb/JsonStringEqualsJsonStringTest.php:5

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

13.31 assertLessThan()

`assertLessThan(mixed $expected, mixed $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si el valor de `$actual` no es menor que el valor de `$expected`.

`assertAttributeLessThan()` es un envoltorio conveniente que usa el atributo `public`, `protected` o `private` de una clase u objeto como el valor real.

Example 13.36: Uso del método assertLessThan()

```php
<?php
use PHPUnit\Framework\TestCase;

class LessThanTest extends TestCase
{
    public function testFailure()
    {
        $this->assertLessThan(1, 2);
    }
}
```

```
$ phpunit LessThanTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) LessThanTest::testFailure
```

```
Failed asserting that 2 is less than 1.

/home/sb/LessThanTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.32 assertLessThanOrEqual()

`assertLessThanOrEqual(mixed $expected, mixed $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si el valor de `$actual` no es menor o igual que el valor de `$expected`.

`assertAttributeLessThanOrEqual()` es un envoltorio conveniente que usa el atributo `public`, `protected` o `private` de una clase u objeto como el valor real.

Example 13.37: Uso del método assertLessThanOrEqual()

```
<?php
use PHPUnit\Framework\TestCase;

class LessThanOrEqualTest extends TestCase
{
    public function testFailure()
    {
        $this->assertLessThanOrEqual(1, 2);
    }
}
```

```
$ phpunit LessThanOrEqualTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) LessThanOrEqualTest::testFailure
Failed asserting that 2 is equal to 1 or is less than 1.

/home/sb/LessThanOrEqualTest.php:6

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

13.33 assertNan()

`assertNan(mixed $variable[, string $message = ''])`

Reporta un error identificado por el `$message` si `$variable` no es `NAN`.

Example 13.38: Uso del método assertNan()

```
<?php
use PHPUnit\Framework\TestCase;

class NanTest extends TestCase
{
    public function testFailure()
    {
        $this->assertNan(1);
    }
}
```

```
$ phpunit NanTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) NanTest::testFailure
Failed asserting that 1 is nan.

/home/sb/NanTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.34 assertNull()

`assertNull(mixed $variable[, string $message = ''])`

Reporta un error identificado por el `$message` si `$variable` no es `null`.

`assertNotNull()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.39: Uso del método assertNull()

```
<?php
use PHPUnit\Framework\TestCase;

class NullTest extends TestCase
{
    public function testFailure()
    {
        $this->assertNull('foo');
    }
}
```

```
$ phpunit NotNullTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F
```

```
Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) NullTest::testFailure
Failed asserting that 'foo' is null.

/home/sb/NotNullTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.35 assertObjectHasAttribute()

`assertObjectHasAttribute(string $attributeName, object $object[, string $message = ''])`

Reporta un error identificado por el `$message` si `$object->attributeName` no existe.

`assertObjectNotHasAttribute()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.40: Uso del método assertObjectHasAttribute()

```php
<?php
use PHPUnit\Framework\TestCase;

class ObjectHasAttributeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertObjectHasAttribute('foo', new stdClass);
    }
}
```

```
$ phpunit ObjectHasAttributeTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ObjectHasAttributeTest::testFailure
Failed asserting that object of class "stdClass" has attribute "foo".

/home/sb/ObjectHasAttributeTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.36 assertRegExp()

`assertRegExp(string $pattern, string $string[, string $message = ''])`

Reporta un error identificado por el `$message` si `$string` no coincide con la expresión regular `$pattern`.

`assertNotRegExp()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.41: Uso del método assertRegExp()

```
<?php
use PHPUnit\Framework\TestCase;

class RegExpTest extends TestCase
{
    public function testFailure()
    {
        $this->assertRegExp('/foo/', 'bar');
    }
}
```

```
$ phpunit RegExpTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) RegExpTest::testFailure
Failed asserting that 'bar' matches PCRE pattern "/foo/".

/home/sb/RegExpTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.37 assertStringMatchesFormat()

`assertStringMatchesFormat(string $format, string $string[, string $message = ''])`

Reporta un error identificado por el `$message` si el `$string` no coincide con el formato dado en `$format`.

`assertStringNotMatchesFormat()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.42: Uso del método assertStringMatchesFormat()

```
<?php
use PHPUnit\Framework\TestCase;

class StringMatchesFormatTest extends TestCase
{
    public function testFailure()
    {
```

```
        $this->assertStringMatchesFormat('%i', 'foo');
    }
}
```

```
$ phpunit StringMatchesFormatTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) StringMatchesFormatTest::testFailure
Failed asserting that 'foo' matches PCRE pattern "/^[+-]?\d+$/s".

/home/sb/StringMatchesFormatTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

La cadena de caracteres que designa el formato puede contener los siguientes sustitutos:

- `%e`: Representa el separador de carpetas, por ejemplo `/` en GNU/Linux.
- `%s`: Uno o más de cualquier carácter (o espacio en blanco) excepto el carácter de fin de línea.
- `%S`: Cero o más de cualquier carácter (o espacio en blanco) excepto el carácter de fin de línea.
- `%a`: Uno o más de cualquier carácter (o espacio en blanco) incluyendo el carácter de fin de línea.
- `%A`: Cero o más de cualquier carácter (o espacio en blanco) incluyendo el carácter de fin de línea.
- `%w`: Cero o más caracteres de espacio en blanco.
- `%i`: Un valor entero con signo, por ejemplo `+3142`, `-3142`.
- `%d`: Un valor entero sin signo, por ejemplo `123456`.
- `%x`: Uno o más caracteres hexadecimales. Esto es, caracteres en el rango de `0-9`, `a-f` y `A-F`.
- `%f`: Un número de punto flotante, por ejemplo: `3.142`, `-3.142`, `3.142E-10` o `3.142e+10`.
- `%c`: Un solo carácter de cualquier tipo.
- `%%`: Un carácter literal de porcentaje: `%`.

13.38 assertStringMatchesFormatFile()

`assertStringMatchesFormatFile(string $formatFile, string $string[, string $message = ''])`

Reporta un error identificado por el `$message` si el `$string` no coincide con el contenido de `$formatFile`.

`assertStringNotMatchesFormatFile()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.43: Uso del método assertStringMatchesFormatFile()

```
<?php
use PHPUnit\Framework\TestCase;
```

```
class StringMatchesFormatFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringMatchesFormatFile('/path/to/expected.txt', 'foo');
    }
}
```

```
$ phpunit StringMatchesFormatFileTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) StringMatchesFormatFileTest::testFailure
Failed asserting that 'foo' matches PCRE pattern "/^[+-]?\d+
$/s".

/home/sb/StringMatchesFormatFileTest.php:6

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

13.39 assertSame()

`assertSame(mixed $expected, mixed $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si las dos variables `$expected` y `$actual` no tienen el mismo tipo y valor.

`assertNotSame()` es el inverso de esta aserción y toma los mismos argumentos.

`assertAttributeSame()` y `assertAttributeNotSame()` son envoltorios convenientes que usan el atributo `public`, `protected` o `private` de una clase u objeto como el valor real.

Example 13.44: Uso del método assertSame()

```
<?php
use PHPUnit\Framework\TestCase;

class SameTest extends TestCase
{
    public function testFailure()
    {
        $this->assertSame('2204', 2204);
    }
}
```

```
$ phpunit SameTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F
```

```
Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) SameTest::testFailure
Failed asserting that 2204 is identical to '2204'.

/home/sb/SameTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

`assertSame(object $expected, object $actual[, string $message = ''])`

Reporta un error identificado por el `$message` si las dos variables `$expected` y `$actual` no hacen referencia al mismo objeto.

Example 13.45: Uso del método assertSame() con objetos

```
<?php
use PHPUnit\Framework\TestCase;

class SameTest extends TestCase
{
    public function testFailure()
    {
        $this->assertSame(new stdClass, new stdClass);
    }
}
```

```
$ phpunit SameTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) SameTest::testFailure
Failed asserting that two variables reference the same object.

/home/sb/SameTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.40 assertStringEndsWith()

`assertStringEndsWith(string $suffix, string $string[, string $message = ''])`

Reporta un error identificado por el `$message` si `$string` no termina en `$suffix`.

`assertStringEndsNotWith()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.46: Uso del método assertStringEndsWith()

```
<?php
use PHPUnit\Framework\TestCase;

class StringEndsWithTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringEndsWith('suffix', 'foo');
    }
}
```

```
$ phpunit StringEndsWithTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 1 second, Memory: 5.00Mb

There was 1 failure:

1) StringEndsWithTest::testFailure
Failed asserting that 'foo' ends with "suffix".

/home/sb/StringEndsWithTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.41 assertStringEqualsFile()

`assertStringEqualsFile(string $expectedFile, string $actualString[, string $message = ''])`

Reporta un error identificado por el `$message` si el archivo especificado en `$expectedFile` no tiene a `$actualString` como su contenido.

`assertStringNotEqualsFile()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.47: Uso del método assertStringEqualsFile()

```
<?php
use PHPUnit\Framework\TestCase;

class StringEqualsFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringEqualsFile('/home/sb/expected', 'actual');
    }
}
```

```
$ phpunit StringEqualsFileTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.
```

```
F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) StringEqualsFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'expected
-'
+'actual'

/home/sb/StringEqualsFileTest.php:6

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

13.42 assertStringStartsWith()

`assertStringStartsWith(string $prefix, string $string[, string $message = ''])`

Reporta un error identificado por el `$message` si el `$string` no comienza con el `$prefix`.

`assertStringStartsNotWith()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.48: Uso del método assertStringStartsWith()

```php
<?php
use PHPUnit\Framework\TestCase;

class StringStartsWithTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringStartsWith('prefix', 'foo');
    }
}
```

```
$ phpunit StringStartsWithTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) StringStartsWithTest::testFailure
Failed asserting that 'foo' starts with "prefix".

/home/sb/StringStartsWithTest.php:6
```

```
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.43 assertThat()

Aserciones más complejas se pueden formular usando las clases `PHPUnit\Framework\Constraint`. Ellas se pueden evaluar usando el método `assertThat()`. El Example 13.49 muestra como las restricciones `logicalNot()` y `equalTo()` se pueden usar para expresar la aserción `assertNotEquals()`.

`assertThat(mixed $value, PHPUnit\Framework\Constraint $constraint[, $message = ''])`

Reporta un error identificado por el `$message` si el `$value` no coincide con el `$constraint`.

Example 13.49: Uso del método assertThat()

```
<?php
use PHPUnit\Framework\TestCase;

class BiscuitTest extends TestCase
{
    public function testEquals()
    {
        $theBiscuit = new Biscuit('Ginger');
        $myBiscuit  = new Biscuit('Ginger');

        $this->assertThat(
          $theBiscuit,
          $this->logicalNot(
            $this->equalTo($myBiscuit)
          )
        );
    }
}
```

La Table 13.1 muestra las clases disponibles en `PHPUnit\Framework\Constraint`.

Constraint
`PHPUnit\Framework\Constraint\Attribute attribute(PHPUnit\Framework\Constraint $constraint,`
`PHPUnit\Framework\Constraint\IsAnything anything()`
`PHPUnit\Framework\Constraint\ArrayHasKey arrayHasKey(mixed $key)`
`PHPUnit\Framework\Constraint\TraversableContains contains(mixed $value)`
`PHPUnit\Framework\Constraint\TraversableContainsOnly containsOnly(string $type)`
`PHPUnit\Framework\Constraint\TraversableContainsOnly containsOnlyInstancesOf(string $class`
`PHPUnit\Framework\Constraint\IsEqual equalTo($value, $delta = 0, $maxDepth = 10)`
`PHPUnit\Framework\Constraint\Attribute attributeEqualTo($attributeName, $value, $delta =`
`PHPUnit\Framework\Constraint\DirectoryExists directoryExists()`
`PHPUnit\Framework\Constraint\FileExists fileExists()`
`PHPUnit\Framework\Constraint\IsReadable isReadable()`
`PHPUnit\Framework\Constraint\IsWritable isWritable()`
`PHPUnit\Framework\Constraint\GreaterThan greaterThan(mixed $value)`

Table 1

Constraint
PHPUnit\Framework\Constraint\Or greaterThanOrEqual(mixed $value)
PHPUnit\Framework\Constraint\ClassHasAttribute classHasAttribute(string $attributeName)
PHPUnit\Framework\Constraint\ClassHasStaticAttribute classHasStaticAttribute(string $attr
PHPUnit\Framework\Constraint\ObjectHasAttribute objecthasAttribute(string $attributeName)
PHPUnit\Framework\Constraint\IsIdentical identicalTo(mixed $value)
PHPUnit\Framework\Constraint\IsFalse isFalse()
PHPUnit\Framework\Constraint\IsInstanceOf isInstanceOf(string $className)
PHPUnit\Framework\Constraint\IsNull isNull()
PHPUnit\Framework\Constraint\IsTrue isTrue()
PHPUnit\Framework\Constraint\IsType isType(string $type)
PHPUnit\Framework\Constraint\LessThan lessThan(mixed $value)
PHPUnit\Framework\Constraint\Or lessThanOrEqual(mixed $value)
logicalAnd()
logicalNot(PHPUnit\Framework\Constraint $constraint)
logicalOr()
logicalXor()
PHPUnit\Framework\Constraint\PCREMatch matchesRegularExpression(string $pattern)
PHPUnit\Framework\Constraint\StringContains stringContains(string $string, bool $case)
PHPUnit\Framework\Constraint\StringEndsWith stringEndsWith(string $suffix)
PHPUnit\Framework\Constraint\StringStartsWith stringStartsWith(string $prefix)

13.44 assertTrue()

`assertTrue(bool $condition[, string $message = ''])`

Reporta un error identificado por el `$message` si `$condition` es `false`.

`assertNotTrue()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.50: Uso del método assertTrue()

```
<?php
use PHPUnit\Framework\TestCase;

class TrueTest extends TestCase
{
    public function testFailure()
    {
        $this->assertTrue(false);
    }
}
```

```
$ phpunit TrueTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) TrueTest::testFailure
```

```
Failed asserting that false is true.

/home/sb/TrueTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

13.45 assertXmlFileEqualsXmlFile()

`assertXmlFileEqualsXmlFile(string $expectedFile, string $actualFile[, string $message = ''])`

Reporta un error identificado por el `$message` si the XML document in `$actualFile` no es igual al documento XML en `$expectedFile`.

`assertXmlFileNotEqualsXmlFile()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.51: Uso del método assertXmlFileEqualsXmlFile()

```
<?php
use PHPUnit\Framework\TestCase;

class XmlFileEqualsXmlFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertXmlFileEqualsXmlFile(
          '/home/sb/expected.xml', '/home/sb/actual.xml');
    }
}
```

```
$ phpunit XmlFileEqualsXmlFileTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) XmlFileEqualsXmlFileTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
 <foo>
-  <bar/>
+  <baz/>
 </foo>

/home/sb/XmlFileEqualsXmlFileTest.php:7

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

13.46 assertXmlStringEqualsXmlFile()

`assertXmlStringEqualsXmlFile(string $expectedFile, string $actualXml[, string $message = ''])`

Reporta un error identificado por el `$message` si the XML document in `$actualXml` is not equal to the XML document in `$expectedFile`.

`assertXmlStringNotEqualsXmlFile()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.52: Uso del método assertXmlStringEqualsXmlFile()

```
<?php
use PHPUnit\Framework\TestCase;

class XmlStringEqualsXmlFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertXmlStringEqualsXmlFile(
          '/home/sb/expected.xml', '<foo><baz/></foo>');
    }
}
```

```
$ phpunit XmlStringEqualsXmlFileTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) XmlStringEqualsXmlFileTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
 <foo>
-  <bar/>
+  <baz/>
 </foo>

/home/sb/XmlStringEqualsXmlFileTest.php:7

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

13.47 assertXmlStringEqualsXmlString()

`assertXmlStringEqualsXmlString(string $expectedXml, string $actualXml[, string $message = ''])`

Reporta un error identificado por el `$message` si the XML document in `$actualXml` is not equal to the XML document in `$expectedXml`.

`assertXmlStringNotEqualsXmlString()` es el inverso de esta aserción y toma los mismos argumentos.

Example 13.53: Uso del método assertXmlStringEqualsXmlString()

```
<?php
use PHPUnit\Framework\TestCase;

class XmlStringEqualsXmlStringTest extends TestCase
{
    public function testFailure()
    {
        $this->assertXmlStringEqualsXmlString(
          '<foo><bar/></foo>', '<foo><baz/></foo>');
    }
}
```

```
$ phpunit XmlStringEqualsXmlStringTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) XmlStringEqualsXmlStringTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
 <foo>
-  <bar/>
+  <baz/>
 </foo>

/home/sb/XmlStringEqualsXmlStringTest.php:7

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

CAPÍTULO 14

Anotaciones

Una anotación es una forma especial de meta datos sintácticos que se pueden agregar al código fuente de algún lenguaje de programación. Aunque PHP no tiene entre sus características la anotación de código fuente, el uso de etiquetas como `@annotation arguments` en un bloque de documentación se ha establecido en la comunidad PHP como el modo de anotar el código fuente. En PHP los bloques de documentación son reflexivos: se puede acceder a ellos por medio del método `getDocComment()` de la «API Reflection» a nivel de función, clase, método y atributo. Aplicaciones como PHPUnit usan esta información en tiempo de ejecución para configurar su comportamiento.

Nota

Un comentario de documentación en PHP debe comenzar con `/**` y terminar con `*/`. Las anotaciones en cualquier otro estilo de comentarios serán ignoradas.

Este apéndice muestra toda la variedad de anotaciones soportada por PHPUnit.

14.1 @author

La anotación `@author` es un alias de la anotación `@group` (ver *@group*) y permite filtrar las pruebas en base a sus autores.

14.2 @after

La anotación `@after` se puede usar para especificar métodos que deben ser llamados después de todos los métodos de prueba de la clase de casos de prueba.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
```

```
     * @after
     */
    public function tearDownSomeFixtures()
    {
        // ...
    }

    /**
     * @after
     */
    public function tearDownSomeOtherFixtures()
    {
        // ...
    }
}
```

14.3 @afterClass

La anotación `@afterClass` se puede usar para especificar métodos estáticos que limpian los ambientes compartidos. Estos métodos se llaman después de que todos los métodos de prueba de una clase de prueba se han ejecutado.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @afterClass
     */
    public static function tearDownSomeSharedFixtures()
    {
        // ...
    }

    /**
     * @afterClass
     */
    public static function tearDownSomeOtherSharedFixtures()
    {
        // ...
    }
}
```

14.4 @backupGlobals

Las operaciones de respaldo y restauración para las variables globales se pueden desactivar completamente para todas las pruebas de una clase de casos de prueba, de la siguiente manera:

```
use PHPUnit\Framework\TestCase;

/**
 * @backupGlobals disabled
 */
class MyTest extends TestCase
```

```
{
    // ...
}
```

Además, la anotación `@backupGlobals` se puede usar a nivel de método de prueba. Esto permite una configuración «de grano fino» sobre las operaciones de respaldo y restauración:

```
use PHPUnit\Framework\TestCase;

/**
 * @backupGlobals disabled
 */
class MyTest extends TestCase
{
    /**
     * @backupGlobals enabled
     */
    public function testThatInteractsWithGlobalVariables()
    {
        // ...
    }
}
```

14.5 @backupStaticAttributes

La anotación `@backupStaticAttributes` se puede usar para respaldar todos los valores de las propiedades estáticas en todas las clases declaradas antes de cada prueba y restaurarlos después. Esta anotación se puede usar a nivel de clase de caso de prueba o de método de prueba.

```
use PHPUnit\Framework\TestCase;

/**
 * @backupStaticAttributes enabled
 */
class MyTest extends TestCase
{
    /**
     * @backupStaticAttributes disabled
     */
    public function testThatInteractsWithStaticAttributes()
    {
        // ...
    }
}
```

Nota

La anotación `@backupStaticAttributes` está limitada por PHP y en algunas circunstancias puede causar que valores estáticos no deseados persistan y se filtren en las pruebas siguientes.

Para más detalles podemos ver *Estado Global*.

14.6 @before

La anotación `@before` se puede usar para especificar métodos que se deben llamar antes de los métodos de prueba de la clase de casos de prueba.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @before
     */
    public function setupSomeFixtures()
    {
        // ...
    }

    /**
     * @before
     */
    public function setupSomeOtherFixtures()
    {
        // ...
    }
}
```

14.7 @beforeClass

La anotación `@beforeClass` se puede usar para especificar métodos estáticos que limpian los ambientes compartidos. Estos métodos se llaman antes de cualquier método de prueba de una clase de prueba.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @beforeClass
     */
    public static function setUpSomeSharedFixtures()
    {
        // ...
    }

    /**
     * @beforeClass
     */
    public static function setUpSomeOtherSharedFixtures()
    {
        // ...
    }
}
```

14.8 @codeCoverageIgnore*

Las anotaciones `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` y `@codeCoverageIgnoreEnd` se pueden usar para excluir líneas de código del análisis de cobertura de código.

Para conocer el uso de estas anotaciones podemos ver *Ignorar Bloques de Código*.

14.9 @covers

La anotación `@covers` se puede usar en el código de prueba para especificar el o los métodos que queremos probar dentro de un método de prueba :

```
/**
 * @covers BankAccount::getBalance
 */
public function testBalanceIsInitiallyZero()
{
    $this->assertSame(0, $this->ba->getBalance());
}
```

Si esta anotación se usa, solo se considerará la información de cobertura de código para el o los métodos especificados.

El Table 14.1 muestra la sintaxis de la anotación `@covers`.

Table 14.1: Anotaciones que permite especificar que métodos son cubiertos por una prueba

Anotación	Descripción
`@covers ClassName::methodName`	Indica que el método de prueba anotado cubre el método especificado.
`@covers ClassName`	Indica que el método de prueba anotado cubre todos los métodos de una clase dada.
`@covers ClassName<extended>`	Indica que el método de prueba anotado cubre todos los métodos de una clase dada y sus clases o interfaces padre.
`@covers ClassName::<public>`	Indica que el método de prueba anotado cubre todos los métodos públicos de una clase dada.
`@covers ClassName::<protected>`	Indica que el método de prueba anotado cubre todos los métodos protegidos de una clase dada.
`@covers ClassName::<private>`	Indica que el método de prueba anotado cubre todos los métodos privados de una clase dada.
`@covers ClassName::<!public>`	Indica que el método de prueba anotado cubre todos los métodos que no son públicos de una clase dada.
`@covers ClassName::<!protected>`	Indica que el método de prueba anotado cubre todos los métodos que no son protegidos de una clase dada.
`@covers ClassName::<!private>`	Indica que el método de prueba anotado cubre todos los métodos que nos son privados de una clase dada.
`@covers ::functionName`	Indica que el método de prueba anotado cubre la función global especificada.

14.10 @coversDefaultClass

La anotación `@coversDefaultClass` se puede usar para especificar un espacio de nombres o un nombre de clase por defecto. Para que nombres muy largos no necesiten ser repetido para cada anotación `@covers`. Ver Example 14.1.

Example 14.1: Usar @coversDefaultClass para acortar las anotaciones

```
<?php
use PHPUnit\Framework\TestCase;

/**
 * @coversDefaultClass \Foo\CoveredClass
 */
class CoversDefaultClassTest extends TestCase
{
    /**
     * @covers ::publicMethod
     */
    public function testSomething()
    {
        $o = new Foo\CoveredClass;
        $o->publicMethod();
    }
}
```

14.11 @coversNothing

La anotación `@coversNothing` se puede usar en el código de prueba para especificar que el caso de prueba anotado no se guarde en la información de cobertura de código.

Esta anotación se puede usar para las pruebas de integración. Para un ejemplo podemos ver *Una prueba que especifica que ningún método será cubierto*

Esta anotación se puede usar a nivel de clase o de método y sobrescribirá cualquier etiqueta `@covers`.

14.12 @dataProvider

Un método de prueba puede aceptar argumentos arbitrarios. Estos argumentos se proveen por uno o más métodos proveedores de datos (`provider()` en *Usar un proveedor de datos que regresa un arreglo de arreglos*). El método proveedor de datos a ser usado se especifica usando la anotación `@dataProvider`.

Para más detalles podemos ver *Proveedores de Datos*.

14.13 @depends

PHPUnit soporta la declaración de dependencia explicitas entre métodos de pruebas. Estas dependencias no definen el orden en que los métodos de prueba serán ejecutados pero ellos permiten regresar una instancia del ambiente de pruebas desde el productor y pasarla al consumidor. El *Usar la anotación @depends para expresar dependencias* muestra como usar la anotación `@depends` para expresar dependencias entre métodos de prueba.

Ver *Dependencia de Pruebas* para más detalles.

14.14 @doesNotPerformAssertions

Evita que una prueba que no ejecuta aserciones sea considerado riesgoso.

14.15 @expectedException

Usar el método expectException() shows how to use the `@expectedException` annotation to test whether an exception is thrown inside the tested code.

See *Probar Excepciones* for more details.

14.16 @expectedExceptionCode

La anotación `@expectedExceptionCode` en conjunción con `@expectedException` permite hacer aserciones sobre el código de error lanzado por una excepción, esto nos permite seleccionar una excepción en específico.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @expectedException     MyException
     * @expectedExceptionCode 20
     */
    public function testExceptionHasErrorCode20()
    {
        throw new MyException('Some Message', 20);
    }
}
```

Para probar con facilidad y reducir la duplicidad se puede usar un atajo para especificar una constante de clase como `@expectedExceptionCode` usando la sintaxis «`@expectedExceptionCode ClassName::CONST`».

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
      * @expectedException     MyException
      * @expectedExceptionCode MyClass::ERRORCODE
      */
    public function testExceptionHasErrorCode20()
    {
      throw new MyException('Some Message', 20);
    }
}
class MyClass
{
    const ERRORCODE = 20;
}
```

14.17 @expectedExceptionMessage

La anotación `@expectedExceptionMessage` funciona de manera similar a la anotación `@expectedExceptionCode` y permite hacer una aserción sobre el mensaje de error de una excepción.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @expectedException        MyException
     * @expectedExceptionMessage Some Message
     */
    public function testExceptionHasRightMessage()
    {
        throw new MyException('Some Message', 20);
    }
}
```

El mensaje esperado puede ser una sub cadena de caracteres del «Message» de la excepción. Esta característica puede ser útil para afirmar solo un determinado nombre o parámetro que está presente en el mensaje de la excepción y no considerar todo el mensaje de la excepción.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
     /**
      * @expectedException        MyException
      * @expectedExceptionMessage broken
      */
     public function testExceptionHasRightMessage()
     {
         $param = "broken";
         throw new MyException('Invalid parameter "'.$param.'".', 20);
     }
}
```

Para facilitar las pruebas y reducir la duplicidad se puede usar un atajo para especificar una constante de clase como `@expectedExceptionMessage` usando la sintaxis «`@expectedExceptionMessage ClassName::CONST`». Podemos encontrar un ejemplo en *@expectedExceptionCode*.

14.18 @expectedExceptionMessageRegExp

El mensaje esperado puede ser especificado con una expresión regular usando la anotación `@expectedExceptionMessageRegExp`. Esta etiqueta es útil para situaciones donde una sub cadena de caracteres no es adecuada para comparar un mensaje dado.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
     /**
      * @expectedException             MyException
      * @expectedExceptionMessageRegExp /Argument \d+ can not be an? \w+/
```

```
     */
    public function testExceptionHasRightMessage()
    {
        throw new MyException('Argument 2 can not be an integer');
    }
}
```

14.19 @group

Una prueba puede ser etiquetada como perteneciendo a uno o más grupos usando la anotación `@group` de la siguiente manera:

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @group specification
     */
    public function testSomething()
    {
    }

    /**
     * @group regresssion
     * @group bug2204
     */
    public function testSomethingElse()
    {
    }
}
```

La anotación `@group` se puede colocar en la clase de prueba. En este caso todos los métodos de prueba «heredan» de la clase de prueba.

Las pruebas se pueden seleccionar para su ejecución en base a grupos usando las opciones `--group` y `--exclude-group` del ejecutor de pruebas en línea de comandos o usando las directivas respectivas en el archivo de configuración XML.

14.20 @large

La anotación `@large` es una alias para `@group large`.

Si el paquete `PHP_Invoker` está instalado y el modo estricto esta habilitado, las pruebas largas fallarán si ellas toman más de 60 segundo en ejecución. Este tiempo límite se configura con el atributo `timeoutForLargeTests` en el archivo de configuración XML.

14.21 @medium

La anotación `@medium` es una alias para `@group medium`. Una prueba media no debería depender de una prueba marcada como `@large`-

Si el paquete `PHP_Invoker` está instalado y el modo estricto está habilitado, una prueba mediana fallará si su ejecución tarda más de 10 segundos. Este tiempo de espera se configura a través del atributo `timeoutForMediumTests` en el archivo de configuración XML.

14.22 @preserveGlobalState

Cuando una prueba se ejecuta en un proceso separado PHPUnit intentará preservar el estado global a partir del proceso padre mediante la serialización de todas las globales que están en el proceso padre y luego descerializandolas en el proceso hijo. Esto puede causar problemas si el proceso padre contiene globales que no son serializables. Para corregir esto, podemos evitar que PHPUnit preserve el estado global con la anotación `@preserveGlobalState`.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @runInSeparateProcess
     * @preserveGlobalState disabled
     */
    public function testInSeparateProcess()
    {
        // ...
    }
}
```

14.23 @requires

La anotación `@requires` se puede usar para saltar pruebas cuando las precondiciones comunes, como la versión de PHP o las extensiones instaladas no se encuentran.

Una lista completa de posibilidades y ejemplos se puede encontrar en *Posibles usos para @requires*

14.24 @runTestsInSeparateProcesses

Indica que todas las pruebas en la clase de pruebas deben ser ejecutados en un proceso PHP separado.

```
use PHPUnit\Framework\TestCase;

/**
 * @runTestsInSeparateProcesses
 */
class MyTest extends TestCase
{
    // ...
}
```

Nota: Por defecto, PHPUnit intentará preservar el estado global desde el proceso padre serializando todas las globales del proceso padre y deserialzandolas en el proceso hijo. Esto puede causar problemas si el proceso padre contiene globales que no son serializables. Para información sobre como corregir esto podemos revisar *@preserveGlobalState*.

14.25 @runInSeparateProcess

Indica que la prueba debe ser ejecutada en un proceso PHP separado.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @runInSeparateProcess
     */
    public function testInSeparateProcess()
    {
        // ...
    }
}
```

Nota: Por defecto, PHPUnit intentará preservar el estado global desde el proceso padre serializando todas las globales del proceso padre y deserialzandolas en el proceso hijo. Esto puede causar problemas si el proceso padre contiene globales que no son serializables. Para información sobre como corregir esto podemos revisar *@preserveGlobalState*.

14.26 @small

La anotación `@small` es un alias para `@group small`. Una prueba pequeña no debe depender de una prueba marcada como `@medium` o `@large`.

Si el paquete `PHP_Invoker` está instalado y el modo estricto está habilitado, una prueba «pequeña» fallará si toma más de un segundo en ejecutarse. El tiempo de espera se configura con el atributo `timeoutForSmallTests` en el archivo de configuración XML.

Nota

Para habilitar los límites de tiempo las pruebas tienen que ser anotadas explícitamente con `@small`, `@medium` y `@large`.

14.27 @test

Como alternativa a añadir el prefijo `test` al nombre del método podemos usar la anotación `@test` en un bloque de documentación de método para marcarlo como un método de prueba.

```
/**
 * @test
 */
public function initialBalanceShouldBe0()
{
    $this->assertSame(0, $this->ba->getBalance());
}
```

14.28 @testdox

Especifica una descripción alternativa que se usa cuando se genera la documentación ágil de sentencias.

La anotación `@testdox` se pueden aplicar tanto a clases de prueba como a métodos de prueba.

```
/**
 * @testdox A bank account
 */
class BankAccountTest extends TestCase
{
    /**
     * @testdox has an initial balance of zero
     */
    public function balanceIsInitiallyZero()
    {
        $this->assertSame(0, $this->ba->getBalance());
    }
}
```

Nota

Antes de PHPUnit 7.0 (debido a un error en el análisis sintáctico de las anotaciones, «parsing»), cuando se usa la anotación `@testdox` además se activa la anotación `@test`.

14.29 @testWith

En lugar de implementar un método como `@dataProvider` podemos definir un conjunto de datos usando la anotación `@testWith`.

Un conjunto de datos consiste de uno o varios elementos. Para definir un conjunto de datos con multiples elementos definimos cada elemento en una línea separada. Cada elemento del conjunto de datos debe ser un arreglo definido en JSON.

Ver *Proveedores de Datos* para aprender más sobre como pasar un conjunto de datos a una prueba.

```
/**
 * @param string    $input
 * @param int       $expectedLength
 *
 * @testWith        ["test", 4]
 *                  ["longer-string", 13]
 */
public function testStringLength(string $input, int $expectedLength)
{
    $this->assertSame($expectedLength, strlen($input));
}
```

Una representación de un objeto en JSON será convertido a un arreglo asociado.

```
/**
 * @param array     $array
 * @param array     $keys
 *
```

```
     * @testWith        [{"day": "monday", "conditions": "sunny"}, ["day", "conditions"]]
     */
    public function testArrayKeys($array, $keys)
    {
        $this->assertSame($keys, array_keys($array));
    }
```

14.30 @ticket

The `@ticket` annotation is an alias for the `@group` annotation (see *@group*) and allows to filter tests based on their ticket ID.

14.31 @uses

La anotación `@uses` especifica el código que será ejecutado por una prueba pero que no deseamos que sea cubierto por la prueba. Un buen ejemplo es un objeto de valor, «value object», que se necesita para probar una unidad de código.

```
    /**
     * @covers BankAccount::deposit
     * @uses   Money
     */
    public function testMoneyCanBeDepositedInAccount()
    {
        // ...
    }
```

Esta anotación es especialmente útil en el modo de cobertura estricto en donde código cubierto causa involuntariamente que la prueba falle. Para más información sobre el modo de cobertura estricto podemos ver *Cobertura Involuntaria de Código*.

CAPÍTULO 15

Archivo de Configuración XML

15.1 PHPUnit

Los atributos del elemento <phpunit> se pueden usar para configurar la funcionalidad del núcleo de PHPUnit.

```
<phpunit
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="https://schema.phpunit.de/|version|/phpunit.
↪xsd"
         backupGlobals="true"
         backupStaticAttributes="false"
         <!--bootstrap="/path/to/bootstrap.php"-->
         cacheTokens="false"
         colors="false"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         forceCoversAnnotation="false"
         mapTestClassNameToCoveredClassName="false"
         printerClass="PHPUnit\TextUI\ResultPrinter"
         <!--printerFile="/path/to/ResultPrinter.php"-->
         processIsolation="false"
         stopOnError="false"
         stopOnFailure="false"
         stopOnIncomplete="false"
         stopOnSkipped="false"
         stopOnRisky="false"
         testSuiteLoaderClass="PHPUnit\Runner\StandardTestSuiteLoader"
         <!--testSuiteLoaderFile="/path/to/StandardTestSuiteLoader.php"-->
         timeoutForSmallTests="1"
         timeoutForMediumTests="10"
         timeoutForLargeTests="60"
         verbose="false">
  <!-- ... -->
</phpunit>
```

El XML de configuración de arriba corresponde al comportamiento por defecto del ejecutor de pruebas TextUI documentado en *Opciones de la línea de comandos*.

Las opciones adicionales que no están disponibles como opciones desde la línea de comandos son:

`convertErrorsToExceptions`

Por defecto PHPUnit instalará un gestor de errores que convierte los siguientes errores en excepciones:

- `E_WARNING`
- `E_NOTICE`
- `E_USER_ERROR`
- `E_USER_WARNING`
- `E_USER_NOTICE`
- `E_STRICT`
- `E_RECOVERABLE_ERROR`
- `E_DEPRECATED`
- `E_USER_DEPRECATED`

Colocamos `convertErrorsToExceptions` en `false` para desactivar esta característica.

`convertNoticesToExceptions`

Cuando colocamos `false`, el gestor de errores instalado por `convertErrorsToExceptions` no convertirá los errores `E_NOTICE`, `E_USER_NOTICE` o `E_STRICT` en excepciones.

`convertWarningsToExceptions`

Cuando colocamos `false`, el gestor de errores instalado por `convertErrorsToExceptions` no convertirá los errores `E_WARNING` o `E_USER_WARNING` en excepciones.

`forceCoversAnnotation`

El código de cobertura solo será reportado para pruebas que usan la anotación `@covers` documentada en *@covers*.

`timeoutForLargeTests`

Si los límites de tiempo basados en el tamaño de la prueba son forzados entonces este atributo designará el tiempo límite para todos las pruebas marcadas con `@large`. Si una prueba no se completa dentro del tiempo límite configurado entonces la prueba fallará.

`timeoutForMediumTests`

Si los límites de tiempo basados en el tamaño de la prueba son forzados entonces este atributo designará el tiempo límite para todas las pruebas marcadas con `@medium`. Si una prueba no se completa dentro del tiempo límite configurado entonces la prueba fallará.

`timeoutForSmallTests`

Si los límites de tiempo basados en el tamaño de la prueba son forzados entonces este atributo designa el tiempo límite para todas las pruebas que no han sido marcadas como `@medium` o `@large`. Si una prueba no se completa dentro del tiempo límite configurado, la prueba fallará.

15.2 Suites de Pruebas

El elemento `<testsuites>` y su o sus hijos `<testsuite>` se pueden usar para construir una suite de prueba compuesta por suites de pruebas y casos de prueba.

```
<testsuites>
  <testsuite name="My Test Suite">
    <directory>/path/to/*Test.php files</directory>
    <file>/path/to/MyTest.php</file>
    <exclude>/path/to/exclude</exclude>
  </testsuite>
</testsuites>
```

Si usamos los atributos `phpVersion` y `phpVersionOperator` se puede especificar una determinada versión de PHP. El ejemplo de abajo solo agregará los archivos `/path/to/*Test.php` y el archivo `/path/to/MyTest.php` si la versión de PHP es superior o igual a 5.3.0.

```
<testsuites>
  <testsuite name="My Test Suite">
    <directory suffix="Test.php" phpVersion="5.3.0" phpVersionOperator=">=">/path/to/
↪files</directory>
    <file phpVersion="5.3.0" phpVersionOperator=">=">/path/to/MyTest.php</file>
  </testsuite>
</testsuites>
```

El atributo `phpVersionOperator` es opcional y por defecto tiene el valor >=.

15.3 Grupos

El elemento `<groups>` y sus hijos `<include>`, `<exclude>` y `<group>` se pueden usar para seleccionar grupos de pruebas marcadas con la anotación `@group` (documentada en *@group*) que serán o no serán ejecutadas.

```
<groups>
  <include>
    <group>name</group>
  </include>
  <exclude>
    <group>name</group>
  </exclude>
</groups>
```

La configuración XML de arriba es equivalente a invocar el ejecutor de pruebas TextUI con las siguientes opciones:

- `--group name`
- `--exclude-group name`

15.4 Lista Blanca de Archivos para la Cobertura de Código

El elemento `<filter>` y sus hijos se pueden usar para configurar la lista blanca usada para el reporte de cobertura de código.

```
<filter>
  <whitelist processUncoveredFilesFromWhitelist="true">
    <directory suffix=".php">/path/to/files</directory>
    <file>/path/to/file</file>
    <exclude>
      <directory suffix=".php">/path/to/files</directory>
      <file>/path/to/file</file>
    </exclude>
  </whitelist>
</filter>
```

15.5 Logging

El elemento `<logging>` y su hijo `<log>` se puede usar para configurar el registro de sucesos de la ejecución de la prueba.

```
<logging>
  <log type="coverage-html" target="/tmp/report" lowUpperBound="35"
       highLowerBound="70"/>
  <log type="coverage-clover" target="/tmp/coverage.xml"/>
  <log type="coverage-php" target="/tmp/coverage.serialized"/>
  <log type="coverage-text" target="php://stdout" showUncoveredFiles="false"/>
  <log type="junit" target="/tmp/logfile.xml"/>
  <log type="testdox-html" target="/tmp/testdox.html"/>
  <log type="testdox-text" target="/tmp/testdox.txt"/>
</logging>
```

La configuración XML de arriba es equivalente a invocar el ejecutor de pruebas TextUI con las siguiente opciones:

- `--coverage-html /tmp/report`
- `--coverage-clover /tmp/coverage.xml`
- `--coverage-php /tmp/coverage.serialized`
- `--coverage-text`
- `> /tmp/logfile.txt`
- `--log-junit /tmp/logfile.xml`
- `--testdox-html /tmp/testdox.html`
- `--testdox-text /tmp/testdox.txt`

Los atributos `lowUpperBound`, `highLowerBound` y `showUncoveredFiles` no tienen opciones equivalentes en el ejecutor de pruebas TextUI.

- `lowUpperBound`: Máximo porcentaje de cobertura de código para considerar la cobertura como baja, «lowly».
- `highLowerBound`: Mínimo porcentaje de cobertura para que se considere como alta, «highly».
- `showUncoveredFiles`: Mostrar todos los archivos de la lista blanca en la salida `--coverage-text` y no solo los archivos con la información de cobertura.
- `showOnlySummary`: Solo mostrar el resumen en la salida `--coverage-text`.

15.6 «Listeners» de Prueba

El elemento `<listeners>` y su hijo `<listener>` se pueden usar para adjuntar escuchas de prueba, «test listeners», adicionales a la prueba en ejecución.

```
<listeners>
  <listener class="MyListener" file="/optional/path/to/MyListener.php">
    <arguments>
      <array>
        <element key="0">
          <string>Sebastian</string>
        </element>
      </array>
      <integer>22</integer>
      <string>April</string>
      <double>19.78</double>
      <null/>
      <object class="stdClass"/>
    </arguments>
  </listener>
</listeners>
```

La configuración XML de arriba corresponde a adjuntar el objeto `$listener` (ver abajo) a la ejecución de una prueba:

```
$listener = new MyListener(
    ['Sebastian'],
    22,
    'April',
    19.78,
    null,
    new stdClass
);
```

15.7 Registrar Extensiones de TestRunner

El elemento `<extensions>` y su hijo `<extension>` se puede usar para registrar extensiones de TextRunner personalizadas.

El Example 15.1 muestra como registrar una extensión.

Example 15.1: Registrar una Extensión de TestRunner

```
  <?xml version="1.0" encoding="UTF-8"?>
  <phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"␣
↪xsi:noNamespaceSchemaLocation="https://schema.phpunit.de/7.1/phpunit.xsd">
      <extensions>
          <extension class="Vendor\MyExtension"/>
      </extensions>
  </phpunit>
```

15.8 Asignar las configuraciones de PHP INI, Constantes y Variables Globales

El elemento <php> y sus hijos se pueden usar para establecer las configuraciones de PHP, constantes y variables globales. Se puede usar para agregar el include_path.

```
<php>
  <includePath>.</includePath>
  <ini name="foo" value="bar"/>
  <const name="foo" value="bar"/>
  <var name="foo" value="bar"/>
  <env name="foo" value="bar"/>
  <post name="foo" value="bar"/>
  <get name="foo" value="bar"/>
  <cookie name="foo" value="bar"/>
  <server name="foo" value="bar"/>
  <files name="foo" value="bar"/>
  <request name="foo" value="bar"/>
</php>
```

La configuración XML de arriba se corresponde con el siguiente código PHP:

```
ini_set('foo', 'bar');
define('foo', 'bar');
$GLOBALS['foo'] = 'bar';
$_ENV['foo'] = 'bar';
$_POST['foo'] = 'bar';
$_GET['foo'] = 'bar';
$_COOKIE['foo'] = 'bar';
$_SERVER['foo'] = 'bar';
$_FILES['foo'] = 'bar';
$_REQUEST['foo'] = 'bar';
```

CAPÍTULO 16

Bibliography

[Astels2003] David Astels. *Test Driven Development*.

[Beck2002] Kent Beck. *Test Driven Development by Example*.

[Meszaros2007] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*.

CAPÍTULO 17

Copyright

```
Copyright (c) 2005-2018 Sebastian Bergmann.

This work is licensed under the Creative Commons Attribution 3.0
Unported License.

A summary of the license is given below, followed by the full legal
text.

---------------------------------------------------------------------

You are free:

    * to Share - to copy, distribute and transmit the work
    * to Remix - to adapt the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by
the author or licensor (but not in any way that suggests that they
endorse you or your use of the work).

    * For any reuse or distribution, you must make clear to others
      the license terms of this work. The best way to do this is with
      a link to this web page.

    * Any of the above conditions can be waived if you get
      permission from the copyright holder.

    * Nothing in this license impairs or restricts the author's moral
      rights.

Your fair dealing and other rights are in no way affected by the
above.

This is a human-readable summary of the Legal Code (the full
```

```
license) below.

=============================================================================

Creative Commons Legal Code
Attribution 3.0 Unported

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS
INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO
WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS
LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS
CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS
PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE
WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW
IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND
AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS
LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU
THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF
SUCH TERMS AND CONDITIONS.

1. Definitions

   a. "Adaptation" means a work based upon the Work, or upon the
      Work and other pre-existing works, such as a translation,
      adaptation, derivative work, arrangement of music or other
      alterations of a literary or artistic work, or phonogram or
      performance and includes cinematographic adaptations or any
      other form in which the Work may be recast, transformed, or
      adapted including in any form recognizably derived from the
      original, except that a work that constitutes a Collection
      will not be considered an Adaptation for the purpose of this
      License. For the avoidance of doubt, where the Work is a
      musical work, performance or phonogram, the synchronization of
      the Work in timed-relation with a moving image ("synching")
      will be considered an Adaptation for the purpose of this
      License.

   b. "Collection" means a collection of literary or artistic works,
      such as encyclopedias and anthologies, or performances,
      phonograms or broadcasts, or other works or subject matter
      other than works listed in Section 1(f) below, which, by
      reason of the selection and arrangement of their contents,
      constitute intellectual creations, in which the Work is
      included in its entirety in unmodified form along with one or
      more other contributions, each constituting separate and
      independent works in themselves, which together are assembled
      into a collective whole. A work that constitutes a Collection
      will not be considered an Adaptation (as defined above) for
      the purposes of this License.
```

```
c. "Distribute" means to make available to the public the
   original and copies of the Work or Adaptation, as appropriate,
   through sale or other transfer of ownership.

d. "Licensor" means the individual, individuals, entity or
   entities that offer(s) the Work under the terms of this License.

e. "Original Author" means, in the case of a literary or artistic
   work, the individual, individuals, entity or entities who
   created the Work or if no individual or entity can be
   identified, the publisher; and in addition (i) in the case of
   a performance the actors, singers, musicians, dancers, and
   other persons who act, sing, deliver, declaim, play in,
   interpret or otherwise perform literary or artistic works or
   expressions of folklore; (ii) in the case of a phonogram the
   producer being the person or legal entity who first fixes the
   sounds of a performance or other sounds; and, (iii) in the
   case of broadcasts, the organization that transmits the
   broadcast.

f. "Work" means the literary and/or artistic work offered under
   the terms of this License including without limitation any
   production in the literary, scientific and artistic domain,
   whatever may be the mode or form of its expression including
   digital form, such as a book, pamphlet and other writing; a
   lecture, address, sermon or other work of the same nature; a
   dramatic or dramatico-musical work; a choreographic work or
   entertainment in dumb show; a musical composition with or
   without words; a cinematographic work to which are assimilated
   works expressed by a process analogous to cinematography; a
   work of drawing, painting, architecture, sculpture, engraving
   or lithography; a photographic work to which are assimilated
   works expressed by a process analogous to photography; a work
   of applied art; an illustration, map, plan, sketch or three-
   dimensional work relative to geography, topography,
   architecture or science; a performance; a broadcast; a
   phonogram; a compilation of data to the extent it is protected
   as a copyrightable work; or a work performed by a variety or
   circus performer to the extent it is not otherwise considered
   a literary or artistic work.

g. "You" means an individual or entity exercising rights under
   this License who has not previously violated the terms of
   this License with respect to the Work, or who has received
   express permission from the Licensor to exercise rights under
   this License despite a previous violation.

h. "Publicly Perform" means to perform public recitations of the
   Work and to communicate to the public those public
   recitations, by any means or process, including by wire or
   wireless means or public digital performances; to make
   available to the public Works in such a way that members of
   the public may access these Works from a place and at a place
   individually chosen by them; to perform the Work to the public
   by any means or process and the communication to the public of
   the performances of the Work, including by public digital
   performance; to broadcast and rebroadcast the Work by any
   means including signs, sounds or images.
```

```
   i. "Reproduce" means to make copies of the Work by any means
      including without limitation by sound or visual recordings and
      the right of fixation and reproducing fixations of the Work,
      including storage of a protected performance or phonogram in
      digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to
   reduce, limit, or restrict any uses free from copyright or rights
   arising from limitations or exceptions that are provided for in
   connection with the copyright protection under copyright law or
   other applicable laws.

3. License Grant. Subject to the terms and conditions of this
   License, Licensor hereby grants You a worldwide, royalty-free,
   non-exclusive, perpetual (for the duration of the applicable
   copyright) license to exercise the rights in the Work as stated
   below:

   a. to Reproduce the Work, to incorporate the Work into one or
      more Collections, and to Reproduce the Work as incorporated
      in the Collections;

   b. to create and Reproduce Adaptations provided that any such
      Adaptation, including any translation in any medium, takes
      reasonable steps to clearly label, demarcate or otherwise
      identify that changes were made to the original Work. For
      example, a translation could be marked "The original work was
      translated from English to Spanish," or a modification could
      indicate "The original work has been modified.";

   c. to Distribute and Publicly Perform the Work including as
      incorporated in Collections; and,

   d. to Distribute and Publicly Perform Adaptations.

   e. For the avoidance of doubt:

      i. Non-waivable Compulsory License Schemes. In those
         jurisdictions in which the right to collect royalties
         through any statutory or compulsory licensing scheme cannot
         be waived, the Licensor reserves the exclusive right to
         collect such royalties for any exercise by You of the
         rights granted under this License;

      ii. Waivable Compulsory License Schemes. In those
          jurisdictions in which the right to collect royalties
          through any statutory or compulsory licensing scheme can
          be waived, the Licensor waives the exclusive right to
          collect such royalties for any exercise by You of the
          rights granted under this License; and,

      iii. Voluntary License Schemes. The Licensor waives the right
           to collect royalties, whether individually or, in the
           event that the Licensor is a member of a collecting
           society that administers voluntary licensing schemes, via
           that society, from any exercise by You of the rights
           granted under this License.
```

```
The above rights may be exercised in all media and formats whether
now known or hereafter devised. The above rights include the right
to make such modifications as are technically necessary to exercise
the rights in other media and formats. Subject to Section 8(f), all
rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly
   made subject to and limited by the following restrictions:

   a. You may Distribute or Publicly Perform the Work only under the
      terms of this License. You must include a copy of, or the
      Uniform Resource Identifier (URI) for, this License with every
      copy of the Work You Distribute or Publicly Perform. You may
      not offer or impose any terms on the Work that restrict the
      terms of this License or the ability of the recipient of the
      Work to exercise the rights granted to that recipient under
      the terms of the License. You may not sublicense the Work. You
      must keep intact all notices that refer to this License and to
      the disclaimer of warranties with every copy of the Work You
      Distribute or Publicly Perform. When You Distribute or
      Publicly Perform the Work, You may not impose any effective
      technological measures on the Work that restrict the ability
      of a recipient of the Work from You to exercise the rights
      granted to that recipient under the terms of the License. This
      Section 4(a) applies to the Work as incorporated in a
      Collection, but this does not require the Collection apart
      from the Work itself to be made subject to the terms of this
      License. If You create a Collection, upon notice from any
      Licensor You must, to the extent practicable, remove from the
      Collection any credit as required by Section 4(b), as
      requested. If You create an Adaptation, upon notice from any
      Licensor You must, to the extent practicable, remove from the
      Adaptation any credit as required by Section 4(b), as requested.

   b. If You Distribute, or Publicly Perform the Work or any
      Adaptations or Collections, You must, unless a request has
      been made pursuant to Section 4(a), keep intact all copyright
      notices for the Work and provide, reasonable to the medium or
      means You are utilizing: (i) the name of the Original Author
      (or pseudonym, if applicable) if supplied, and/or if the
      Original Author and/or Licensor designate another party or
      parties (e.g., a sponsor institute, publishing entity,
      journal) for attribution ("Attribution Parties") in Licensor's
      copyright notice, terms of service or by other reasonable
      means, the name of such party or parties; (ii) the title of
      the Work if supplied; (iii) to the extent reasonably
      practicable, the URI, if any, that Licensor specifies to be
      associated with the Work, unless such URI does not refer to
      the copyright notice or licensing information for the Work;
      and (iv), consistent with Section 3(b), in the case of an
      Adaptation, a credit identifying the use of the Work in the
      Adaptation (e.g., "French translation of the Work by Original
      Author," or "Screenplay based on original Work by Original
      Author"). The credit required by this Section 4 (b) may be
      implemented in any reasonable manner; provided, however, that
      in the case of a Adaptation or Collection, at a minimum such
      credit will appear, if a credit for all contributing authors
```

```
      of the Adaptation or Collection appears, then as part of these
      credits and in a manner at least as prominent as the credits
      for the other contributing authors. For the avoidance of
      doubt, You may only use the credit required by this Section
      for the purpose of attribution in the manner set out above
      and, by exercising Your rights under this License, You may not
      implicitly or explicitly assert or imply any connection with,
      sponsorship or endorsement by the Original Author, Licensor
      and/or Attribution Parties, as appropriate, of You or Your use
      of the Work, without the separate, express prior written
      permission of the Original Author, Licensor and/or
      Attribution Parties.

   c. Except as otherwise agreed in writing by the Licensor or as
      may be otherwise permitted by applicable law, if You
      Reproduce, Distribute or Publicly Perform the Work either by
      itself or as part of any Adaptations or Collections, You must
      not distort, mutilate, modify or take other derogatory action
      in relation to the Work which would be prejudicial to the
      Original Author's honor or reputation. Licensor agrees that in
      those jurisdictions (e.g. Japan), in which any exercise of the
      right granted in Section 3(b) of this License (the right to
      make Adaptations) would be deemed to be a distortion,
      mutilation, modification or other derogatory action
      prejudicial to the Original Author's honor and reputation, the
      Licensor will waive or not assert, as appropriate, this
      Section, to the fullest extent permitted by the applicable
      national law, to enable You to reasonably exercise Your right
      under Section 3(b) of this License (right to make Adaptations)
      but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING,
LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR
WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED,
STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF
TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE,
NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS,
ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT
DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF
IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY
   APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY
   LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE
   OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF
   THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate
      automatically upon any breach by You of the terms of this
      License. Individuals or entities who have received Adaptations
      or Collections from You under this License, however, will not
      have their licenses terminated provided such individuals or
      entities remain in full compliance with those licenses.
```

```
      Sections 1, 2, 5, 6, 7, and 8 will survive any termination of
      this License.

   b. Subject to the above terms and conditions, the license granted
      here is perpetual (for the duration of the applicable
      copyright in the Work). Notwithstanding the above, Licensor
      reserves the right to release the Work under different license
      terms or to stop distributing the Work at any time; provided,
      however that any such election will not serve to withdraw this
      License (or any other license that has been, or is required to
      be, granted under the terms of this License), and this License
      will continue in full force and effect unless terminated as
      stated above.

8. Miscellaneous

   a. Each time You Distribute or Publicly Perform the Work or a
      Collection, the Licensor offers to the recipient a license to
      the Work on the same terms and conditions as the license
      granted to You under this License.

   b. Each time You Distribute or Publicly Perform an Adaptation,
      Licensor offers to the recipient a license to the original
      Work on the same terms and conditions as the license granted
      to You under this License.

   c. If any provision of this License is invalid or unenforceable
      under applicable law, it shall not affect the validity or
      enforceability of the remainder of the terms of this License,
      and without further action by the parties to this agreement,
      such provision shall be reformed to the minimum extent
      necessary to make such provision valid and enforceable.

   d. No term or provision of this License shall be deemed waived
      and no breach consented to unless such waiver or consent shall
      be in writing and signed by the party to be charged with such
      waiver or consent.

   e. This License constitutes the entire agreement between the
      parties with respect to the Work licensed here. There are no
      understandings, agreements or representations with respect to
      the Work not specified here. Licensor shall not be bound by
      any additional provisions that may appear in any communication
      from You. This License may not be modified without the mutual
      written agreement of the Licensor and You.

   f. The rights granted under, and the subject matter referenced,
      in this License were drafted utilizing the terminology of the
      Berne Convention for the Protection of Literary and Artistic
      Works (as amended on September 28, 1979), the Rome Convention
      of 1961, the WIPO Copyright Treaty of 1996, the WIPO
      Performances and Phonograms Treaty of 1996 and the Universal
      Copyright Convention (as revised on July 24, 1971). These
      rights and subject matter take effect in the relevant
      jurisdiction in which the License terms are sought to be
      enforced according to the corresponding provisions of the
      implementation of those treaty provisions in the applicable
      national law. If the standard suite of rights granted under
```

```
      applicable copyright law includes additional rights not
      granted under this License, such additional rights are deemed
      to be included in the License; this License is not intended to
      restrict the license of any rights under applicable law.

Creative Commons is not a party to this License, and makes no
warranty whatsoever in connection with the Work. Creative Commons
will not be liable to You or any party on any legal theory for any
damages whatsoever, including without limitation any general,
special, incidental or consequential damages arising in connection
to this license. Notwithstanding the foregoing two (2) sentences,
if Creative Commons has expressly identified itself as the Licensor
hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the
Work is licensed under the CCPL, Creative Commons does not authorize
the use by either party of the trademark "Creative Commons" or any
related trademark or logo of Creative Commons without the prior
written consent of Creative Commons. Any permitted use will be in
compliance with Creative Commons' then-current trademark usage
guidelines, as may be published on its website or otherwise made
available upon request from time to time. For the avoidance of
doubt, this trademark restriction does not form part of this
License.

Creative Commons may be contacted at http://creativecommons.org/.

======================================================================
```