Behavior Driven Development

Behavior Driven Development (BDD) is a popular software development methodology. BDD is considered an extension of TDD, and is greatly inspired by Agile (http://agilemanifesto.org/) practices. The primary reason to choose BDD as your development process is to break down communication barriers between business and technical teams. BDD encourages the use of automated testing to verify all documented features of a project from the very beginning. This is why it is common to talk about BDD in the context of test frameworks (like Codeception). The BDD approach, however, is about much more than testing - it is a common language for all team members to use during the development process.

What is Behavior Driven Development

BDD was introduced by Dan North (https://dannorth.net/introducing-bdd/). He described it as:

> outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.

BDD has its own evolution from the days it was born, started by replacing “test” to “should” in unit tests, and moving towards powerful tools like Cucumber and Behat, which made user stories (human readable text) to be executed as an acceptance test.

The idea of story BDD can be narrowed to:

- describe features in a scenario with a formal text
- use examples to make abstract things concrete
- implement each step of a scenario for testing
- write actual code implementing the feature

By writing every feature in User Story format that is automatically executable as a test we ensure that: business, developers, QAs and managers are in the same boat.

BDD encourages exploration and debate in order to formalize the requirements and the features that needs to be implemented by requesting to write the User Stories in a way that everyone can understand.

By making tests to be a part of User Story, BDD allows non-technical personnel to write (or edit) Acceptance tests.

With this procedure we also ensure that everyone in a team knows what has been developed, what has not, what has been tested and what has not.

Ubiquitous Language

The ubiquitous language is always referred as *common* language. That is it's main benefit. It is not a couple of our business specification's words, and not a couple of developer's technical terms. It is a common words and terms that can be understood by people for whom we are building the software and should be understood by developers. Establishing correct communication between this two groups people is vital for building successful project that will fit the domain and fulfill all business needs.

Each feature of a product should be born from a talk between

- business (analysts, product owner)
- developers
- QAs

which are known in BDD as "three amigos".

Such talks should produce written stories. There should be an actor that doing some things, the feature that should be fulfilled within the story and the result achieved.

We can try to write such simple story:

```
As a customer I want to buy several products
I put first product with $600 price to my cart
And then another one with $1000 price
When I go to checkout process
I should see that total number of products I want to buy is 2
And my order amount is $1600
```

As we can see this simple story highlights core concepts that are called *contracts*. We should fulfill those contracts to model software correctly. But how we can verify that those contracts are being satisfied? Cucumber (http://cucumber.io) introduced a special language for such stories called **Gherkin**. Same story transformed to Gherkin will look like this:

```
Feature: checkout process
  In order to buy products
  As a customer
  I want to be able to buy several products

  Scenario:
    Given I have product with $600 price in my cart
    And I have product with $1000 price
    When I go to checkout process
    Then I should see that total number of products is 2
    And my order amount is $1600
```

Cucumber, Behat, and sure, **Codeception** can execute this scenario step by step as an automated test. Every step in this scenario requires a code which defines .

Gherkin

Let's learn some more about Gherkin format and then we will see how to execute it with Codeception:

Features

Whenever you start writing a story you are describing a specific feature of an application, with a set of scenarios and examples describing this feature.

Feature file is written in Gherkin format. Codeception can generate a feature file for you. We will assume that we will use scenarios in feature files for acceptance tests, so feature files to be placed in `acceptance` suite directory:

```
php codecept g:feature acceptance checkout
```

Generated template will look like this:

```
Feature: checkout
  In order to ...
  As a ...
  I need to ...

  Scenario: try checkout
```

This template can be fulfilled by setting actor and goals:

```
Feature: checkout
  In order to buy product
  As a customer
  I need to be able to checkout the selected products
```

Next, we will describe this feature by writing examples for it

Scenarios

Scenarios are live examples of feature usage. Inside a feature file it should be written inside a *Feature* block. Each scenario should contain its title:

```
Feature: checkout
  In order to buy product
  As a customer
  I need to be able to checkout the selected products

Scenario: order several products
```

Scenarios are written in step-by-step manner using Given-When-Then approach. At start, scenario should describe its context with **Given** keyword:

```
  Given I have product with $600 price in my cart
  And I have product with $1000 price in my cart
```

Here we also use word **And** to extend the Given and not to repeat it in each line.

This is how we described the initial conditions. Next, we perform some action. We use **When** keyword for it:

```
  When I go to checkout process
```

And in the end we are verifying our expectation using **Then** keyword. The action changed the initial given state, and produced some results. Let's check that those results are what we actually expect.

```
  Then I should see that total number of products is 2
  And my order amount is $1600
```

We can test this scenario by executing it in dry-run mode. In this mode test won't be executed (actually, we didn't define any step for it, so it won't be executed in any case).

```
$ codecept dry-run acceptance checkout.feature
```

```
checkout: order several products
Signature: checkout:order several products
Test: tests/acceptance/checkout.feature:order several products
Scenario --
 In order to buy product
 As a customer
 I need to be able to checkout the selected products
 Given i have product with $600 price in my cart
 And i have product with $1000 price in my cart
 When i go to checkout process
 Then i should see that total number of products is 2
 And my order amount is $1600

 INCOMPLETE
Step definition for `I have product with $600 price in my cart` not found in contexts
Step definition for `I have product with $1000 price` not found in contexts
Step definition for `I go to checkout process` not found in contexts
Step definition for `I should see that total number of products is 2` not found in contexts
Step definition for `my order amount is $1600` not found in contexts
Run gherkin:snippets to define missing steps
```

Besides the scenario steps listed we got the notification that our steps are not defined yet. We can define them easily by executing `gherkin:snippets` command for the given suite:

```
codecept gherkin:snippets acceptance
```

This will produce code templates for all undefined steps in all feature files of this suite. Our next step will be to define those steps and transforming feature-file into valid test.

Step Definitions

To match steps from a feature file to PHP code we use annotation which are added to class methods. By default Codeception expects that all methods marked with `@Given`, `@When`, `@Then` annotation. Each annotation should contain a step string.

```
/** @Given I am logged as admin */
```

Steps can also be matched with regex expressions. This way we can make more flexible steps

```
/** @Given /I am (logged|authorized) as admin/  */
```

Please note that regular expressions should start and end with `/` char. Regex is also used to match parameters and pass them as arguments into methods.

```
<?php
/** @Given /I am (?:logged|authorized) as "(\w+)"/  */
function amAuthorized($role)
{
  // logged or authorized does not matter to us
  // so we added ?: for this capture group
}
```

Parameters can be also passed in non-regex strings using ":" params placeholder.

```
/** @Given I am logged in as :role */
```

This will match any word (passed in double quotes) or a number passed:

```
Given I am logged in as "admin"
Given I am logged in as 1
```

Steps are defined in Context files. Default context is an actor class, i.e. for acceptance testing suite default context is `AcceptanceTester` class. However, you can define steps in any classes and include them as contexts. It is useful to define steps in StepObject and PageObject classes.

To list all defined steps run `gherkin:steps` command:

```
codecept gherkin:steps
```

Testing Behavior

As it was mentioned, feature files is not just a user story. By writing features in formal language called Gherkin we can execute those scenarios as automated tests. There is no restrictions in the way how those scenarios are supposed to be tested. Tests can be executed at functional, acceptance, or domain level. However, we will concentrate on acceptance or UI tests in current guide.

Acceptance Testing

As we generated snippets for missing steps with `gherkin:snippets` command, we will define them in `AcceptanceTester` file.

```
<?php
class AcceptanceTester extends \Codeception\Actor
{
    use _generated\AcceptanceTesterActions;

    /**
     * @Given I have product with :num1 price in my cart
     */
     public function iHaveProductWithPriceInMyCart($num1)
     {
        throw new \Codeception\Exception\Incomplete("Step `I have product with :num1 price in my cart` is not defined");
     }

    /**
     * @When I go to checkout process
     */
     public function iGoToCheckoutProcess()
     {
        throw new \Codeception\Exception\Incomplete("Step `I go to checkout process` is not defined");
     }

    /**
     * @Then I should see that total number of products is :num1
     */
     public function iShouldSeeThatTotalNumberOfProductsIs($num1)
     {
        throw new \Codeception\Exception\Incomplete("Step `I should see that total number of products is :num1` is not defined");
     }

    /**
     * @Then my order amount is :num1
     */
     public function myOrderAmountIs($num1)
     {
        throw new \Codeception\Exception\Incomplete("Step `my order amount is :num1` is not defined");
     }
}
```

Please note that `:num1` placeholder can be used for strings and numbers (may contain currency sign).
In current case `:num1` matches `$600` and `$num1` is assigned to be 600. If you need to receive exact string, wrap the value into quotes: `"600$"`

By default they throw Incomplete exceptions to ensure test with missing steps won't be accidentally marked as successful. We will need to implement those steps. As we are in acceptance suite we are probably using PHPBrowser (http://codeception.com/docs/modules/PhpBrowser) or WebDriver (http://codeception.com/docs/modules/WebDriver) modules. This means that we can use their methods inside Tester file, as we do with writing tests using `$I->`. You can use `amOnPage`, `click`, `see` methods inside a step definitions, so each Gherkin scenario step to be extended with basic Codeception steps. Let's show how it can be implemented in our case:

```
<?php
class AcceptanceTester extends \Codeception\Actor
{
    use _generated\AcceptanceTesterActions;

    /**
     * @Given I have product with :num1 price in my cart
     */
     public function iHaveProductWithPriceInMyCart($num1)
     {
        // haveRecord method is available in Laravel, Phalcon, Yii modules
        $productId = $this->haveRecord('Product', ['name' => 'randomProduct'.uniqid(), 'price' => $num1]);
        $this->amOnPage("/item/$productId");
        $this->click('Order');
     }

    /**
     * @When I go to checkout process
     */
     public function iGoToCheckoutProcess()
     {
        $this->amOnPage('/checkout');
     }

    /**
     * @Then I should see that total number of products is :num1
     */
     public function iShouldSeeThatTotalNumberOfProductsIs($num1)
     {
        $this->see($num1, '.products-count');
     }

    /**
     * @Then my order amount is :num1
     */
     public function myOrderAmountIs($num1)
     {
        $this->see($num1, '.total');
     }
}
```

To make testing more effective we assumed that we are using one of the ActiveRecord frameworks like Laravel, Yii, or Phalcon so we are able to dynamically create records in database with `haveRecord` method. After that we are opening browser and testing our web pages to see that after selecting those products we really see the price was calculated correctly.

We can dry-run (or run) our feature file to see that Given/When/Then are expanded with substeps:

```
 Given i have product with $600 price in my cart
   I have record 'Product',{"name":"randomProduct571fad4f88a04","price":"600"}
   I am on page "/item/1"
   I click "Order"
 And i have product with $1000 price in my cart
   I have record 'Product',{"name":"randomProduct571fad4f88b14","price":"1000"}
   I am on page "/item/2"
   I click "Order"
 When i go to checkout process
   I am on page "/checkout"
 Then i should see that total number of products is 2
   I see "2",".products-count"
 And my order amount is $1600
   I see "1600",".total"
```

This way feature file runs just the same as any other Codeception test. Substeps give us detailed information of how the scenario is being executed.

One of the criticism for testing with Gherkin was that only technical team were aware of how the test scenario is executed. This could have lead to false-positive tests. Developers could have used empty steps for scenarios (or irrelevant ones) and produced invalid tests for valid scenarios. Codeception brings communication to a next level, everyone in a team can understand what happens on a lower (technical) level. Scenario expanding to substeps shows the actual test execution process. Anyone in a team can read the output, and invest their efforts into improving the test suite.

Advanced Gherkin

Let's improve our BDD suite by using the advanced features of Gherkin language.

Background

If a group of scenarios have the same initial steps, let's that for dashboard we need always need to be logged in as administrator. We can use *Background* section to do the required preparations and not to repeat same steps across scenarios.

```
Feature: Dashboard
  In order to view current state of business
  As an owner
  I need to be able to see reports on dashboard

  Background:
    Given I am logged in as administrator
    And I open dashboard page
```

Steps in background are defined the same way as in scenarios.

Tables

Scenarios can become more descriptive when you represent repeating data as tables. Instead of writing several steps "I have product with :num1 $ price in my cart" we can have one step with multiple values in it.

```
  Given i have products in my cart
    | name         | category    | price  |
    | Harry Potter | Books       | 5      |
    | iPhone 5     | Smartphones | 1200   |
    | Nuclear Bomb | Weapons     | 100000 |
```

Tables is a recommended ways to pass arrays into test scenarios. Inside a step definition data is stored in argument passed as `\Behat\Gherkin\Node\TableNode` instance.

```
<?php
    /**
     * @Given i have products in my cart
     */
    public function iHaveProductsInCart(\Behat\Gherkin\Node\TableNode $products)
    {
        // iterate over all rows
        foreach ($node->getRows() as $index => $row) {
            if ($index === 0) { // first row to define fields
                $keys = $row;
                continue;
            }
            $this->haveRecord('Product', array_combine($keys, $row));
        }
    }
```

Examples

In case scenarios represent the same logic but differ on data, we can use *Scenario Outline* to provide different examples for the same behavior. Scenario outline is just like a basic scenario with some values replaced with placeholders, which are filled from a table. Each set of values is executed as a different test.

```
  Scenario Outline: order discount
    Given I have product with price <price>$ in my cart
    And discount for orders greater than $20 is 10 %
    When I go to checkout
    Then I should see overall price is "<total>" $

    Examples:
      | price | total |
      | 10    | 10    |
      | 20    | 20    |
      | 21    | 18.9  |
      | 30    | 27    |
      | 50    | 45    |
```

Long Strings

Text values inside a scenarios can be set inside a `"""` block:

```
  Then i see in file "codeception.yml"
"""
paths:
    tests: tests
    log: tests/_output
    data: tests/_data
    helpers: tests/_support
    envs: tests/_envs
"""
```

This string is passed as a standard PHP string parameter

```
<?php
    /**
     * @Then i see in file :filename
     */
    public function seeInFile($fileName, $fileContents)
    {
        // note: module "Asserts" is enabled in this suite
        if (!file_exists($fileName)) {
            $this->fail("File $fileName not found");
        }
        $this->assertEquals(file_get_contents($fileName), $fileContents);
    }
```

Tags

Gherkin scenarios and features can contain tags marked with `@`. Tags are equal to groups in Codeception. This way if you define a feature with `@important` tag, you can execute it inside `important` group by running:

```
codecept run -g important
```

Tag should be placed before *Scenario:* or before *Feature:* keyword. In the last case all scenarios of that feature will be added to corresponding group.

Configuration

As we mentioned earlier, steps should be defined inside context classes. By default all the steps are defined inside an Actor class, for instance, `AcceptanceTester`. However, you can include more contexts. This can be configured inside global `codeception.yml` or suite configuration file:

```
gherkin:
    contexts:
        default:
            - AcceptanceTester
            - AdditionalSteps
```

`AdditionalSteps` file should be accessible by autoloader and can be created by `Codeception\Lib\Di`. This means that practically any class can be a context. If a class receives an actor class in constructor or in `_inject` method, DI can inject it into it.

```
<?php
class AdditionalSteps
{
  protected $I;

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

  /**
   * @When I do something
   */
  function additionalActions()
  {
  }
}
```

This way PageObjects, Helpers and StepObjects can become contexts as well. But more preferable to include context classes by their tags or roles.

If you have `Step\Admin` class which defines only admin steps, it is a good idea to use it as context for all features containing with "As an admin". In this case "admin" is a role and we can configure it to use additional context.

```
gherkin:
    contexts:
        role:
            admin:
                - "Step\Admin"
```

Contexts can be attached to tags as well. This may be useful if you want to redefine steps for some scenarios. Let's say we want to bypass login steps for some scenarios loading already defined session. In this case we can create `Step\FastLogin` class with redefined step "I am logged in as".

```
gherkin:
    contexts:
        tag:
            fastlogin:
                - "Step\FastLogin"
```

Migrating From Behat

While Behat is a great tool for Behavior Driven Development, you still may prefer to use Codeception as your primary testing framework. In case you want to unify all your tests (unit/functional/acceptance), and make them be executed with one runner, Codeception is a good choice. Also Codeception provides rich set of well-maintained modules for various testing backends like Selenium Webdriver, Symfony, Laravel, etc.

If you decided to run your features with Codeception, we recommend to start with symlinking your `features` directory into one of the test suites:

```
ln -s $PWD/features tests/acceptance
```

Then you will need to implement all step definitions. Run `gherkin:snippets` to generate stubs for them. By default it is recommended to place step definitions into actor class (Tester) and use its methods for steps implementation.

Tests vs Features

It is common to think that BDD scenario is equal to test. But it's actually not. Not every test should be described as a feature. Not every test is written to test real business value. For instance, regression tests or negative scenario tests are not bringing any value to business. Business analysts don't care about scenario reproducing bug #13, or what error message is displayed when user tries to enter wrong password on login screen. Writing all the tests inside a feature files creates informational overflow.

In Codeception you can combine tests written in Gherkin format with tests written in Cept/Cest/Test formats. This way you can keep your feature files compact with minimal set of scenarios, and write regular tests to cover all cases.

Corresponding features and tests can be attached to the same group. And what is more interesting, you can make tests to depend on feature scenarios. Let's say we have `login.feature` file with "Log regular user" scenario in it. In this case you can specify that every test which requires login to pass to depend on "Log regular user" scenario:

```
@depends login:Log regular user
```

Inside `@depends` block you should use test signature. Execute your feature with `dry-run` to see signatures for all scenarios in it. By marking tests with `@depends` you ensure that this test won't be executed before the test it depends on.

Conclusions

If you like the concept of Behavior Driven Development or prefer to keep test scenarios in human readable format, Codeception allows you to write and execute scenarios in Gherkin. Feature files is just another test format inside Codeception, so it can be combined with Cept and Cest files inside the same suite. Steps definitions of your scenarios can use all the power of Codeception modules, PageObjects, and StepObjects.

- **Next Chapter: Customization > (/docs/08-Customization)**
- **Previous Chapter: < AdvancedUsage (/docs/07-AdvancedUsage)**

Looking for commercial support? Request official consulting service (http://sdclabs.com/codeception?utm_source=codeception.com&utm_medium=docs_bottom&utm_term=link&utm_campaign=reference)