Reusing Test Code

Codeception uses modularity to create a comfortable testing environment for every test suite you write. Modules allow you to choose the actions and assertions that can be performed in tests.

What are Actors

All actions and assertions that can be performed by the Actor object in a class are defined in modules. It might look like Codeception limits you in testing, but that's not true. You can extend the testing suite with your own actions and assertions, by writing them into a custom module, called a Helper. We will get back to this later in this chapter, but for now let's look at the following test:

```
<?php
$I = new AcceptanceTester($scenario);
$I->amOnPage('/');
$I->see('Hello');
$I->seeInDatabase('users', ['id' => 1]);
$I->seeFileFound('running.lock');
```

It can operate with different entities: the web page can be loaded with the PhpBrowser module, the database assertion uses the Db module, and the file state can be checked with the Filesystem module.

Modules are attached to Actor classes in the suite config. For example, in `tests/acceptance.suite.yml` we should see:

```
actor: AcceptanceTester
modules:
    enabled:
        - PhpBrowser:
            url: http://localhost
        - Db
        - Filesystem
```

The AcceptanceTester class has its methods defined in modules. Let's see what's inside the `AcceptanceTester` class, which is located inside the `tests/_support` directory:

```
<?php
/**
 * Inherited Methods
 * @method void wantToTest($text)
 * @method void wantTo($text)
 * @method void execute($callable)
 * @method void expectTo($prediction)
 * @method void expect($prediction)
 * @method void amGoingTo($argumentation)
 * @method void am($role)
 * @method void lookForwardTo($achieveValue)
 * @method void comment($description)
 * @method void haveFriend($name, $actorClass = null)
 *
 * @SuppressWarnings(PHPMD)
*/
class AcceptanceTester extends \Codeception\Actor
{
    use _generated\AcceptanceTesterActions;

   /**
    * Define custom actions here
    */
}
```

The most important part is the `_generated\AcceptanceTesterActions` trait, which is used as a proxy for enabled modules. It knows which module executes which action and passes parameters into it. This trait was created by running `codecept build` and is regenerated each time module or configuration changes.

Authorization

It is recommended to put widely used actions inside an Actor class. A good example is the `login` action which would probably be actively involved in acceptance or functional testing:

```
<?php
class AcceptanceTester extends \Codeception\Actor
{
    // do not ever remove this line!
    use _generated\AcceptanceTesterActions;

    public function login($name, $password)
    {
        $I = $this;
        $I->amOnPage('/login');
        $I->submitForm('#loginForm', [
            'login' => $name,
            'password' => $password
        ]);
        $I->see($name, '.navbar');
    }
}
```

Now you can use the `login` method inside your tests:

```
<?php
$I = new AcceptanceTester($scenario);
$I->login('miles', '123456');
```

However, implementing all actions for reuse in a single actor class may lead to breaking the Single Responsibility Principle (http://en.wikipedia.org/wiki/Single_responsibility_principle).

Session Snapshot

If you need to authorize a user for each test, you can do so by submiting the login form at the beginning of every test. Running those steps takes time, and in the case of Selenium tests (which are slow by themselves) that time loss can become significant.

Codeception allows you to share cookies between tests, so a test user can stay logged in for other tests.

Let's improve the code of our `login` method, executing the form submission only once and restoring the session from cookies for each subsequent login function call:

```
<?php
    public function login($name, $password)
    {
        $I = $this;
        // if snapshot exists - skipping login
        if ($I->loadSessionSnapshot('login')) {
            return;
        }
        // logging in
        $I->amOnPage('/login');
        $I->submitForm('#loginForm', [
            'login' => $name,
            'password' => $password
        ]);
        $I->see($name, '.navbar');
         // saving snapshot
        $I->saveSessionSnapshot('login');
    }
```

Note that session restoration only works for `WebDriver` modules (modules implementing `Codeception\Lib\Interfaces\SessionSnapshot`).

StepObjects

StepObjects are great if you need some common functionality for a group of tests. Let's say you are going to test an admin area of a site. You probably won't need the same actions from the admin area while testing the front end, so it's a good idea to move these admin-specific tests into their own class. We call such a classes StepObjects.

Lets create an Admin StepObject with the generator:

```
php codecept generate:stepobject acceptance Admin
```

You can supply optional action names. Enter one at a time, followed by a newline. End with an empty line to continue to StepObject creation.

```
php codecept generate:stepobject acceptance Admin
Add action to StepObject class (ENTER to exit): loginAsAdmin
Add action to StepObject class (ENTER to exit):
StepObject was created in /tests/acceptance/_support/Step/Acceptance/Admin.php
```

This will generate a class in `/tests/_support/Step/Acceptance/Admin.php` similar to this:

```
<?php
namespace Step\Acceptance;

class Admin extends \AcceptanceTester
{
    public function loginAsAdmin()
    {
        $I = $this;
    }
}
```

As you see, this class is very simple. It extends the `AcceptanceTester` class, meaning it can access all the methods and properties of `AcceptanceTester`.

The `loginAsAdmin` method may be implemented like this:

```
<?php
namespace Step\Acceptance;

class Admin extends \AcceptanceTester
{
    public function loginAsAdmin()
    {
        $I = $this;
        $I->amOnPage('/admin');
        $I->fillField('username', 'admin');
        $I->fillField('password', '123456');
        $I->click('Login');
    }
}
```

In tests, you can use a StepObject by instantiating `Step\Acceptance\Admin` instead of `AcceptanceTester`:

```
<?php
use Step\Acceptance\Admin as AdminTester;

$I = new AdminTester($scenario);
$I->loginAsAdmin();
```

The same way as above, a StepObject can be instantiated automatically by the Dependency Injection Container when used inside the Cest format:

```
<?php
class UserCest
{
    function showUserProfile(\Step\Acceptance\Admin $I)
    {
        $I->loginAsAdmin();
        $I->amOnPage('/admin/profile');
        $I->see('Admin Profile', 'h1');
    }
}
```

If you have a complex interaction scenario, you may use several step objects in one test. If you feel like adding too many actions into your Actor class (which is AcceptanceTester in this case) consider moving some of them into separate StepObjects.

PageObjects

For acceptance and functional testing, we will not only need to have common actions being reused across different tests, we should have buttons, links and form fields being reused as well. For those cases we need to implement the PageObject pattern (http://docs.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern), which is widely used by test automation engineers. The PageObject pattern represents a web page as a class and the DOM elements on that page as its properties, and some basic interactions as its methods. PageObjects are very important when you are developing a flexible architecture of your tests. Do not hardcode complex CSS or XPath locators in your tests but rather move them into PageObject classes.

Codeception can generate a PageObject class for you with command:

```
php codecept generate:pageobject Login
```

This will create a `Login` class in `tests/_support/Page`. The basic PageObject is nothing more than an empty class with a few stubs. It is expected that you will populate it with the UI locators of a page it represents and then those locators will be used on a page. Locators are represented with public static properties:

```
<?php
namespace Page;

class Login
{
    public static $URL = '/login';

    public static $usernameField = '#mainForm #username';
    public static $passwordField = '#mainForm input[name=password]';
    public static $loginButton = '#mainForm input[type=submit]';
}
```

And this is how this page object can be used in a test:

```
<?php
use Page\Login as LoginPage;

$I = new AcceptanceTester($scenario);
$I->wantTo('login to site');
$I->amOnPage(LoginPage::$URL);
$I->fillField(LoginPage::$usernameField, 'bill evans');
$I->fillField(LoginPage::$passwordField, 'debby');
$I->click(LoginPage::$loginButton);
$I->see('Welcome, bill');
```

As you see, you can freely change markup of your login page, and all the tests interacting with this page will have their locators updated according to properties of LoginPage class.

But let's move further. The PageObject concept specifies that the methods for the page interaction should also be stored in a PageObject class. It now stores a passed instance of an Actor class. An AcceptanceTester can be accessed via the `AcceptanceTester` property of that class. Let's define a `login` method in this class:

```
<?php
namespace Page;

class Login
{
    public static $URL = '/login';

    public static $usernameField = '#mainForm #username';
    public static $passwordField = '#mainForm input[name=password]';
    public static $loginButton = '#mainForm input[type=submit]';

    /**
     * @var AcceptanceTester
     */
    protected $tester;

    public function __construct(\AcceptanceTester $I)
    {
        $this->tester = $I;
    }

    public function login($name, $password)
    {
        $I = $this->tester;

        $I->amOnPage(self::$URL);
        $I->fillField(self::$usernameField, $name);
        $I->fillField(self::$passwordField, $password);
        $I->click(self::$loginButton);

        return $this;
    }
}
```

And here is an example of how this PageObject can be used in a test:

```
<?php
use Page\Login as LoginPage;

$I = new AcceptanceTester($scenario);
$loginPage = new LoginPage($I);
$loginPage->login('bill evans', 'debby');
$I->amOnPage('/profile');
$I->see('Bill Evans Profile', 'h1');
```

If you write your scenario-driven tests in the Cest format (which is the recommended approach), you can bypass the manual creation of a PageObject and delegate this task to Codeception. If you specify which object you need for a test, Codeception will try to create it using the dependency injection container. In the case of a PageObject you should declare a class as a parameter for a test method:

```
<?php
class UserCest
{
    function showUserProfile(AcceptanceTester $I, \Page\Login $loginPage)
    {
        $loginPage->login('bill evans', 'debby');
        $I->amOnPage('/profile');
        $I->see('Bill Evans Profile', 'h1');
    }
}
```

The dependency injection container can construct any object that requires any known class type. For instance, `Page\Login` required `AcceptanceTester`, and so it was injected into `Page\Login` constructor, and PageObject was created and passed into method arguments. You should explicitly specify the types of required objects for Codeception to know what objects should be created for a test. Dependency Injection will be described in the next chapter.

Conclusion

There are lots of ways to create reusable and readable tests. Group common actions together and move them to an Actor class or StepObjects. Move CSS and XPath locators into PageObjects. Write your custom actions and assertions in Helpers. Scenario-driven tests should not contain anything more complex than `$I->doSomething` commands. Following this approach will allow you to keep your tests clean, readable, stable and make them easy to maintain.

- **Next Chapter: AdvancedUsage > (/docs/07-AdvancedUsage)**
- **Previous Chapter: < ModulesAndHelpers (/docs/06-ModulesAndHelpers)**

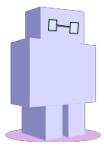

(https://www.jetbrains.com/phpstorm/) Rebilly (https://www.rebilly.com/)


(https://github.com/codeception/codeception) 
(https://twitter.com/codeception)


(http://www.facebook.com/pages/Codeception/288959711204412)

- Installation (/install)
- Credits (/credits)
- Releases (/changelog)
- Commercial Services (http://sdclabs.com/codeception?utm_source=codeception.com&utm_medium=bottom_menu&utm_term=link&utm_campaign=reference)
- License (https://github.com/Codeception/Codeception/blob/master/LICENSE)

Codeception Family



Robo

(http://robo.li)(http://robo.li)
Modern PHP **Task Runner**. Allows to declare tasks with zero configuration in pure PHP.

CodeceptJS

(http://codecept.io)(http://codecept.io)
Codeception for **NodeJS**. Write acceptance tests in ES6 and execute in webdriverio, Selenium WebDriver, and Protractor.