# Модульные тесты – кратко

Сначала для удобства установим расширение [Verify](https://github.com/Codeception/Verify). Которое сделает код наших тестов более читаемым. Используем для этого Composer:

$ composer require "codeception/verify:\*"

После добавления создаем базовые классы тестов:  
codecept build

Здесь стоит заметить что у Codeception есть команда для генерации тестов. Давайте ей воспользуемся - создадим файл для тестирования модели продукта:

codecept generate:phpunit unit Product

Файлы с тестами находятся в папке test\codeseption\unit

Теперь у нас есть файл tests/unit/ProductTest.php с которым мы можем работать как PHPUnit тестом. Например:

class ProductTest extends \PHPUnit\_Framework\_TestCase

{

protected $product;

protected function setUp()

{

$this->product = new Product();

}

protected function tearDown()

{

$this->product = null;

}

/\*\*

\* Проверка на обязательные поля.

\*/

public function testRequiredFields()

{

$this->product->name = '';

$this->product->price = '';

verify($this->product->save())->false();

$this->product->name = 'имя продукта';

$this->product->price = 5000;

verify($this->product->save())->true();

}

/\*\*

\* Проверка на получение категории.

\*/

public function testGetCategory()

{

$category = $this->product->getCategory();

verify($category)->isInstanceOf('Category');

verify($category->id)->equals(1);

}

}

Запускаем модульные тесты командой:

codecept run unit

При успешном выполнении увидим такой результат:

Unit Tests (2) ------------------------------------✔ ProductTest: Required fields (0.05s)

✔ ProductTest: Get category (0.02s)

---------------------------------------------------

Time: 934 ms, Memory: 11.50MB

OK (2 tests, 4 assertions)

## Создание тестов

Codeception имеет набор простых генераторов для того чтобы упростить создание тестов.  
Для начала вы можете создать классический PHPUnit тест унаследованный от класса \PHPUnit\_Framework\_TestCase.  
Это можно сделать при помощи следующей команды:

codecept generate:phpunit unit Example

Codeception имеет собственные дополнения к стандартным модульным тестам. Давайте попробуем создать их.  
Для создания модульных тестов дополненных Codeception используется другая команда.

$ codecept generate:test unit Example

Обе команды создадут новый файл ExampleTest расположенный в каталоге tests/unit.

Тест созданный командой generate:test будет выглядеть так:

**<?php  
  
  
class** ExampleTest **extends** \Codeception\TestCase\Test  
{  
 */\*\*  
 \** ***@var*** *\UnitTester  
 \*/* **protected** $tester;  
  
 **protected function** \_before()  
 {  
 }  
  
 **protected function** \_after()  
 {  
 }  
  
 // tests  
 **public function** testMe()  
 {  
  
 }  
  
}

Этот класс имеет два предопределенных метода в начале \_before и \_after. Вы можете использовать их для создания объектов перед началом выполнения тестов и их удаления после того как тест будет выполнен.

Как вы могли заметить PHPUnit методы setUp/tearDown заменены на методы: \_before, \_after.  
Настоящие методы setUp и tearDown реализованы внутри родительского класса: \Codeception\TestCase\Test и используются для подключения файла загрузчика (bootstrap) (По умолчанию \_bootstrap.php) а так же для настройки класса codeGuy что позволяет использовать все возможности Cept-файлов внутри модульных тестов. Точно так же как и в других видах тестов, вы можете настроить модули специфичные для класса CodeGuy в конфигурационном файле unit.suite.yml.  
Таким образом, если вы реализуете методы setUp и tearDown не забудьте вызвать их родительские методы внутри своих.

# Codeception Test Suite Configuration

# suite for unit (internal) tests.

class\_name: CodeGuy

modules:

enabled: [Unit, CodeHelper]

### Тестирование баз данных

Вероятно нет особо полезных модулей которые должны быть установлены для класса CodeGuy по умолчанию. Все потому, что класс CodeGuy по большей части используется для scenario-driven модульных тестов, описанных в следующих разделах. Однако в этом нет ни чего страшного, мы легко можем добавить необходимые модули. К примеру мы можем добавить модуль Db для тестирования данных в БД.

# Codeception Test Suite Configuration

# suite for unit (internal) tests.

class\_name: CodeGuy

modules:

enabled: [Unit, Db, CodeHelper]

После выполнения комманды "сборки":

codecept build

Новые методы будут добавлены в класс CodeGuy. Таким образом теперь вы можете использовать в ваших тестах методы для работы с БД:

<?php

function testSavingUser()

{

$user = new User();

$user->setName('Miles');

$user->save();

$this->codeGuy->seeInDatabase('users',array('name' => 'Miles'));

}

?>

База данных будет очищена и заполнена после прохождения каждого теста, точно так же как это происходит в функциональных и приемочных тестах.  
Если такое поведения модуля Db вам не подходит для текущего набора тестов, просто смените его настройки.

### Модули

добавлено в 1.5.2

Codeception позволяет получить доступ к свойствам и методам любого из модулей определенных для данного набора. Вместо использования класса CodeGuy, использование прямого доступа к модулям позволяет получить доступ к любым публичным свойствам конкретного модуля.

К примеру, если вы используете модуль Symfony2 обратится к Symfony контейнеру можно следующим способом:

<?php

/\*\*

\* @var Symfony\Component\DependencyInjection\Container

\*/

$container = $this->getModule('Symfony2')->container;

?>

Все публичные свойства модулей описаны в соответствующих разделах справки.

### Начальная загрузка (Bootstrap)

Файл загрузки (bootstrap) расположен в директории набора тестов и имеет название \_bootstrap. Данный файл **подвлючается перед выполнением каждого теста** (с помощью метода setUp родительского класса). Он широко используется в приемочных (acceptance) и функциональных (functional) тестах для инициализации различных предустановленных переменных. В модульных тестах он может быть полезен при использовании одинаковых данных с разными тестами. Однако его главное предназначение - установка автозагрузчика для вашего проекта внутри данного класса. Иначе Codeception не сможет найти тестируемые классы и тесты не выполнятся.

### Заглушки (Stubs)

В первой строке класса подключается класс Strub для использования заглушек в тестах. Заглушки (strubs) позволяют вам легко подменять реальные классы их муляжами.  
Не тратьте ваше время на указание большого количетва параметров в конструкторе просто вызовите Stub::make для содания класса заглушки.

Заглушки (Stubs) созданы на базе PHPUnit's mocking framework. Изучите [PHPUnit Manual](http://www.phpunit.de/manual/3.6/en/test-doubles.html) если вы нуждаетесь в дополнительной информации на тему заглушек (stubs or mocks).

Полное описание класса для работы с заглушками (stub util class) находится [здесь](http://allframeworks.ru/docs/reference/stubs).

### Соединим полученные знания вместе!

Меньше слов, больше кода для лучшего усвоения материала.

user = new User();

}

public function testUserCanBeBanned()

{

$this->user->setIsBanned(true);

$this->user->setUpdatedBy(Stub::make('User', array('name' => 'admin')));

$this->user->save();

$this->codeGuy->seeInDatabase('users', array('name' => 'Miles', 'is\_banned' => true));

}

}

?>

Чтобы протестировать класс, написанный на PHP, с использованием фреймворка PHPUnit, необходимо создать тестовый класс, расширяющий базовый класс PHPUnit\_Framework\_TestCase. Затем создать в этом классе публичные методы, начинающиеся со слова test (если создать метод, который будет называться по-другому, он не будет автоматически вызван при прогоне тестов), и поместить в них код, выполняющий действия с объектами тестируемого класса и проверяющий результат. На этом можно закончить и скормить полученный класс phpunit, который, в свою очередь, последовательно вызовет все тестовые методы и любезно предоставит отчет об их работе. Однако в большинстве случаев в каждом из тестовых методов будет повторяющийся код, подготавливающий систему для работы с тестируемым объектом. Для того, чтобы избежать дублирования кода, в классе PHPUnit\_Framework\_TestCase созданы защищенные методы setUp и tearDown, имеющие пустую реализацию. Эти методы вызываются перед и после запуска очередного тестового метода соответственно и служат для подготовки системы к выполнению тестовых действий и очистки ее после завершения каждого теста. В тестовом классе, расширяющем PHPUnit\_Framework\_TestCase, можно переопределить эти методы и поместить повторяющийся ранее в каждом тестовом методе код в них. В результате последовательность вызова методов при прогонке тестов будет следующая:

SetUPBeforeClass() /\* Это статический метод вызываемый один раз при создании сласса

setUp() {/\* Установили систему в нужное состояние \*/}

testMethod1() {/\* протестировали метод 1 класса \*/}

tearDown() {/\* Очистили систему \*/}

setUp() {/\* Установили систему в нужное состояние \*/}

testMethod2() {/\* протестировали метод 2 класса \*/}

tearDown() {/\* Очистили систему \*/}

…

setUp() {/\* Установили систему в нужное состояние \*/}

testMethodN() {/\* протестировали метод N класса \*/}

tearDown() {/\* Очистили систему \*/}

tearDown BeforeClass() /\* Это статический метод вызываемый один раз при удалении класса

Пример файла с тестам использующим Codeception

В файле \_bootstrap.php необходимо вставить строку

ini\_set('xdebug.max\_nesting\_level', 1000);

и добавить памяты выделяемой системой memorylimit = 512Mb

Фукция specify() снимает слепок с системы, выполняет тесты и затем восстанавливает состояние

<?php

namespace tests\unit;

use app\models\DbUser;

use Codeception\Specify;

use tests\codeception\fixtures\UserFixture;

use Yii;

use yii\codeception\DbTestCase;

/\*\*

\* @property array $users

\*/

class DbUserTest extends DbTestCase

{

use Specify;

/\*\*

\* @var \UnitTester

\*/

protected $tester;

/\*\*

\* @var DbUser

\*/

private $user;

public function \_before()

{

$this->user = new DbUser();

}

public function testValidation()

{

$this->specify('fields are required', function() {

$this->user->username = null;

$this->user->email = null;

expect('model is not valid', $this->user->validate())->false();

expect('username has error', $this->user->getErrors())->hasKey('username');

expect('email has error', $this->user->getErrors())->hasKey('email');

});

$this->specify('fields are wrong', function() {

$this->user->username = 'Test DbUser';

$this->user->email = 'test\_email.com';

expect('model is not valid', $this->user->validate())->false();

expect('username has error', $this->user->getErrors())->hasKey('username');

expect('email has error', $this->user->getErrors())->hasKey('email');

});

$this->specify('fields are unique', function() {

$this->user->username = $this->users[0]['username'];

$this->user->email = $this->users[0]['email'];

expect('model is not valid', $this->user->validate())->false();

expect('username has error', $this->user->getErrors())->hasKey('username');

expect('email has error', $this->user->getErrors())->hasKey('email');

});

$this->specify('fields are correct', function() {

$this->user->username = 'TestDbUser';

$this->user->email = 'test@email.com';

expect('model is not valid', $this->user->validate())->true();

});

}

public function testSaveIntoDatabase()

{

$user = new DbUser([

'username' => 'TestDbUsername',

'email' => 'test@email.com',

]);

expect('model is saved', $user->save())->true();

}

public function fixtures()

{

return [

'users' => [

'class' => UserFixture::className(),

//'dataFile' => '@tests/codeception/fixtures/data/user.php',

],

];

}

}

Для модульного тестирования, как и для других типов, в фреймворке Codeception существует два способа написания тестов, которым соответствуют два типа наименования файлов: Cept и Cest. **Cept** строится по сценарию (функциональное программирование), а **Cest** на базе объектов (ООП). Кроме того, Codeception может выполнять оригинальные **PHPUnit**-тесты для модульного тестирования т.к. наследуется от данной библиотеки.

##### Создание по типу cept:

codecept generate:cept unit Name

где «Name» - названия файла теста.  
  
Появится файл **tests\unit\NameCept.php** с содержимым:

<?php

$I = new UnitTester($scenario);

$I->wantTo('perform actions and see result');

##### Создание по типу cest:

codecept generate:cest unit Name

Появится файл **tests\unit\OneCest.php** с содержимым:

<?php

class OneCest

{

public function \_before(UnitTester $I)

{

}

public function \_after(UnitTester $I)

{

}

// tests

public function tryToTest(UnitTester $I)

{

}

}

Каждый публичный метод, не начинающийся с нижнего подчеркивания, будет выполнен как тест.  
  
Это обычные для Codeception типы организации тестов.   
Для использования стандартных **PHPUnit** утверждений (asserts), должен быть включен модуль «Asserts» в конфигурационном файле tests\unit.suite.yml (по-умолчанию уже включен).  
  
Доступ к утверждениям (asserts) можно получить с помощью переменной **$I**:

$I->assertTrue(true);

Кроме того есть возможность последним аргументом указывать текстовые сообщения, которые, во-первых, делают более читаемыми теста, а во-вторых отобразятся в консоли в случае провала указанного теста:

$I->assertTrue(true, 'message');

Тип cest может использовать события \_before и \_after непосредственно в тестирующем классе. Эти и другие события, для обоих типов тестов могут быть использованы в помощнике **tests\\_support\Helper\Unit.php**  
Список событий можно посмотреть по [ссылке](http://codeception.com/docs-2.0/03-ModulesAndHelpers).   
Правда нет полного аналога методу **setUpBeforeClass()**, код которого выполнялся бы перед выполнением тестов из каждого отдельного класса. Можно использовать **\_beforeSuite()**, код которого выполнится только один раз до начала тестирования.

##### Создание тестов по типу phpUnit:

codecept generate:test unit Name

Появится файл **tests\unit\NameTest.php** с содержимым:

<?php

class TwoTest extends \Codeception\Test\Unit

{

/\*\*

\* @var \UnitTester

\*/

protected $tester;

protected function \_before()

{

}

protected function \_after()

{

}

// tests

public function testSomeFeature()

{

}

}

В данном случае, создается класс, который по цепочке наследования наследуется от **phpUnit** и поэтому тесты можно писать в стиле этого фреймворка и используя его методы. При необходимости, можно перенести ваши готовые тесты из phpUnit целиком. Только если вы использовали доп.пакет **phpunit/dbunit** для работы с базой данных в тестах phpUnit, не забываем подключить данный пакет в классе теста. Для для этого достаточно подключить трейт:

use PHPUnit\_Extensions\_Database\_TestCase\_Trait;

или

use PHPUnit\DbUnit\TestCaseTrait;

в зависимости от версии phpUnit. И папку **\_data** с xml файлом для заполнения данными тестовой таблицы нужно скопировать в папку **tests\unit**.  
  
Методы **\_before()** и **\_after()** вызывают методы **setUp()** и **tearDown()** у **phpUnit**, поэтому в них так же можно писать код который выполняется перед каждым методом в тестовом классе и после каждого метода соответственно.   
При данном способе написания модульных тестов можно использовать все методы phpUnit, например **setUpBeforeClass()**.  
  
Так же как и для других типов тестов есть возможность подключать и использовать модули Codeception. Таким образом можно использовать тот же метод seeInDatabase() модуля Db для проверки наличия данных в БД (подробнее [тут](http://klisl.com/codeception_db.html)).   
  
В своей статье "Основы PHPUnit - 1 часть." и я писал как создаются модульные тесты. Codeception выступает в качестве обертки для PHPUnit. Поэтому подробнее ознакомиться с модульным тестированием и посмотреть примеры можно по [ссылке](http://klisl.com/phpunit_basics_1.html).

#### Работа с имитирующими объектами в Codeception.

Для удобства использования, **Codeception** имеет свою обертку поверх стандартной работы с имитирующими объектами в **phpUnit**. Использовать ее можно во всех типах модульных тестов Codeception. А почитать про имитирующие объекты в phpUnit можно [тут](http://klisl.com/phpunit_mock.html).  
  
Подключаем класс Stub в нужном файле теста:

use Codeception\Util\Stub;

##### Методы создающие имитирующий объект без выполнения метода конструктора класса.

* Метод **make()** позволяет переопределить нужное свойство или метод. Остальные свойства и методы остаются без изменений. Не забываем, что метод конструктора не будет выполнен, а значит некоторые свойства объекта могут быть не инициализированы, если это предусмотрено в конструкторе.

Переопределяем свойство name класса **User**:

$user = Stub::make(\app\User::class, ['name' => null]);

Переопределяем метод **readAll()** класса **User**:

$user = Stub::make(\app\User::class, ['readAll' =>function(){

return true;

}]);

* Метод **makeEmpty()**:

- создает класс, все методы которого возвращают null.  
- конструктор не запускается.  
- свойства и методы могут быть установлены как второй параметр.  
- можно установить даже защищенные и частные свойства.

// создаем пустой класс с указанными свойствами

$user = Stub::makeEmpty(\app\User::class, array('name' => 'Serj'));

* Метод **makeEmptyExcept()**:

- создает класс, все методы которого возвращают null, кроме одного указанного.  
- конструктор не запускается.  
- свойства и методы могут быть установлены как третий параметр.  
- можно установить даже защищенные и частные свойства.

$user = Stub::makeEmptyExcept(\app\User::class, 'read', array('name' => 'Serj'));

т.е. в данном примере метод read будет выполнять код указанный в классе User, тогда как другие методы будут возвращать null.

##### Методы создающие имитирующий объект с выполнением метода конструктора класса.

Есть несколько схожих методов, которые создают имитирующий объект с запуском конструктора класса: Stub::construct, Stub::constructEmpty, Stub::constructEmptyExcep в остальном они работают аналогично методам типа “make”.

* Метод **construct()** позволяет переопределить указанные свойства и методы.
* Метод **constructEmpty()** создает объект класса, все методы которого возвращают null и переопределить указанные свойства/методы.
* Метод **constructEmptyExcep()** создает объект класса, все методы которого возвращают null кроме одного указанного метода, который будет выполнять свой оригинальный код. так же есть возможность переопределить нужные свойства/методы.

Их использование позволяет, выполнив код конструктора, инициализировать указанные в нем свойства, и при этом, в зависимости от выбора одного из этих методов, переопределить нужные свойства/методы или расставить заглушки (значения null).  
  
Например создадим имитирующий объект на базе класса **User** аналогичный оригиналу но с переопределенным методом **readAll()**:

$db = new \app\Db();

$user['name'] = 'Test';

$user['email'] = 'test@gmail.com';

$user = Stub::construct(\app\User::class, array($db, $user), array('readAll' => function(){

return ['name' => 'Sveta', 'email' => 'sveta@mail.ru'];

}));

Вместо названия класса, в качестве первого аргумента данным методам можно указывать объект данного класса.

* Метод **Stub::update** позволяет переопределить свойства/методы у созданного ранее имитирующего объекта.

Например:

$db = new \app\Db();

$user['name'] = 'Evgen';

$user['email'] = 'evgen@gmail.com';

$user = Stub::constructEmpty(\app\User::class, array($db, $user));

Stub::update($user, [

'name' => 'newName',

'email' => 'newNamegmail.com'

]);

Это позволяет «по-быстрому» внести изменения в существующий имитирующий объект. Так же можно добавить новое свойство которое не указано в оригинальным классе/объекте.  
  
Еще пример – допустим создали пустой имитирующий объект у которого все методы являются заглушками (возвращают null):

$user = Stub::makeEmpty(\app\User::class);

а далее, для какой-то проверки понадобилось переопределить метод **readAll()**:

Stub::update($user, [

'readAll' => function(){

return true;

}

]);

Так же есть методы **never**, **once**, **exactly** и др. для уточнения того сколько раз должен вызываться указанный метод.   
Например укажем что метод **getName()** не должен быть вызван ни разу в процессе тестирования иначе выбросить исключение:

$user = Stub::make('User', array('getName' => Stub::never(), 'someMethod' => function() {}));

Подробное описание этих и других методов с примерами можно посмотреть  
по [ссылке](http://codeception.com/docs/reference/Stub) или в файле vendor\codeception\codeception\src\Codeception\Util\Stub.php

#### Синтаксический сахар Codeception.

Синтаксический сахар (англ. syntactic sugar) в языке программирования — это синтаксические возможности, применение которых не влияет на поведение программы, но делает использование языка более удобным для человека. Для этих целей предлагается установка двух пакетов:

* **codeception/verify**;
* **codeception/specify**.

Данные пакеты могут использоваться при написании модульных и интеграционных тестов в Codeception для удобства и расширения основного функционала PHPUnit.

##### codeception/verify

Глобальная установка (в каталог текущего пользователя):

composer global require --dev "codeception/verify"

Для установки в текущий проект:

composer require --dev "codeception/verify"

После установки сразу можно использовать данный синтаксис, как-то подключать не нужно.  
  
Сравните два теста, которые описаны ниже - первый с применением стандартного синтаксиса **PHPUnit**, а второй — с использованием пакета **verify**.

$this->assertEquals($user->getName(), 'test');

и

verify($user->getName())->equals('test');

Можно сначала указывать комментарий, который будет так же выведен при ошибке:

verify('сравниваем имя пользователя', $user->getName())->equals('test');

Вместо функции **verify()** можно использовать синоним:

expect()

с такими же аргументами.  
  
Методы утверждений (asserts) используемые данным пакетом схожи с теми, что использует **PHPUnit**. В примере вместо **assertEquals()** используется **equals()**. Чаще всего достаточно просто убрать приставку «assert»:  
assertTrue – true;assertFalse – false;assertNotNull – notNull;  
  
но есть и такие:  
assertEmpty - isEmpty  
  
Посмотреть соответствие методов можно в файле vendor\codeception\verify\src\Codeception\Verify.php.  
Еще [примеры](https://github.com/Codeception/Verify).

##### codeception/specify

Пакет позволяет писать несколько тестов PHPUnit в одном методе разбивая их на блоки. Код внутри **specify** блоков изолирован. Любое изменение свойств объектов и переменных не будет отражено в других блоках кода.  
Так же можно указать комментарий к каждому блоку теста.  
  
Установка:

composer global require --dev "codeception/specify"

для установки каталог vendor текущего проекта:

composer require --dev "codeception/specify"

После установки нужно подключить трейт в классе теста:

class UserTest extends \Codeception\Test\Unit

{

use \Codeception\Specify; // подключаем пакет Specify

...

Пример.

public function testReadAll()

{

$this->specify('при наличии данных в БД массив не пуст', function (){

$this->assertNotEmpty($this->user->readAll());

});

$this->specify('при пустой таблице возвращает пустой массив', function (){

//кол-во элементов в массиве данных должно быть = 0

$this->assertCount(0, $this->user->readAll());

});

}

При этом я заметил конфликт данного пакета с расширением для PHPUnit используемом для работы с БД - **DBUnit**.   
В данном случае возникает ошибка:  
[PHPUnit\_Framework\_Exception] ReflectionProperty::setValue(): Cannot assign to an array of nodes (duplicate subnodes or attr detected)