**Тестовые макеты PHPUnit**

Жерар Месзарос (Gerard Meszaros) представил новыую идею тестовых макетов (имитаций) в [[Meszaros2007](http://yapro.ru/web-master/php/appendixes.bibliography.html#Meszaros2007)] следующим образом:

 Иногда очень сложно протестировать испытываемую систему, потому что она зависит от других компонентов, которые нельзя использовать в тестовом окружении. Это происходит из-за того, что они недоступны, они не вернут нужный результат для теста, или вызов этих компонент повлечёт за собой нежелательные побочные эффекты. В других случаях наша стратегия тестирования требует большего контроля и видения того что происходит внутри системы.

  Когда мы пишем тест в котором не можем (или не хотим) использовать реальные зависимости, мы можем заменить их тестовыми макетами. Тестовые макеты не должны вести себя в точности как реальные компоненты; они должны предоставлять тот же API, который предоставляет и реальный компонент, так, чтобы тестируемая система думала что они реальны!

Метод **getMock($className)**, предоставляемый в PHPUnit может быть использован в тесте для генерации объекта, который ведёт себя как тестовый макеты для указанного класса. Этот объект-имитация может быть использован в любом месте кода, где ожидается объект оригинального класса.

По умолчанию все методы оригинального класса заменяются методами-пустышками, которые возвращают **NULL** (не вызывая оригинальных методов, кроме конструктора). Используя метод **will($this->returnValue())**, например, можно настроить эти методы-пустышки таким образом, чтоб они возвращали заданное значение.

**Ограничения**

Обратите внимание, что для **final**, **private** и **static** методов, нельзя сделать заглушку (stub) и мок (mock). Они игнорируются функциональностью PHPUnit по созданию тестовых макетов и сохраняют изначальное поведение.

**Предупреждение**

Пожалуйста, обратите внимание на тот факт, что управление параметрами было изменено. Предыдущая версия клонировала все параметры объекта. Это не позволяло проверить был ли передан методу тот же самый объект или нет. [Пример 10.14, «Проверка того что метод был вызван единожды с объектом идентичным заданному»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.mock-objects.examples.clone-object-parameters-usecase.php) показывает где новый подход может быть полезен. [Пример 10.15, «Создание мок-объекта с клонированием параметров»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.mock-objects.examples.enable-clone-object-parameters.php) показывает как можно вернуться к поведению котрое было раньше.

 Заглушки (Stubs)

Практика замены объекта тестовым макетом, который (опционально) возвращает заданные значения называется *заглушением (stubbing)*. *Заглушку (stub)* можно использовать для того чтобы: "заменить исходный компонент от которого зависит тестируемая система таким образом, что тест будет иметь возможность контролировать тестируемую систему изнутри. Это позволяет заставить тестируемую систему выполнить действия, которые она не выполнила бы в обычных условиях."

[Пример 10.2, «Заглушка метода для возвращения фиксированного значения»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.stubs.examples.StubTest.php) показывает как сделать заглушку метода и задать возвращаемое значение. Первым использован метод **getMock()**, предоставляемый классом **PHPUnit\_Framework\_TestCase**, для создания заглушки класса **SomeClass** ([Пример 10.1, «Класс, заглушку которого мы хотим создать»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.stubs.examples.SomeClass.php)), который ведет себя так же как и оригинальный объект. Затем используется [текучий интерфейс](http://yapro.ru/system/redirect?to=http://martinfowler.com/bliki/FluentInterface.html), предоставляемый PHPUnit для указания поведения заглушки. В конечном счёте не нужно создавать несколько временных объектов и соединять их после. Это приводит к более читабельному "текучему" коду.

**Пример 10.1. Класс, заглушку которого мы хотим создать**

<?php

class SomeClass

{

public function doSomething()

{

// Do something.

}

}

?>

**Пример 10.2. Заглушка метода для возвращения фиксированного значения**

<?php

require\_once 'SomeClass.php';

class StubTest extends PHPUnit\_Framework\_TestCase

{

public function testStub()

{

// Создание заглушки класса SomeClass.

$stub = $this->getMock('SomeClass');

// Настройка заглушки.

$stub->expects($this->any())

->method('doSomething')

->will($this->returnValue('foo'));

// Вызов $stub->doSomething() теперь вернет

// 'foo'.

$this->assertEquals('foo', $stub->doSomething());

}

}

?>

"За кулисами" PHPUnit автоматически создаёт новый PHP класс, который, после использования метода **getMock()**, повторяет заданное поведение. Созданный тестовый макет можно настроить с помощью необязательных аргументов метода **getMock()**.

По умолчанию все методы заданного класса заменяются двойниками, которые возвращают значение **NULL**, если не задано другое значение, например, с помощью метода **will($this->returnValue())**.

Когда задан второй (необязательный) параметр, заменяются только те методы тестового макета, которые присутствуют в массиве. Поведение других методов не изменяется.

Третий (необязательный) параметр может содержать массив аргументов, передаваемых в конструктор оригинального класса (который не заменяется методом-пустышкой по умолчанию).

Четвертый (необязательный) параметр указывает имя класса для создаваемого тестового макета.

Пятый (необязательный) параметр указывает на то, использовать конструктор исходного класса или нет.

Шестой (необязательный) параметр используется для указания использования метода клонирования исходного класса или нет.

Седьмой (необязательный) параметр может запретить использование **\_\_autoload()** во время создания тестового макета.

Для настройки создаваемого макета также можно использовать Mock Builder API . [Пример 10.3, «Использование Mock Builder API для настройки создаваемого класса макета»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.stubs.examples.StubTest2.php) показывает пример использования. Ниже приведён список методов, которые могут быть использованы:

**setMethods(array $methods)** вызывается объектом Mock Builder для указания того, какие методы будут заменены в макете. Поведение других методов не затрагивается.

**setConstructorArgs(array $args)** вызывается для указания массива параметров, передаваемых в конструктор исходного класса (который по умолчанию не заменяется методом-пустышкой).

**setMockClassName($name)** вызывается для указания имени класса создаваемого класса макета.

**disableOriginalConstructor()** вызывается для отмены использования конструктора исходного класса.

**disableOriginalClone()** вызывается для отмены вызова оригинального метода клонирования.

**disableAutoload()** вызывается для отмены использования **\_\_autoload()** во время создания макета класса.

**Пример 10.3. Использование Mock Builder API для настройки создаваемого класса макета**

<?php

require\_once 'SomeClass.php';

class StubTest extends PHPUnit\_Framework\_TestCase

{

public function testStub()

{

// Создание заглушки класса SomeClass.

$stub = $this->getMockBuilder('SomeClass')

->disableOriginalConstructor()

->getMock();

// Настройка заглушки.

$stub->expects($this->any())

->method('doSomething')

->will($this->returnValue('foo'));

// Вызов $stub->doSomething() теперь вернет

// 'foo'.

$this->assertEquals('foo', $stub->doSomething());

}

}

?>

Иногда необходимо вернуть (неизмененный) аргумент метода в заглушке. [Пример 10.4, «Заглушка метода для возврата одного из аргументов»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.stubs.examples.StubTest3.php) показывает как достичь этого, используя **returnArgument()** вместо **returnValue()**.

**Пример 10.4. Заглушка метода для возврата одного из аргументов**

<?php

require\_once 'SomeClass.php';

class StubTest extends PHPUnit\_Framework\_TestCase

{

public function testReturnArgumentStub()

{

// Создание заглушки класса SomeClass.

$stub = $this->getMock('SomeClass');

// Настройка заглушки.

$stub->expects($this->any())

->method('doSomething')

->will($this->returnArgument(0));

// $stub->doSomething('foo') вернёт 'foo'

$this->assertEquals('foo', $stub->doSomething('foo'));

// $stub->doSomething('bar') вернёт 'bar'

$this->assertEquals('bar', $stub->doSomething('bar'));

}

}

?>

Когда тестируется текучий интерфейс, будет полезным если заглушка вернёт ссылку на себя. [Пример 10.5, «Заглушка метода для возврата ссылки на себя»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.stubs.examples.StubTest4.php) показывает как использовать **returnSelf()** для того чтобы достичь этого.

**Пример 10.5. Заглушка метода для возврата ссылки на себя**

<?php

require\_once 'SomeClass.php';

class StubTest extends PHPUnit\_Framework\_TestCase

{

public function testReturnSelf()

{

// Создание заглушки класса SomeClass.

$stub = $this->getMock('SomeClass');

// Настройка заглушки.

$stub->expects($this->any())

->method('doSomething')

->will($this->returnSelf());

// $stub->doSomething() возвращает $stub

$this->assertSame($stub, $stub->doSomething());

}

}

?>

Иногда нужно чтоб заглушка метода возвращала разные значения в зависимости от предустановленного списка параметров. Для этого можно использовать метод **returnValueMap()** для создания списка, который ассоциирует аргументы с соответствующими значениями. Для примера см. [Пример 10.6, «Заглушка метода для возврата значений из списка в соответствии с аргументами»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.stubs.examples.StubTest5.php)

**Пример 10.6. Заглушка метода для возврата значений из списка в соответствии с аргументами**

<?php

require\_once 'SomeClass.php';

class StubTest extends PHPUnit\_Framework\_TestCase

{

public function testReturnValueMapStub()

{

// Создание заглушки класса SomeClass.

$stub = $this->getMock('SomeClass');

// Создание списка агрументов и соответствующих значений.

$map = array(

array('a', 'b', 'c', 'd'),

array('e', 'f', 'g', 'h')

);

// Настройка заглушки.

$stub->expects($this->any())

->method('doSomething')

->will($this->returnValueMap($map));

// $stub->doSomething() возвращает значения в соответствии с

// переданными аргументами.

$this->assertEquals('d', $stub->doSomething('a', 'b', 'c'));

$this->assertEquals('h', $stub->doSomething('e', 'f', 'g'));

}

}

?>

Если заглушка должна возвращать вычисляемое значение вместо фиксированного (см. **returnValue()**) или (неизмененного) аргумента (см. **returnArgument()**), можно использовать метод **returnCallback()** для того чтобы посчитать возвращаемое значение в коллбеке. Для примера см. [Пример 10.7, «Заглушка метода для возврата значения из коллбэк функции»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.stubs.examples.StubTest6.php).

**Пример 10.7. Заглушка метода для возврата значения из коллбэк функции**

<?php

require\_once 'SomeClass.php';

class StubTest extends PHPUnit\_Framework\_TestCase

{

public function testReturnCallbackStub()

{

// Создание заглушки класса SomeClass.

$stub = $this->getMock('SomeClass');

// Настройка заглушки.

$stub->expects($this->any())

->method('doSomething')

->will($this->returnCallback('str\_rot13'));

// $stub->doSomething($argument) возвращает результат str\_rot13($argument)

$this->assertEquals('fbzrguvat', $stub->doSomething('something'));

}

}

?>

Иногда проще задать список значений, которые возвращает заглушка при повторных вызовах. Для этого можно использовать метод **onConsecutiveCalls()**. Для примера см. [Пример 10.8, «Заглушка метода для возврата заданных значений в определенном порядке»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.stubs.examples.StubTest7.php).

**Пример 10.8. Заглушка метода для возврата заданных значений в определенном порядке**

<?php

require\_once 'SomeClass.php';

class StubTest extends PHPUnit\_Framework\_TestCase

{

public function testOnConsecutiveCallsStub()

{

// Создание заглушки класса SomeClass.

$stub = $this->getMock('SomeClass');

// Настройка заглушки.

$stub->expects($this->any())

->method('doSomething')

->will($this->onConsecutiveCalls(2, 3, 5, 7));

// $stub->doSomething() возвращает пределенное значение при каждом

// следующем вызове

$this->assertEquals(2, $stub->doSomething());

$this->assertEquals(3, $stub->doSomething());

$this->assertEquals(5, $stub->doSomething());

}

}

?>

Вместо возврата значения метод может выкинуть исключение. [Пример 10.9, «Заглушка, которая выкидывает исключение»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.stubs.examples.StubTest8.php) показывает пример использования метода **throwException()** для этих целей.

**Пример 10.9. Заглушка, которая выкидывает исключение**

<?php

require\_once 'SomeClass.php';

class StubTest extends PHPUnit\_Framework\_TestCase

{

public function testThrowExceptionStub()

{

// Создание заглушки класса SomeClass.

$stub = $this->getMock('SomeClass');

// Настройка заглушки.

$stub->expects($this->any())

->method('doSomething')

->will($this->throwException(new Exception));

// $stub->doSomething() выкидывает Exception

$stub->doSomething();

}

}

?>

Можно написать заглушку без использования **getMock($className)** и улучшить дизайн таким образом. Обычно широко используемые ресурсы доступны из одного и того же места, поэтому будет довольно просто заменить ресурс его заглушкой. Например, вместо того чтобы вызывать базу данных из разных мест, можно иметь один объект класса **Database**. Затем, создать заглушку **IDatabase** и использовать ее в тестах. Также можно создать опцию пробега теста с заглушкой и без нее, таким образом вы сможете тестировать все компоненты независимо в юнит тестах, и провести интеграционные тесты с использованием реальной базы данных.

Функционал, для которого нужна заглушка стремится к внутренней связанности в одном объекте. Предоставляя единую функциональность в одном объекте, вы, тем временем уменьшаете связанность всей системы вцелом.

 Моки (mock) объектов

Методика замены объекта тестовым макетом, который подтверждает ожидания, например, утверждение того, что метод был вызван, называется *моками (mocking)*.

Можно использовать *мок объект* "как наблюдателя, который используется, для подтверждения косвенных выводов данных тестируемой системы в то время как она выполняется. Обычно, моки включают в себя функциональность заглушек, в том, что они должны возвращать значения в тестируемую систему, если она ещё не провалила тест. Но упор делается на косвенных выводах информации. Поэтому мок объекты это больше чем заглушка плюс утверждение состояния - это фундаментально другой метод тестирования".

Например, предположим, что необходимо протестировать что верный метод **update()** из примера вызывается в объекте, обозреваемом другим объектом. [Пример 10.10, «Классы Subject и Observer из части тестируемой системы»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.mock-objects.examples.SUT.php) показывает код для классов **Subject** и **Observer**, которые являются частью тестируемой системы.

**Пример 10.10. Классы Subject и Observer из части тестируемой системы**

<?php

class Subject

{

protected $observers = array();

public function attach(Observer $observer)

{

$this->observers[] = $observer;

}

public function doSomething()

{

// Что-то происходит.

// ...

// 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);

}

}

// Другие методы.

}

class Observer

{

public function update($argument)

{

// Сделать что-то.

}

public function reportError($errorCode, $errorMessage, Subject $subject)

{

// Сделать что-то

}

// Другие методы.

}

?>

[Пример 10.11, «Тестирование того что метод будет вызван один раз и будут переданы указанные параметры»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.mock-objects.examples.SubjectTest.php) показывает как использовать моки для тестирования взаимодействия объектов **Subject** и **Observer**.

Сперва используется метод **getMock()**, предоставляемый классом **PHPUnit\_Framework\_TestCase**, для создания мока класса **Observer**. Так как во втором (необязательном) параметре метода **getMock()** передаётся массив, на реализацию мока будет заменён только метод **update()** из класса **Observer**.

**Пример 10.11. Тестирование того что метод будет вызван один раз и будут переданы указанные параметры**

<?php

class SubjectTest extends PHPUnit\_Framework\_TestCase

{

public function testObserversAreUpdated()

{

// Создает мок класса Observer,

// будет заменен только метод update().

$observer = $this->getMock('Observer', array('update'));

// Настроить что ожидается что метод update()

// будет вызван один раз и будет передан аргумент 'something'

$observer->expects($this->once())

->method('update')

->with($this->equalTo('something'));

// Создать экземпляр класса Subject и прикрепить мок класса

// Observer к нему.

$subject = new Subject;

$subject->attach($observer);

// Вызвать метод doSomething() объекта $subject,

// который должен вызвать мок метода update() из мока класса

// Observer с параметром 'something'.

$subject->doSomething();

}

}

?>

Метод **with()** принимает любое количество аргументов, соответствующих аргументам метода мока. Можно настроить более продвинутые ограничения, нежели обычное совпадения. Например, **$this->greaterThan(0)** указывает что тест ожидает что аргумент мока вызовется с числом, большим нуля.

**Пример 10.12. Тестирование того что мок метод будет вызван с определенным количеством аргументов, ограниченых разными способами**

<?php

class SubjectTest extends PHPUnit\_Framework\_TestCase

{

public function testErrorReported()

{

// Создаёт мок класса Observer, мок будет реализован

// только для метода reportError()

$observer = $this->getMock('Observer', array('reportError'));

$observer->expects($this->once())

->method('reportError')

->with($this->greaterThan(0),

$this->stringContains('Something'),

$this->anything());

$subject = new Subject;

$subject->attach($observer);

// Вызов метод doSomethingBad() должен вызвать метод reportError()

$subject->doSomethingBad();

}

}

?>

[Таблица 4.3, «Условия»](http://yapro.ru/web-master/php/writing-tests-for-phpunit.html#writing-tests-for-phpunit.assertions.assertThat.tables.constraints) показывает все ограничения, которые можно наложить на аргументы метода мока и [Таблица 10.1, «Методы совпадения»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.mock-objects.tables.matchers) показывает методы определяющие совпадения, предназначеные для определения количества вызовов.

**Таблица 10.1. Методы совпадения**

**Методы совпадения**

**Описание**

**PHPUnit\_Framework\_MockObject\_Matcher\_AnyInvokedCount any()**

Возвращает метод соответствия (matcher), который срабатывает при любом количестве вызовов.

**PHPUnit\_Framework\_MockObject\_Matcher\_InvokedCount never()**

Возвращает метод соответствия (matcher), который проверяет что метод никогда не был вызван.

**PHPUnit\_Framework\_MockObject\_Matcher\_InvokedAtLeastOnce atLeastOnce()**

Возвращает метод соответствия (matcher), который проверяет что метод был вызван хотя-бы один раз.

**PHPUnit\_Framework\_MockObject\_Matcher\_InvokedCount once()**

Возвращает метод соответствия (matcher), который проверяет что метод был вызван только один раз.

**PHPUnit\_Framework\_MockObject\_Matcher\_InvokedCount exactly(int $count)**

Возвращает метод соответствия (matcher), который проверяет что метод был вызван **$count** раз.

**PHPUnit\_Framework\_MockObject\_Matcher\_InvokedAtIndex at(int $index)**

Возвращает метод соответствия (matcher), который проверяет дальнейшие условия после достижения **$index** вызовов.

Метод **getMockForAbstractClass()** возвращает объект мока для абстрактного класса. Все абстрактные методы класса будут пустышками. Это позволяет протестировать реализованные методы класса.

**Пример 10.13. Тестирование реализованых методов абстрактного класса**

<?php

abstract class AbstractClass

{

public function concreteMethod()

{

return $this->abstractMethod();

}

public abstract function abstractMethod();

}

class AbstractClassTest extends PHPUnit\_Framework\_TestCase

{

public function testConcreteMethod()

{

$stub = $this->getMockForAbstractClass('AbstractClass');

$stub->expects($this->any())

->method('abstractMethod')

->will($this->returnValue(TRUE));

$this->assertTrue($stub->concreteMethod());

}

}

?>

**Пример 10.14. Проверка того что метод был вызван единожды с объектом идентичным заданному**

<?php

class FooTest extends PHPUnit\_Framework\_TestCase

{

public function testIdenticalObjectPassed()

{

$expectedObject = new stdClass;

$mock = $this->getMock('stdClass', 'foo');

$mock->expects($this->once())

->method('foo')

->with($this->identicalTo($expectedObject));

$mock->foo($expectedObject);

}

}

?>

**Пример 10.15. Создание мок-объекта с клонированием параметров**

<?php

class FooTest extends PHPUnit\_Framework\_TestCase

{

public function testIdenticalObjectPassed()

{

$enableParameterCloning = true;

$mock = $this->getMock(

'stdClass',

array(),

array(),

'',

TRUE,

TRUE,

TRUE,

$enableParameterCloning

);

// now your mock clones parameters so the identicalTo constraint will fail.

}

}

?>

 Заглушки и моки для веб сервисов

Когда в приложении есть взаимодействие с внешними веб сервисами - необходимо тестировать систему без обращения к ним. Чтобы сделать реализацию заглушек и моков простой, используется метод **getMockFromWsdl()**. Он используется таким же способом, как и **getMock()** (см. выше). Единственное отличие состоит в том, что **getMockFromWsdl()** возвращает заглушку или мок опираясь на описание WSDL, когда **getMock()** возвращает заглушку или мок, опираясь на код PHP класса или интерфейса.

[Пример 10.16, «Заглушка для веб сервиса»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.stubbing-and-mocking-web-services.examples.GoogleTest.php) показывает как **getMockFromWsdl()** может быть использован для создания заглушки сервиса описанном в **GoogleSearch.wsdl**.

**Пример 10.16. Заглушка для веб сервиса**

<?php

class GoogleTest extends PHPUnit\_Framework\_TestCase

{

public function testSearch()

{

$googleSearch = $this->getMockFromWsdl(

'GoogleSearch.wsdl', 'GoogleSearch'

);

$directoryCategory = new StdClass;

$directoryCategory->fullViewableName = '';

$directoryCategory->specialEncoding = '';

$element = new StdClass;

$element->summary = '';

$element->URL = 'http://www.phpunit.de/';

$element->snippet = '...';

$element->title = '<b>PHPUnit</b>';

$element->cachedSize = '11k';

$element->relatedInformationPresent = TRUE;

$element->hostName = 'www.phpunit.de';

$element->directoryCategory = $directoryCategory;

$element->directoryTitle = '';

$result = new StdClass;

$result->documentFiltering = FALSE;

$result->searchComments = '';

$result->estimatedTotalResultsCount = 378000;

$result->estimateIsExact = FALSE;

$result->resultElements = array($element);

$result->searchQuery = 'PHPUnit';

$result->startIndex = 1;

$result->endIndex = 1;

$result->searchTips = '';

$result->directoryCategories = array();

$result->searchTime = 0.248822;

$googleSearch->expects($this->any())

->method('doGoogleSearch')

->will($this->returnValue($result));

/\*\*

\* $googleSearch->doGoogleSearch() вернет заглушку и

\* метод веб сервиса doGoogleSearch() не будет вызван.

\*/

$this->assertEquals(

$result,

$googleSearch->doGoogleSearch(

'00000000000000000000000000000000',

'PHPUnit',

0,

1,

FALSE,

'',

FALSE,

'',

'',

''

)

);

}

}

?>

 Моки для файловой системы

[vfsStream](http://yapro.ru/system/redirect?to=http://code.google.com/p/bovigo/wiki/vfsStream) - это [обёртка потока (stream wrapper)](http://yapro.ru/system/redirect?to=http://ru2.php.net/streams) для создания [виртуальной файловой системы](http://yapro.ru/system/redirect?to=http://ru.wikipedia.org/wiki/Virtual_file_system), которая может быть использована в юнит тестах для файловой системы.

Для того чтобы установить vfsStream, необходимо использовать PEAR канал (

pear.php-tools.net

). Канал необходимо зарегистрировать в локальном окружении PEAR:

pear channel-discover pear.php-tools.net

Это необходимо выполнить только один раз. А затем можно использовать PEAR для установки vfsStream:

pear install pat/vfsStream-beta

[Пример 10.17, «Класс, который взаимодействует с файловой системой»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.mocking-the-filesystem.examples.Example.php) показывает пример класса, который взаимодействует с файловой системой.

**Пример 10.17. Класс, который взаимодействует с файловой системой**

<?php

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);

}

}

}?>

Без использования виртуальной файловой системы, такой как vfsStream, нельзя протестировать метод **setDirectory()** без изоляции от внешних факторов (см. [Пример 10.18, «Тестирование класса, который взаимодействует с файловой системой»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.mocking-the-filesystem.examples.ExampleTest.php)).

**Пример 10.18. Тестирование класса, который взаимодействует с файловой системой**

<?php

require\_once 'Example.php';

class ExampleTest extends PHPUnit\_Framework\_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');

}

}

}

?>

Подход, описаный выше, имеет несколько недостатков:

Как и с любым внешним ресурсом, файловая система может быть недоступна в какой-либо промежуток времени. Это может дать случайные падения тестов. (Вообще-то эти ошибки тоже надо тестировать в этом кейсе, но VFS бывает полезной при тестировании "идеальной ФС" которая стабильна, или тесты на ошибки были сделаны где-то в другом месте (прим. пер.)

В методах **setUp()** и **tearDown()** необходимо удостоверится, что директория не существует до и после таеста.

Если выполнение теста прерывается до выполнения метода **tearDown()** - директория может остаться на файловой системе.

[Пример 10.19, «Мок файловой системы в классе, который взаимодействует с ней»](http://yapro.ru/web-master/php/test-doubles.html#test-doubles.mocking-the-filesystem.examples.ExampleTest2.php) показывает пример использования vfsStream для мока файловой системы во время тестирования класса, взаимодействующего с файловой системой.

**Пример 10.19. Мок файловой системы в классе, который взаимодействует с ней**

<?php

require\_once 'vfsStream/vfsStream.php';

require\_once 'Example.php';

class ExampleTest extends PHPUnit\_Framework\_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'));

}

}

?>

Подход, описанный выше имеет несколько плюсов:

Тест получается более кратким.

vfsStream дает разработчику полный контроль над поведением и окружением файловой системы.

Так как операции над файловой системой в действительности не взаимодействуют с ней, то больше нет необходимости в операции очистки в методе **tearDown()**.