```
- agpaypalTest: Start create buttons if searchstring is in content one time and co✔ agpaypalTest: Start create buttons if searchstring is in content one time and contains valuefor amount (0.00s)
- agpaypalTest: Start create buttons if searchstring is in content one time and co✔ agpaypalTest: Start create buttons if searchstring is in content one time and contains incorrect value (0.00s)
✔ agpaypalTest: Start create buttons if searchstring is in content many times (0.00s)
✔ agpaypalTest: Start create buttons if searchstring is in content no time (0.00s)
---------------------------------------------------------------------------
Time: 149 ms, Memory: 10.00MB
OK (5 tests, 5 assertions)
```

Das Plugin ist nun wieder übersichtlich und den Testcode haben wir auch angepasst. Es gibt noch viel zu tun. Welche Funktion wollen Sie als nächstes in Angriff nehmen Beginnen Sie wieder mit dem Test ….

Hier im Buch packen wir die Arbeit nicht an. Wir implementieren keine neuen Funktionen in das Plugin. Hier geht es ja nicht in erster Linie um die Programmierung von Joomla! Erweiterungen – obwohl es dazu auch viel zu schreiben gäbe. Vielmehr legen wir den Schwerpunkt auf die Verbesserung der Tests. Wie testen wir besser und welche Möglichkeiten stehen uns zur Verfügung?

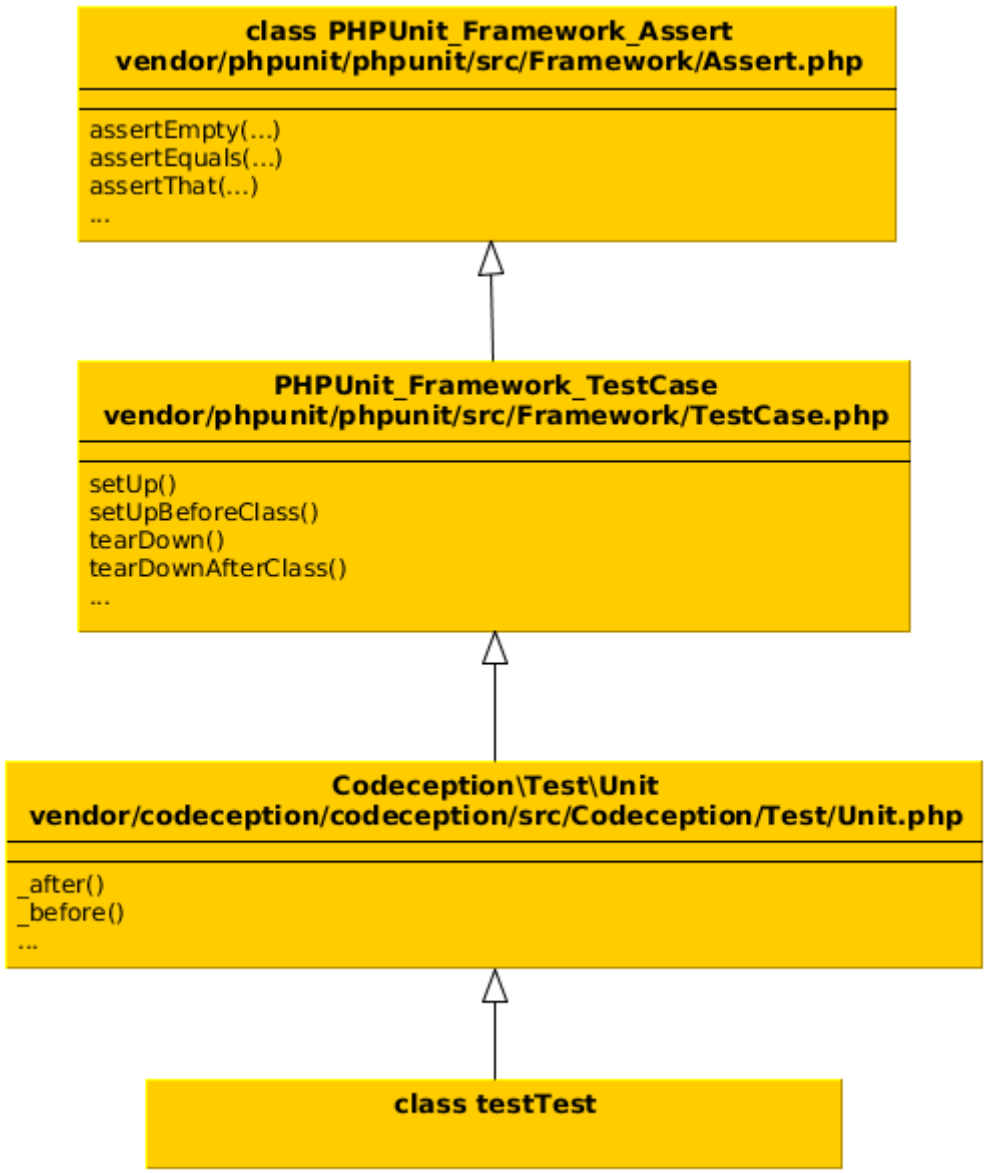
Was bietet PHPUnit Ihnen

Fassen wir noch einmal zusammen. Wir haben eine Testdatei erstellt. Diese beinhaltet die Klasse `AgpaypalTest`. Sie heißt somit genau so, wie die Klasse, die sie überprüfen soll. An das Ende des Namens haben wir lediglich das Wort `Test` angefügt. In dieser Testklasse werden alle Testfälle mit Bezug zur Klasse `AgpaypalTest` aufgenommen.

Mit dieser Benennung halten wir uns an allgemeine Regeln. Testklassen könnten theoretisch auch ganz anders heißen. Sie können sogar auf das Wort `Test` zu Beginn einer Testmethode verzichten. Wenn Sie die Annotation `@test` im Kommentar der Methode verwenden erkennt PHPUnit die Methode trotzdem als Test. Mehr zum Thema Annotationen finden Sie im übernächsten Kapitel. Bei der Benennung empfehle ich Ihnen aber, sich an die allgemeinen Regeln, die Sebastian Bergmann in seiner Dokumentation aufgestellt hat, zu halten. Es erleichtert eine Zusammenarbeit mit anderen Programmierern ungemein.

Jede Methode der Testklasse prüft eine **Eigenschaft** oder ein **Verhalten**. Eigenschaften und Verhalten müssen die Bedingungen und die Werte, die in der Spezifikation festgelegt wurden, erfüllen. Nur dann darf der Test erfolgreich

durchlaufen. Zur Prüfung bietet PHPUnit mit der Klasse PHPUnit_Framework_Assert in der Datei /joomla/vendor/phpunit/phpunit/src/Framework/Assert.php unterschiedliche Prüfmethoden. Unsere Klasse erbt diese Prüfmethoden.



Die Testmethoden der Klasse PHPUnit_Framework_Assert

PHPUnit bietet eine Vielzahl von Assert-Funktionen, die Sie in Ihren Testfällen verwenden können.

Eine Liste aller Assert-Funktionen finden Sie in der englischsprachigen PHPUnit Dokumentation. Sie können diese im Internet unter der Adresse https://phpunit.de/manual/current/en/appendixes.assertions.html abrufen.

Die wichtigsten Behauptungen - oder Assertions - sind meiner Meinung nach

- Allgemeine Funktionen
  assertEmpty(), assertEquals(), assertFalse(), assertGreaterThan(), assertGreaterThanOrEqual(), assertInternalType(), assertLessThan(), assertLessThanOrEqual(), assertNull(), assertSame(), assertTrue(), assertType()
- Spezielle Funktionen für Arrays
  assertArrayHasKey(), assertContains(), assertContainsOnly()
- Spezielle Funktionen für Klassen/Objekte
  assertAttributeContains(), assertAttributeContainsOnly(), assertAttributeEmpty(), assertAttributeEquals(), assertAttributeGreaterThan(), assertAttributeGreaterThanOrEqual(), assertAttributeInstanceOf(), assertAttributeInternalType(), assertAttributeLessThan(), assertAttributeLessThanOrEqual(), assertAttributeSame(), assertAttributeType(), assertClassHasAttribute(), assertClassHasStaticAttribute(), assertInstanceOf(), assertObjectHasAttribute()
- Spezielle Funktionen für Strings
  assertRegExp(), assertStringMatchesFormat(), assertStringMatchesFormatFile(), assertStringEndsWith(), assertStringStartsWith()
- Spezielle Funktionen für HTML/XML
  assertTag(), assertXmlFileEqualsXmlFile(), assertXmlStringEqualsXmlFile(), assertXmlStringEqualsXmlString()
- Spezielle Funktionen für Dateien
  assertFileEquals(), assertFileExists(), assertStringEqualsFile()
- Undokumentierte Funktionen
  assertEqualXMLStructure(), assertSelectCount(), assertSelectEquals(), assertSelectRegExp()
- Komplexe Assert-Funktionen
  assertThat()

Entwickeln Sie schon Software in der Programmiersprache PHP? Dann sind die Namen der Funktionen für Sie sicherlich größtenteils selbsterklärend. Sie werden sofort darauf kommen, dass zum Beispiel die Funktion assertEmpty() intern die PHP-Funktion empty() verwendet und welche Eingabe somit als - nicht mit einem Wert belegt - ausgewertet wird.

Besonders interessant ist die Funktion assertThat(). Mit ihr können Sie komplexe Assert-Funktionen erstellen. Die Funktion wertet Klassen des Typs PHPUnit_Framework_Constraint aus. Eine vollständige Tabelle der Constraints, also der Bedingungen, ist ebenfalls in der Dokumentation enthalten.

Die Rückgabewerte der Assert-Methoden sind ausschlaggebend dafür, ob ein Test erfolgreich ist oder nicht.

> **RANDBEMERKUNG:**
> Vermeiden Sie es, die Ausgabe einer Assert-Methode von mehreren Bedingungen gleichzeitig abhängig machen. Falls diese Methode einmal fehlschlägt müssen Sie erst herausfinden, welche Bedingung nicht passt. Das Herausfinden der genauen Fehlerursache ist im Nachhinein einfacher, wenn Sie separate Assert-Methoden für verschiedene Bedingungen erstellen.

Annotationen

Annotationen sind Anmerkungen oder Meta-Informationen. Sie können Annotationen im Quellcode einfügen. Hierbei müssen Sie eine spezielle Syntax beachten.

In PHP finden Sie Annotationen in PHPDoc-Kommentaren. PHPDoc-Kommentare werden verwendet, um Dateien, Klassen, Funktionen, Klassen-Eigenschaften und Methoden einheitlich zu beschreiben. Dort steht zum Beispiel, welche Parameter eine Funktion als Eingabe erwartet, welchen Rückgabewert die Funktion errechnet oder welche Variablen-Typen verwendet werden. Außerdem nutzt der PHPDocumentor die Kommentare zur Generierung einer technischen Dokumentation wenn Sie dies wünschen.

> **RANDBEMERKUNG**
> Ein PHPDoc-Kommentar in PHP muss mit den Zeichen /** beginnen und mit den Zeichen */ enden. Annotationen in anderen Programmcodebereichen werden ignoriert.

Eine Liste aller Annotationen können Sie im Internet unter der Adresse https://phpunit.de/manual/current/en/appendixes.annotations.html abrufen.

Ein praktisches Beispiel für die Verwendung einer Annotation werden im nächsten Kapitel sehen. Hier werden wir die Annotation @dataProvider verwenden.

Das erste Testbeispiel verbessern

Nun wissen Sie alles, was Sie zum Erstellen der ersten eigener Tests benötigen und probieren sicherlich schon neugierig eigene Tests aus.

> **RANDBEMERKUNG (todo Beispiel prüfen)**
> Bei Problemen sind Zusatzinformationen, die Sie über die Funktion codecept_debug() selbst bestimmen können, oft hilfreich. Diese Informationen können Sie sich in der Konsole anzuzeigen lassen, indem Sie den Parameter **--debug** an den Befehl zum Start der Tests anhängen. Zum Beispiel so:
> tests/codeception/vendor/bin/codecept run unit --debug
>
> Die Funktion
> ```
> public function testStartCreateButtons($text)
> {
>     $contenttextbefore = 'Text vorher';
>     $contenttextafter = 'Text nachher';
>     $hint = 'Zwei Paypalbuttons';
>     $this->class->startCreateButtons($contenttextbefore);
>     $this->assertEquals($contenttextafter, $contenttextbefore, $hint);
>     codecept_debug('Der Wert der Variablen war hier: ' . $hint);
> }
> ```
> würde zum Beispiel mit dem Parameter --debug folgende Ausgabe bewirken:
>
> /var/www/html/joomla$ **vendor/bin/codecept run unit --debug**
> Codeception PHP Testing Framework v2.2.9
> Powered by PHPUnit 5.7.17 by Sebastian Bergmann and contributors.
> Unit Tests (7)
> ---
> **Modules: Asserts, \Helper\Unit**
> ---
> ✔ agpaypalTest: On content prepare method runs one time (0.02s)
> - agpaypalTest: Start create buttons | #0
> **Der Wert der Variablen war hier: Zwei Paypalbuttons**
> …

```
✔ agpaypalTest: Start create buttons | #5 (0.00s)
------------------------------------------------------------
Time: 161 ms, Memory: 10.00MB
OK (7 tests, 2 assertions)
```

Wenn Sie den Parameter --debug nicht verwenden, fehlt die Ausgabe des Werts der Variablen und auch die aktiven Module werden nicht aufgelistet:

```
/var/www/html/joomla$ vendor/bin/codecept run unit
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.17 by Sebastian Bergmann and contributors.
Unit Tests (7)
------------------------------------------------------------
✔ agpaypalTest: On content prepare method runs one time (0.02s)
✔ agpaypalTest: Start create buttons | #0 (0.00s)
✔ agpaypalTest: Start create buttons | #1 (0.00s)
…
✔ agpaypalTest: Start create buttons | #5 (0.00s)
------------------------------------------------------------------------
Time: 168 ms, Memory: 10.00MB
OK (7 tests, 2 assertions)
```

Data Provider

Wir haben im Kapitel *Der erste selbst programmierte Test* den ersten Test erstellt. Diesen Test haben wir im weiteren Verlauf erweitert. Mit dem Test haben wir sichergestellt, dass die Umwandlung des Textmusters @paypalpaypal@ in einen PayPal Jetzt-kaufen-Button in einem Beitrag richtig erfolgt.

Erinnern Sie sich? Dass das Testen aller möglichen Eingabeparameter in der Realität unmöglich ist und das ein systematisches stichprobenartiges Testen die einzig praktikable Lösung ist hatten wir zu Beginn im Kapitel *Softwaretests - eine Einstellungssache?* | *Tests planen* herausgearbeitet.

Es wäre langweilig und redundant, für jeden Testfall in der Stichprobe eine eigene Testmethode zu schreiben. Sehen wir uns lieber an, wie wir eine Testmethoden automatisch mit unterschiedlichen Testdaten füttern können.

Für diesen Zweck gibt es das Konzept des Data Providers. Hierbei liefert eine Methode mehrere Datensätze, die in einer anderen Methode nacheinander verarbeitet werden.

- Der Lieferant, also der Data Provider, wird als eine öffentliche Methode in die Testklasse integriert. Diese öffentliche Methode muss ein mehrdimensionales Array mit Daten zurück geben.
- Die verarbeitende Testfunktion wird mit der Annotation @dataProvider und dem Namen des Data Providers, also dem Funktionsnamen, versehen. Den Funktionsnamen können Sie frei wählen.

Das nachfolgende Programmcodebeispiel zeigt den Data Provider provider_PatternToPaypalbutton(). Dieser gibt ein mehrdimensionales Array zurück. Jeder Datensatz des Arrays wird als eigener Testfall von der Methode testStartCreateButtons() ausgeführt.

HALTEN WIR ALSO FEST:
Jeder Wert in der zweiten Ebene des mehrdimensionalen Arrays eines Data Providers ist gleichzeitig ein Eingabeparameter für die, über die Annotation @dataProvider verknüpfte, Testmethode.

Sehen wir uns dies an einem Beispiel an:

```
<?php
namespace suites\plugins\content\agpaypal;
class agpaypalTest extends \Codeception\Test\Unit {
 protected $tester;
 protected function _before(){}
 protected function _after() {}
 /**
  * @dataProvider provider_PatternToPaypalbutton
  */
 public function testStartCreateButtons($text) {
        require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
        $subject = \JeventDispatcher::getInstance();
        $config = array(
                'name' => 'agpaypal'",
```

```
                'type' => 'content',
                'params' => new \Jregistry
        );
        $params = new \JRegistry;
        require_once JINSTALL_PATH . '/plugins/content/agpaypal/agpaypal.php';
        $class = new \PlgContentAgpaypal($subject, $config, $params);
        $contenttextbefore = $text['contenttextbefore'];
        $contenttextafter = $text['contenttextafter'];
        $class->startCreateButtons($contenttextbefore);
        $this->assertEquals($contenttextafter, $contenttextbefore);
}
public function provider_PatternToPaypalbutton() {
        return [
                [array('contenttextbefore' => '<p>Texte vor der Schaltfläche.</p>'
                . '<p>@paypal amount=16.00 paypal@</p>'
                . '<p>Text hinter der Schaltfläche.</p>'
                . '<p>Texte vor der Schaltfläche.</p>'
                . '<p>@paypal amount=15.00 paypal@</p>'
                . '<p>Text hinter der Schaltfläche.</p>',
                'contenttextafter' => '<p>Texte vor der Schaltfläche.</p>'
                . '<p><form name="_xclick" action="https://www.paypal.com/de/cgi-bin/webscr"
method="post">'
                . '<input type="hidden" name="amount" value="16.00">'
                . '<input type="image" src="http://www.paypal.com/de_DE/i/btn/x-click-but01.gif"
border="0" name="submit">'
                . '</form></p>'
                . '<p>Text hinter der Schaltfläche.</p>'
                . '<p>Texte vor der Schaltfläche.</p>'
                . '<p><form name="_xclick" action="https://www.paypal.com/de/cgi-bin/webscr"
method="post">'
                . '<input type="hidden" name="amount" value="15.00">'


                . '<input type="image" src="http://www.paypal.com/de_DE/i/btn/x-click-but01.gif"
border="0" name="submit">'
                . '</form></p>'
                . '<p>Text hinter der Schaltfläche.</p>')],
                [array('contenttextbefore' => '<p>Texte vor der Schaltfläche.</p>'
                . '<p>@paypal aouunt=15.00 paypal@</p>'
```

```
                . '<p>Text hinter der Schaltfläche.</p>',
                'contenttextafter' => '<p>Texte vor der Schaltfläche.</p>'
                . '<p><form name="_xclick" action="https://www.paypal.com/de/cgi-bin/webscr"
method="post">'
                . '<input type="hidden" name="amount" value="12.99">'
                . '<input type="image" src="http://www.paypal.com/de_DE/i/btn/x-click-but01.gif"
border="0" name="submit">'
                . '</form></p>'
                . '<p>Text hinter der Schaltfläche.</p>')],
                [array('contenttextbefore' => '<p>Texte vor der Schaltfläche.</p>'
                . '<p>@paypal amount=15.00 paypal@</p>'
                . '<p>Text hinter der Schaltfläche.</p>',
                'contenttextafter' => '<p>Texte vor der Schaltfläche.</p>'
                . '<p><form name="_xclick" action="https://www.paypal.com/de/cgi-bin/webscr"
method="post">'
                . '<input type="hidden" name="amount" value="15.00">'
                . '<input type="image" src="http://www.paypal.com/de_DE/i/btn/x-click-but01.gif"
border="0" name="submit">'
                . '</form></p>'
                . '<p>Text hinter der Schaltfläche.</p>')],
                [array('contenttextbefore' => '<p>Texte vor der Schaltfläche.</p>'
                . '<p>@paypalpaypal@</p>'
                . '<p>Text hinter der Schaltfläche.</p>',
                'contenttextafter' => '<p>Texte vor der Schaltfläche.</p>'
                . '<p><form name="_xclick" action="https://www.paypal.com/de/cgi-bin/webscr"
method="post">'
                . '<input type="hidden" name="amount" value="12.99">'
                . '<input type="image" src="http://www.paypal.com/de_DE/i/btn/x-click-but01.gif"
border="0" name="submit">'
                . '</form></p>'
                . '<p>Text hinter der Schaltfläche.</p>')],
        [array('contenttextbefore' => "jsdkl",
                'contenttextafter' => "jsdkl")],
                [array('contenttextbefore' => "", 'contenttextafter' => "")]
        ];
}
}
```

Was haben wir genau geändert? Zunächst habe wir den Data Provider, beziehungsweise die Funktion provider_PatternToPaypalbutton(), eingefügt. Diese Funktion gibt einen mehrdimensionalen Array zurück. Als nächstes haben wir die Annotation @dataProvider provider_PatternToPaypalbutton in den Kommentar über unserer Testfunktion testOnContentPrepare() eingesetzt und dieser Funktion den Eingabeparameter $text hinzugefügt. Durch die Annotation @dataProvider provider_PatternToPaypalbutton wird der Eingabeparameter $text mit dem Rückgabewert der Methode provider_PatternToPaypalbutton() gleichgesetzt. Dies bewirkt, dass die Testfunktion testOnContentPrepare() für jeden Wert in der zweiten Ebene des Arrays aufgerufen wird. Im konkreten Fall wir die Methode sechsmal ausgeführt.

Starten wir nun den Test erneut, sieht die Ausgabe folgendermaßen aus:

```
/var/www/html/joomla$ vendor/bin/codecept run unit
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.13 by Sebastian Bergmann and contributors.
Unit Tests (6) ----------------------------------------------------
✔ agpaypalTest: Start create buttons | #0 (0.00s)
✔ agpaypalTest: Start create buttons | #1 (0.00s)
✔ agpaypalTest: Start create buttons | #2 (0.00s)
✔ agpaypalTest: Start create buttons | #3 (0.00s)
✔ agpaypalTest: Start create buttons | #4 (0.00s)
✔ agpaypalTest: Start create buttons | #5 (0.00s)
------------------------------------------------------------------
Time: 140 ms, Memory: 10.00MB
OK (6 tests, 6 assertions)
```

Die Verwendung von Data Providern ermöglichte es uns also, eine Testmethode mehrmals mit unterschiedlichen Daten aufzurufen. So können wir mit Data Providern komplexe Testfälle flexibel auf unterschiedliche Situationen anpassen. Der Programmcode einer Funktion wird so mehrmals wiederverwendet. Dies ist ein großer Vorteil. Wenn alles glatt läuft, reicht uns diese Ausgabe. Wenn aber ein Testfall fehlschlägt, hätten wir gerne genauere Informationen. Bei der Verwendung von Data Providern ist die Ausgabe beim Testlauf nun auch einheitlich. Hier können wir leicht Abhilfe schaffen, indem wir für den Fehlerfall einen Hinweistext mitgeben. Sehen Sie sich das nachfolgende Beispiel an. Hier habe ich einen Hinweistext integriert und diesen als dritten Parameter in der Funktion assertEquals() mitgegeben.

```
 …
/**
 * @dataProvider provider_PatternToPaypalbutton
 */
public function testStartCreateButtons($text){
        require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
        $subject = \JEventDispatcher::getInstance();
        $config = array(
                'name' => 'agpaypal'",
                'type' => 'content',
                'params' => new \JRegistry
        );
        $params = new \JRegistry;
        require_once JINSTALL_PATH . '/plugins/content/agpaypal/agpaypal.php';
        $class = new \PlgContentAgpaypal($subject, $config, $params);
        $contenttextbefore = $text['contenttextbefore'];
        $contenttextafter = $text['contenttextafter'];
        $hint = $text['hint'];
        $class->startCreateButtons($contenttextbefore);
        $this->assertEquals($contenttextafter, $contenttextbefore, $hint);
}
public function provider_PatternToPaypalbutton(){
...
[array('contenttextbefore' => "",
'contenttextafter' => "",
'hint' => 'Beitrag enthält keinen Text')]
…
```

Wenn nun ein Test fehlschlägt, wird der Hinweistext in der Konsole mit ausgegeben. Der nachfolgende Text zeigt die Ausgabe der Konsole, wenn der Test mit dem leeren Beitrag fehlschlägt. Anhand des Hinweistextes müssen Sie nun nicht mehr lange suchen. Sie wissen genau, welcher Wert des Data Providers den Fehler ausgelöst hat und können gezielt nach einer Lösung suchen.

```
/var/www/html/joomla$ vendor/bin/codecept run unit
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.13 by Sebastian Bergmann and contributors.
```

```
Unit Tests (6) ----------------------------------------------------------------
✔ agpaypalTest: Start create buttons | #0 (0.00s)
✔ agpaypalTest: Start create buttons | #1 (0.00s)
✔ agpaypalTest: Start create buttons | #2 (0.00s)
✔ agpaypalTest: Start create buttons | #3 (0.00s)
✔ agpaypalTest: Start create buttons | #4 (0.00s)
✖ agpaypalTest: Start create buttons | #5 (0.00s)
------------------------------------------------------------------------------
Time: 153 ms, Memory: 10.00MB
There was 1 failure:
---------
1) agpaypalTest: Start create buttons | #5
 Test tests/unit/suites/plugins/content/agpaypal/agpaypalTest.php:testStartCreateButtons
Beitrag enthält keinen Text
Failed asserting that two strings are equal.
- Expected | + Actual
@@ @@
...
```

Data Provider simulieren mögliche Eingaben in unsere Software. Im Programm selbst gibt es zusätzlich bereits Objekte und Daten, die wir während eines Tests sicher in einem bekannten Zustand haben möchten. Zum Beispiel Konfigurationsparameter oder Datenbankinhalte. Im nächsten Kapitel geht es um Testduplikate und wie wir deren Hilfe eine bekannte feste Testumgebung aufbauen können.

Fixtures in Codeception

Testduplikate oder künstliche Objekte setzten wir ein, wenn wir in einer gut bekannten und festen Umgebung testen möchten. Nur so ist ein Test wirklich wiederholbar. Anderenfalls könnten Situationen eintreten, die nicht reproduzierbar sind. Codeception bietet mit der Klasse `Codeception\Util\Fixtures` eine einfache Möglichkeit Daten in einem globalen Array zu speichern und so in einer Testdatei an mehreren unterschiedlichen Stellen zu verwenden.

Fixtures unterstützen Sie dabei, vor einem Test bestimmte wohldefinierte Daten zur Verfügung zu stellen. Getestet wird also in einer kontrollierten Umgebung. Dies macht Tests einfach und übersichtlich. Konkret bedeutet das, dass Sie die Methode `testonContentPrepare()` testen können, ohne sich darauf verlassen zu müssen, das andere Werte vorher durch andere Klassen während der Programmausführung richtig gesetzt

wurden. Beispielweise die Variablen $config. Das Setzen dieses Wertes ist nicht Bestandteil dieses Tests. Wir testen hier ausschließlich die Methode onContentPrepare() der Klasse PlgContentAgpaypal!

Was ist eine Fixture genau? Eine Fixture ist eine Variable, die mit einer bestimmten Belegung gespeichert wird. Mit dieser Belegung kann sie mehrmals im Testablauf abgerufen werden. In der Regel wird diese Fixture in einer Methode gespeichert, die vor jedem Test in einer Testdatei automatisch abläuft. Zu den automatisch ablaufenden Methoden können Sie im nächsten Abschnitt mehr lesen. Sie können Fixturen auch in einer boostrap-Datei ganz zu Beginn des Testablaufs einmal setzten.

Als nächstes ändern wir unser Beispiel so ab, dass wir für Konfiguration eine Fixture nutzen.

```
<?php
namespace suites\plugins\content\agpaypal;
use \Codeception\Util\Fixtures;
class agpaypalTest extends \Codeception\Test\Unit {
 protected $tester;
 protected function _before(){}
 protected function _after() {}
 /**
  * @dataProvider provider_PatternToPaypalbutton
  */
 public function testStartCreateButtons($text)
 {
        require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
        $subject = \JeventDispatcher::getInstance();
        Fixtures::add('config', [
                'name' => 'agpaypal',
                'type' => 'content',
                'params' => new \JRegistry
        ]);
        $params = new \JRegistry;
        require_once JINSTALL_PATH . '/plugins/content/agpaypal/agpaypal.php';
        $class = new \PlgContentAgpaypal($subject, Fixtures::get('config'), $params);
        $contenttextbefore = $text['contenttextbefore'];
        $contenttextafter = $text['contenttextafter'];
```

```
        $hint = $text['hint'];
        $class->startCreateButtons($contenttextbefore);
        $this->assertEquals($contenttextafter, $contenttextbefore, $hint);
}
public function provider_PatternToPaypalbutton(){
...
```

Was haben wir genau geändert? Zunächst einmal mussten wir mit der Anweisung `use \Codeception\Util\Fixtures;` sicherstellen, dass wir die Klasse `Fixtures` verwenden können. Dann haben wir das Objekt `$config` als Fixture gespeichert und später über die Methode `Fixtures::get()` geladen.

Der Vorteil dieser Änderungen ist so noch nicht offensichtlich. Momentan benötigen wir die Variabel `$config` nur an einer Stelle. Wenn wir aber im weiteren Verlauf immer mal wieder eine Konfiguration mit dieser Belegung benötigen wird der Vorteil klar.

> **RANDBEMERKUNG**
> Sie können eine Fixture überschreiben, indem sie diese erneut speichern. Haben Sie beispielsweise die Anweisung
> **Fixtures::add('benutzer', ['name' => 'Peter']);**
> in einer Testdatei und in der nächsten Testdatei setzten Sie
> **Fixtures::add('benutzer', ['name' => 'Paul']);,**
> dann erhalten Sie über
> **Fixtures::get('benutzer');**
> den Benutzer **Paul**.
> Allgemein gilt, dass alle Testfälle einer Testklasse von den gemeinsamen Fixturen Gebrauch machen sollten. Hat eine Testmethode für keine Fixture Verwendung, dann sollten Sie prüfen, ob die Testmethode nicht besser in eine andere Testklasse passen würde. Oft ist dies nämlich ein Indiz dafür. Es kann durchaus vorkommen, dass zu einer Klasse mehrere korrespondierende Testfallklassen existieren. Jede von diesen besitzt ihre individuellen Fixtures.

Sie sehen schon. Die Verwendung von von Fixturen sollte geplant werden. Schon allein deshalb ist es sinnvoll, Fixturen nur an bestimmten stellen mit Werten zu belegen. Hierzu bieten sich Methoden an, die PHPUnit und Codeception Ihnen zur

Planung der Tests zur Verfügung stellen. Zum Beispiel die Methoden, die ich Ihnen im nächsten Abschnitt näher bringen will.

Vor dem Test den Testkontext herstellen und hinterher aufräumen

Sie wissen es schon: Für Software-Tests ist es wichtig, dass jede Testfunktion unter kontrollierten und immer wieder gleichen Bedingungen ausgeführt wird. So darf eine Testfunktion den Ablauf einer anderen nicht stören oder ungewollt beeinflussen. Wenn Sie ein Objekt erstellen, das von verschiedenen Testfunktionen benutzt werden soll, so muss dieses vor jedem Test wieder in den Ausgangszustand versetzt werden. Diese Vorbereitungen sind mitunter lästig. Auch hinterher das Aufräumen mag niemand gerne tun. Schön, dass Sie von PHPUnit und Codeception Methoden an die Hand bekommt, die Sie bei diesen lästigen und oft routinemäßigen Arbeiten unterstützen. Vor und nach jedem Test werden bestimmt Methoden automatisch in einer bestimmten Reihenfolge ausgeführt.

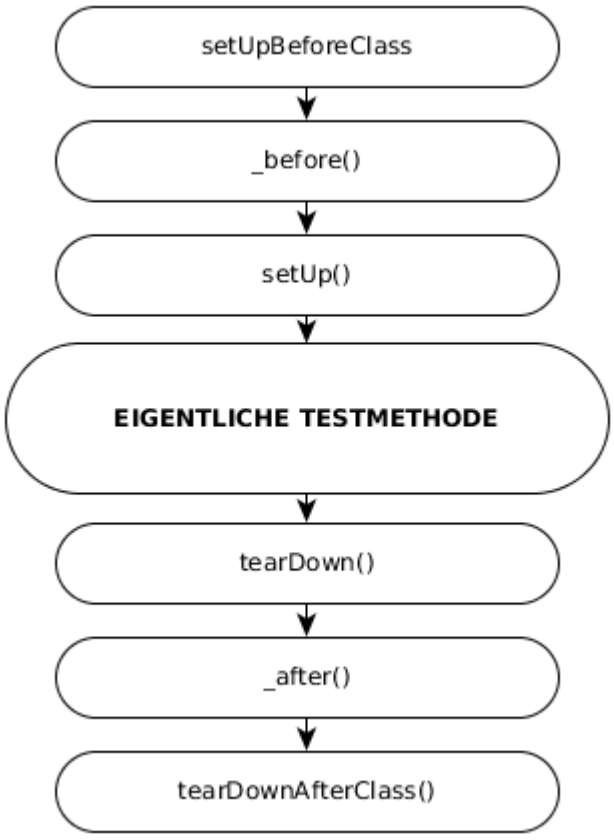


Abbildung 18: Vor und nach jedem Test werden die abgebildeten Methoden automatisch in der angezeigten Reihenfolge ausgeführt. 993.png

Für unsere Testmethode testStartCreateButtons() heißt das konkret, dass folgende Methoden nacheinander aufgerufen werden:

1. tests/unit/suites/plugins/content/agpaypal/agpaypalTest.**setUpBeforeClass()**
2. tests/unit/suites/plugins/content/agpaypal/agpaypalTest.**_before()**
3. tests/unit/suites/plugins/content/agpaypal/agpaypalTest.**setUp()**
4. tests/unit/suites/plugins/content/agpaypal/agpaypalTest.**testStartCreateButtons()**
5. tests/unit/suites/plugins/content/agpaypal/agpaypalTest.**tearDown()**
6. tests/unit/suites/plugins/content/agpaypal/agpaypalTest.**_after()**
7. tests/unit/suites/plugins/content/agpaypal/agpaypalTest.**tearDownAfterClass()**

Das Design von Tests verlangt fast ebenso viel Gründlichkeit wie das Design der Anwendung. Um PHPUnit und Codeception möglichst effektive einzusetzen, müssen Sie wissen, wie Sie effektive Testfälle schreiben. Unter anderem sollten Sie Fixturen und Testduplikate nicht im Konstruktor einer Testklasse initialisieren. Wenn Sie objektorientiert programmieren, ist dies vielleicht Ihr erster Gedanke. Sinnvoller ist es aber, die dafür vorgesehenen Methoden zu verwenden. Dies sind die Methoden, die ich Ihnen eben aufgelistet habe. Diese Methoden werden vor jedem einzelnen Test ausgeführt und bilden so für jeden Test eine kontrollierte Umgebung.

Mit den Methoden _before() und _after() erweitert Codeception mit der Klasse Codeception/Test/Unit das PHPUnit Framework. Alle anderen oben aufgeführten Methoden sind Standard PHPUnit Methoden. Auf diese Weise können Sie über die Klasse Codeception/Test/Unit alle tollen Funktionen der verschiedenen Codeception Module und Hilfsklassen, zusätzlich zu allen PHPUnit Funktionen verwenden!

(Todo Ist klar warum ich meine das die Klasse Codeception/Test/Unit erweitert werden sollte und warum ich helper in _before und _after verwenden kann?)

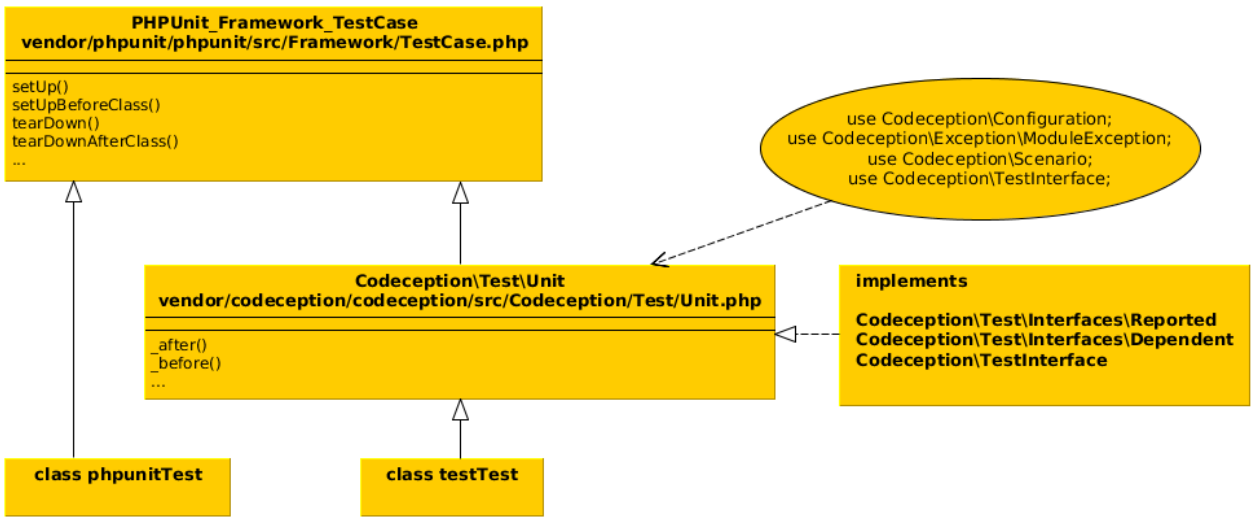


Abbildung 19: 959c.png

Im Falle von PHPUnit Tests werden Sie sicherlich am häufigste die Methoden setUp() und tearDown() verwenden. Möchten Sie Codeception Hilfsklassen oder Module nutzen sind die Methoden _setup() und _after() die richtigen für Sie. Später in diesem Kapitel werden Sie ein Fixture über die Methode _setup() in Ihren Tests zur Verfügung stellen. In diesem Kapitel ist mir aber zunächst folgendes wichtig: Mit Programmcode in diesen Methoden sollten Sie sicherstellen, dass eine wohldefinierte und somit kontrollierte Umgebung für jede Testmethode erstellt wird und nach dem Test alles wieder in den Ursprungszustand zurück gesetzt wird. Falls Sie Objekte erstellt haben oder etwas in einer Datenbank gespeichert haben, dann sollten Sie die Objekte und die Datenbankeinträge nachher wieder löschen.

Es gibt wenige Gründe eine Testumgebung für alle Tests in einer Klasse vorzubereiten, die während des Testdurchlaufs nicht für jeden Test wieder in den Ursprungszustand zurück gesetzt wird. Ein Grund könnte die Einrichtung der Datenbankverbindung sein, wenn Sie diese für alle Tests in der Testklasse benötigen. Natürlich könnten Sie diese in der setUp() Methode für jeden Test separat erstellen und nach jedem Test wieder trennen. Performanter ist es aber ganz sicher, diese Verbindung nur einmal vor der Ausführung aller Tests aufzubauen. Während der Tests können Sie dann immer wieder auf diese Verbindung zuzugreifen, um erst nach dem Abarbeiten aller Testfälle in der Testklasse die Verbindung wieder zu trennen. setUpBeforeClass() und tearDownAfterClass() sind die richtigen Methoden für diese Aufgabe. Sie bilden die äußersten Aufrufe. Ausgeführt werden Sie bevor die erste Testmethode einer Klasse gestartet wird - beziehungsweise nach dem Abarbeiten der letzten Testmethode.

Im Moment haben wir nur eine Methode in unserem Test. Das wird sich aber im Verlauf des Buches ändern. Es werden weitere Testmethoden hinzukommen. Wir möchten ja auch sicherstellen, dass alles richtig läuft, wenn kein Suchtext in der Form @paypalpaypal@ eingegeben wurde oder wenn Parameter innerhalb des Suchtextes mitgegeben wurden. Bereiten wir uns, da wir uns gerade die Methoden die vor und nach jedem Test ablaufen ansehe, darauf vor, mit deren Hilfe unser Testumgebung einzurichten.

```
<?php
namespace suites\plugins\content\agpaypal;
use \Codeception\Util\Fixtures;
class agpaypalTest extends \Codeception\Test\Unit{
 protected $tester;
 protected $class;
 protected function _before(){
        require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
        $subject = \JeventDispatcher::getInstance();
        Fixtures::add('config', [
                'name' => 'agpaypal"',
                'type' => 'content',
                'params' => new \JRegistry
        ]);
        $params = new \JRegistry;
        require_once JINSTALL_PATH . '/plugins/content/agpaypal/agpaypal.php';
        $this->class = new \PlgContentAgpaypal($subject, Fixtures::get('config'), $params);
}
protected function _after() {}
/**
 * @dataProvider provider_PatternToPaypalbutton
 */
public function testStartCreateButtons($text){
        $contenttextbefore = $text['contenttextbefore'];
        $contenttextafter = $text['contenttextafter'];
        $hint = $text['hint'];
        $this->class->startCreateButtons($contenttextbefore);
        $this->assertEquals($contenttextafter, $contenttextbefore, $hint);
}
```

```
public function provider_PatternToPaypalbutton(){
...
```

Was haben wir genau umgestellt? Das Object $class werden wir höchstwahrscheinlich in jedem Test unserer Testklasse verwenden. Deshalb haben wir die Erstellung dieses Objektes in die _before() Methode verschoben. So können wir, auch wenn wir mehrere Testfälle oder Testmethoden nutzen, immer auf dieses Objekt zurückgreifen.

Der Vorteil ist, dass wir nun in jeder Testmethode, die wir der Klasse hinzufügen, das Objekt $class und auch die Fixture in einem wohldefinierten bekannten Zustand verwenden können.

Kurzgefasst

Hauptthema des 4. Kapitels werden Unit Tests sein. Nach einer kurzen Einführung in Unit Tests im Allgemeinen beschreibe ich wie Codeception die Erstellung von Unit Tests unterstützt.

Testduplikate

> Der Test ist der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen.
> [Ernst Denert]

(todo Einleitung)

Mit Unittests testen Sie eine möglichst kleine Funktionseinheit. Wichtig ist, dass diese Tests unabhängig von anderen Einheiten erfolgen. Dies haben wir in den vorhergehenden Kapiteln bereits heraus gearbeitet. Zahlreiche Klassen lassen sich aber gar nicht so ohne Weiteres einzeln und unabhängig testen, weil ihre Objekte in der Anwendung eng mit anderen Objekten zusammen arbeiten. Dafür gibt es eine Lösung, nämlich das Erstellen von Testduplikaten. Wenn Sie dieses Kapitel durchgearbeitet haben, sind Sie in der Lage, Abhängigkeiten in Ihrer Anwendung mithilfe von Stub-Objekten und Mock-Objekten zu lösen und Sie kennen den Unterschied zwischen Stubs und Mocks.

Mit der Codeception Klasse Fixtures haben Sie im vorausgehenden Kapitel bereits ein Testduplikat kennengelernt. Ein Testduplikat ist im Grunde genommen ein Hilfsobjekt. Dieses Hilfsobjekt wird nicht selbst getestet. Es wird im Test lediglich verwendet. Fixtures stehen beispielsweise für triviale unechte Implementierungen. Wie wir gesehen haben werden in der Regel vordefinierte Werte zurückgegeben. Die Variable $config, die wir im vorhergehenden Kapitel über die Klasse Fixture realisiert haben, ist ein optionales Array in dem Konfigurationseinstellungen gespeichert sind. Fixturen sollen komplexe Abläufe oder Berechnungen in der Anwendung ersetzten, indem Sie das Ergebnisobjekt künstlich zur Verfügung stellen.

> Für Testduplikate gibt es unterschiedliche Bezeichnungen. Martin Fowler unterscheidet Dummy, Fake, Stub und Mock Objekte. Und dann haben Sie im vorausgehenden Kapitel die Codeception Klasse Fixtures kennengelernt. Ich könnte noch weitere Begriffe aufführen. Für was welcher Begriff steht ist - wenn überhaupt - nur sehr schwammig definiert.
> Im Grunde genommen geht es bei allen Testduplikaten darum, komplexe Abhängigkeiten mithilfe von neu erzeugten Objekten aufzulösen.

In diesem Kapitel geht es nun um Stubs und Mocks. Diese sollen beim Testen, im Gegensatz zu Fixtures in Codeception, reale Objekte ersetzten.

Externe Abhängigkeiten auflösen - Das erste Stub Objekt

Fangen wir mit einem leicht verständlichen Beispiel an und erstellen ein einfaches Stub Objekt.

WICHTIG
final, private und static Methoden können nicht mit PHPUnit Stub Objekten genutzt werden. PHPUnit unterstützt diese Methoden nicht.

Ich hatte schon geschrieben das Stub Objekte, zumindest nach der allgemeinen Definition, einem in der Anwendung tatsächlich vorkommenden Objekte entsprechen. In unserem Beispiel arbeiten wir mit Objekte, die es auch in der Applikation Joomla! gibt. Ich meine die Objekte $row und $params. $row instantiiert die PHP eigene Klasse stdClass und $params instantiert die Klasse \JRegistry.

> **RANDBEMERKUNG**
> JRegistry finden Sie in Joomla in der Datei joomla/libraries/vendor/joomla/registry/src/Registry.php. Diese wird über die Datei /joomla/libraries/classmap.php in Joomla eingebunden.

Wir könnten diese Objekte als Stubs erstellen und verwenden. Für unser Beispiel bringt dies zunächst keinen Vorteil. Es ist aber ein gutes Beispiel um Ihnen zu zeigen, wie Sie Stub Objekte erstellen und die Rückgabewerte beeinflussen können. Deshalb erstellen wir nun diese beiden Stub Objekte.

```
<?php
namespace suites\plugins\content\agpaypal;
use \Codeception\Util\Fixtures;
class agpaypalTest extends \Codeception\Test\Unit{
 protected $tester;
 protected $class;
 protected $row;
 protected $params;
 protected function _before(){
        require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
        $subject = \JeventDispatcher::getInstance();
        Fixtures::add('config', [
                'name' => 'agpaypal',
                'type' => 'content',
                'params' => new \JRegistry
        ]);
        $params = new \JRegistry;
        require_once JINSTALL_PATH . '/plugins/content/agpaypal/agpaypal.php';
        $this->class = new \PlgContentAgpaypal($subject, Fixtures::get('config'), $params);
        $this->row = $this->getMockBuilder(\stdClass::class)->getMock();
        $this->params = $this->getMockBuilder(\JRegistry::class)->getMock();
}
 protected function _after() {}
 public function testOnContentPrepare(){
        $returnValue = $this->class->onContentPrepare('', $this->row, $this->params);
        $this->assertTrue($returnValue);
```

```
}
...
```

Was haben wir ergänzt? Wir haben für die Methode onContentPrepare() erneut eine Testmethode erstellt. Dieses Mal testen wir diese Methode aber mit Objekten, die wir vorher mit der Methode getMockBuilder() erstellt haben, gefüllt.

RANDBEMERKUNG

Wollen Sie sich die Methode getMockBuilder() genauer ansehen. Sie finden diese in der Klasse PHPUnit_Framework_TestCase – also in der Datei /joomla/vendor/phpunit/phpunit/src/Framework/TestCase.php. Diese erstellt ein Objekt vom Typ PHPUnit_Framework_MockObject_MockBuilder und gibt dieses zurück.

```
....
public function getMockBuilder($className) {
    return new PHPUnit_Framework_MockObject_MockBuilder($this, $className);
}
....
```

Die Klasse PHPUnit_Framework_MockObject_MockBuilder selbst ist nicht im PHPUnit Paket sondern im Paket PHPUnit-Mock-Objects. Dieses Paket haben Sie bei der Installation von Codeception mit installiert und in Ihrem Projekt bekannt gemacht. Dies hat Composer für Sie übernommen. Sehen Sie in der Datei composer.lock nach, wenn Sie es mir nicht glauben.

Das Objekt $params verfügt über die Methode get(). Die Methode get() gibt die aktuellen Werte für einen Parameter zurück, sofern der Parameter mit einem Wert belegt ist. Der Aufruf $params->get('aktive') gibt also den Wert true zurück, falls der Parameter aktive mit true belegt ist. Mit dem folgendem Code bewirken Sie, dass das Objekt $params immer true ausgibt, wenn es mit dem Parameter aktive aufgerufen wird.

```
        $this->params->expects($this->any())
        ->method('get')
        ->with('aktive')
        ->willReturn(true);
```

Die Annahme $this->assertTrue($this->params->get('aktive')); würde also einen Test bestehen. Finden Sie dies kompliziert? Vielleicht hilft Ihnen meine Art den Programmcodeabschnitt zu lesen:

> Immer **($this->any())** wenn die Methode get() **(method('get'))** mit dem Eingabeparameter aktive **(with(‚aktive‘))** aufgerufen wird, wird der Wert true **(willReturn(true))** zurückgegeben.

In unserem Testbeispiel haben wir lediglich Mock Objekte erstellt. Wir haben keinen fixen Rückgabewert für die Objekte $row und $params gesetzt. Uns reichte es aus, dass wir die Objekte zur Verfügung haben. Wir mussten diese nicht bewusst manipulieren. Ich gebe zu, in unserem Falle ist dies ein etwas missbrauchtes, konstruiertes Beispiel. Das Hauptziel dieses Beispiels war es, Ihnen die Erstellung eines Stub Objektes zu zeigen. Und nun haben wir sogar zwei Stub Objekte erstellt. Sie haben nun die grundlegenden Kenntnisse, um Ihre eigenen Tests mit Stub Objekten zu bestücken.

(ToDo Link https://phpunit.de/manual/current/en/test-doubles.html und Beispiel mit Tipps disableOriginalConstructor(), disableOriginalClone(), disableArgumentCloning(), disallowMockingUnknownTypes())

Ein Mock Objekte

Bevor wir im nächsten Abschnitt den Unterschied zwischen Mock Objekten und Stub Objekten klären, erstellen wir hier nun ein Mock Objekt. Mit einem Testfall für die Klasse PlgAgpaypal können wir ein einfaches Beispiel konstruieren. Sehen wir uns zunächst den für unser Beispiel relevanten Abschnitt der Klasse noch einmal genauer an. In der Methode onContentPrepare() sollte die Methode startCreateButtons() auf jeden Fall aufgerufen werden. (todo)

```
...
class PlgContentAgpaypal extends Jplugin{
 public function onContentPrepare($context, &$row, $params, $page = 0) {
        if (is_object($row))
        {
                $this->startCreateButtons($row->text);
        } else {
                $this->startCreateButtons($row);
```

```
        }
        return true;
}
…
```

Um sicherzugehen, dass die Methode startCreateButtons() ausgeführt wird können wir folgende Testmethode erstellen.

```
<?php
namespace suites\plugins\content\agpaypal;
use \Codeception\Util\Fixtures;
class agpaypalTest extends \Codeception\Test\Unit {
…
public function testOnContentPrepareMethodRunsOneTime() {
        $myplugin = $this->getMockBuilder(\PlgContentAgpaypal::class)
        ->disableOriginalConstructor()
        ->setMethods(['startCreateButtons'])
        ->getMock();
        $myplugin->expects($this->once())->method('startCreateButtons');
        $myplugin->onContentPrepare('', $this->row, $this->params);
}
...
```

Führen Sie diesen Test nun aus. Er wird erfolgreich sein!

```
/var/www/html/joomla$ vendor/bin/codecept run unit
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.13 by Sebastian Bergmann and contributors.
Unit Tests (1) ------------------------------------------------------------
✔ agpaypalTest: On content prepare method runs one time (0.01s)
------------------------------------------------------------------------
Time: 101 ms, Memory: 10.00MB
OK (1 test, 1 assertion)
```

Wie Sie sehen, haben wir hier keine Assert-Methode verwendet. Die Zeile `$myplugin->expects($this->once())->method('startCreateButtons');` sorgt dafür, dass der Test nur dann erfolgreich ist, wenn die Methode `startCreateButtons()` ausgeführt wurde.

Probieren Sie es aus. Kommentieren Sie die Zeilen `$this->startCreateButtons($row);` im Plugin Programmcode einfach aus.

```
<?php
defined('_JEXEC') or die;
class PlgContentAgpaypal extends Jplugin{
 public function onContentPrepare($context, &$row, $params, $page = 0){
        if (is_object($row))
        {
                // $this->startCreateButtons($row->text);
        }else{
                $this->startCreateButtons($row);
        }
        return true;
 }
 public function startCreateButtons(&$text){
...
```

Das Auskommentieren der Zeile `$this->startCreateButtons($row);` hat bewirkt, dass die Methode `startCreateButtons()` nun nicht mehr ausgeführt wird, wenn die Variable `$row` ein Objekt ist. In unserem Falle wird der Beitragstext in einem Objekt `$row` übergeben und kann mittels `$row->text` ausgelesen werden. Bei einem erneuter Testlauf mit dem Befehl `vendor/bin/codecept run unit` wird die Konsole Ihnen folgende Meldung anzeigen:

```
/var/www/html/joomla$ vendor/bin/codecept run unit
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.13 by Sebastian Bergmann and contributors.
Unit Tests (1) ------------------------------------------------------------------
✖ agpaypalTest: On content prepare method runs one time (0.02s)
---------------------------------------------------------------------------------
Time: 115 ms, Memory: 10.00MB
There was 1 failure:
---------
```

```
1) agpaypalTest: On content prepare method runs one time
 Test
tests/unit/suites/plugins/content/agpaypal/agpaypalTest.php:testOnContentPrepareMethodRunsOneTi
me
Expectation failed for method name is equal to <string:startCreateButtons> when invoked 1 time(s).
Method was expected to be called 1 times, actually called 0 times.
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Der Test schlug fehl, weil PHPUnit erwartet hat, dass die Methode einmal aufgerufen wird. Sie wurde aber keinmal aufgerufen.

Wie unterscheiden sich Mock Objekte von Stub Objekten

Nun haben Sie unterschiedliche Testduplikate praktisch kennengelernt. Das Fixturen, schwammig definiert, keine realen Arbeitsobjekte darstellen und sie sich so von Mocks und Stubs unterscheiden, habe ich schon abgegrenzt. Den Unterschied zwischen Mock und Stub Objekten haben wir aber noch nicht geklärt.

In einem Satz kann ich es so beschreiben: Stubs prüfen den **Status** einer Anwendung und Mocks das **Verhalten**.

Jeder Testfall prüft eine Annahme bezüglich dem Status oder dem Verhalten eines Programms. Hierfür können mehrere Stub Objekte notwendig sein. In der Regel benötigen wir aber immer nur ein Mock Objekt. Falls Sie das Verhalten von mehreren Objekten in Ihrem Test überprüfen möchten, ist der Test in der Regel zu weit gefasst. Sie testen nicht mehr nur eine Annahme und sollten überlegen, ob Sie den Test nicht besser in mehrere Test unterteilen.

Testlebenszyklus eines Stub Objektes

1. Bereiten Sie das zu prüfende Objekt und seine Stub Objekte vor. In der Regel sollte dies in der _before() Methode erfolgen.
2. Führen Sie die zu testende Funktion aus.
3. Prüfen und vergleichen Sie den Zustand nach dem Ausführen der Funktion.
4. Stellen Sie den Ausgangszustand der Anwendung wieder her. Dies erfolgt in der Regel in er Methode _after().

Testlebenszyklus eines Mock Objektes

1. Bereiten Sie das zu prüfende Objekt und seine Stub Objekte vor. In der Regel sollte dies in der _before() Methode erfolgen.
2. Bereiten Sie das Mock Objekt, das im Zusammenspiel mit dem Testobjekt ein bestimmtes Verhalten zeigen soll, vor.
3. Führen Sie die zu testende Funktion aus.
4. Stellen Sie sicher, dass das erwartete Verhalten eingetreten ist.
5. Prüfen und vergleichen Sie den Zustand nach dem Ausführen der Funktion.
6. Stellen Sie den Ausgangszustand der Anwendung wieder her. Dies erfolgt in der Regel in er Methode _after().

Sowohl Mocks als auch Stubs geben eine Antwort auf die Frage: **Was** ist das Ergebnis.
Mocks sind zusätzlich interessiert daran **wie** das Ergebnis erreicht wurde!

BDD Spezifikationen

Bevor wir nun den Unittest Teil abschließen, möchte ich Ihnen gerne noch zwei nette Funktionen in Codeception zeigen.

Codeception unterstützt Sie mit den Erweiterungspakten Specify und Verify dabei, Beschreibungen des Verhaltens der Software in Tests zu nutzen. Wenn Sie Ihre Software verhaltensgetrieben entwickeln, sollten Sie jederzeit auf eine Beschreibung der umzusetzenden Software zugreifen können. Dies Merkmale der verhaltensgetriebenen Softwareentwicklung hatten ich im Kapitel *Softwaretests - eine Einstellungssache? | Behavior-Driven-Development (BDD)* beschrieben. Gleichzeitig unterstützt Specify Sie dabei Ihre Tests modular und flexibel aufzubauen.

Bei den kurzen Beispielen in diesem Buch ist die Notwendigkeit für einen modularen Aufbau und die Verwendung von Beschreibungen nicht offensichtlich. Sehen sie sich den Ordner mit den aktuell in Joomla! vorhandenen Unittests einmal an. Obwohl die Testabdeckung noch nicht optimal ist, sind diese schon sehr zahlreich - und leider noch überwiegend in einer veralteten PHPUnit Version geschrieben.

Was meinen Sie was passiert, wenn eine beim Integrieren einer neuen Funktion die notwendige Programmcodeänderung Fehler bei der Ausführung eines Test auslöst? Vielleicht sogar in einem Test, der vor mehreren Jahren von jemandem geschrieben

wurde, der heute nicht mehr aktiv im Projekt mitarbeitet? Ich denke es ist klar was ich meine. Entweder gibt es jemanden, der sofort versteht warum der Test fehlschlägt. Dieser Jemand kann dann sicher auch relativ schnell die Fehlerursache beheben. Oder niemand versteht Test, der nun fehlschlägt. Wahrscheinlich wird der fehlerhafte Test nun erst einmal ignoriert oder vielleicht sogar gelöscht. Zumindest dann, wenn die neue Funktion als wichtig angesehen wird und man eher am Test, als an der neuen Funktion zweifelt.

Damit andere Projektbeteiligte vorhandene Tests schnell durchschauen ist es wichtig, dass es klare Regeln gibt, an die sich auch jeder hält. Jedes Team Mitglied sollte genau wissen, wie es einen Test schreibt. Dazu müssen diese Regeln präzise und einfach sein. Es ist nicht förderlich für ein Projekt, wenn ein Entwickler dabei ist, der komplizierten Testcode beisteuert. Auch dann nicht, wenn seine Beiträge fachlich wirklich gut sind. Wenn andere im Team diese guten Ansätze nicht verstehen wird der gute Code am Ende Wegwerfcode sein und im Team wird es vorher viel Frust geben.

Wirklich gut ist Code, egal ob Programmcode oder Testcode, meiner meiner Meinung nach erst, wenn andere diesen schnell nachvollziehen können – und das am besten ohne viele Details nachfragen zu müssen.

Die Erweiterungen Specify und Verify unterstützen Sie darin **Tests richtig zu schreiben** und nicht nur die **Test richtig auszuführen**.

Codeception Werkzeuge Specify und Verify

Installation

Die Pakete Specify und Verify werden nicht automatisch mit Codeception mit installiert. Mit Composer ist es aber einfach die beiden Erweiterungen zu Ihrem Projekt hinzuzufügen.

Ergänzen Sie dazu die Datei Composer.json die Sie im Kapitel *Codeception – ein Überblick | Composer | Die Dateien Composer.json und Composer.lock* in Ihrem Stammverzeichnis angelegt hatten. In meinem Beispiel ist das Stammverzeichnis das Verzeichnis /var/www/html/joomla. Fügen Sie die beiden Zeilen "codeception/specify": "*", und "codeception/verify": "*" in diese Datei ein.

```
{
  "require": {
    "codeception/codeception": "*",
        "codeception/specify": "*",
```

```
        "codeception/verify": "*"
    }
}
```

Wenn Sie nun den Befehl composer update im Stammverzeichnis ausführen, werden die beiden Pakete - inklusive aller in einer Abhängigkeitsbeziehung stehenden Pakete - in Ihrem Projekt installiert und auch untereinander bekannt gegeben. Außerdem werden aufgrund des Befehls composer update auch alle bereits im Projekt vorhandenen Pakete aktualisiert. Falls Ihnen der Unterschied zwischen composer update und composer install nicht mehr klar, ist können Sie diesen im Kapitel *Codeception – ein Überblick* | *Composer* | *Die Dateien Composer.json und Composer.lock* nachlesen.

```
/var/www/html/joomla$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Removing guzzlehttp/psr7 (1.3.1)
  - Installing guzzlehttp/psr7 (1.4.1)
    Loading from cache
  - Removing guzzlehttp/guzzle (6.2.2)
…
  - Installing codeception/specify (0.4.6)
    Downloading: 100%
  - Installing codeception/verify (0.3.3)
    Downloading: 100%
Writing lock file
Generating autoload files
```

Specify

Ich habe im nachfolgenden Programmcodebeispiel unseren Beispieltest umgebaut. Nun wird die Funktion Specify verwendet. Die Tests sind nun tatsächlich in einem besser lesbaren BDD-Stil geschrieben.

Achten Sie darauf, dass Sie das Paket Codeception\Specify innerhalb der Klassendefinition mit dem use-Operator importieren, denn Specify muss als Trait innerhalb einer Klassendefinition hinzugefügt werden.

> **RANDBEMERKUNG**
> In PHP wird der use-Operator unterschiedlich eingesetzt:
> - Als Alias für eine andere Klasse. In diesem Fall muss der use-Operator außerhalb der Klassendefinition deklariert werden.
> - Um einen Trait zu einer Klasse hinzuzufügen In diesem Fall muss der use-Operator innerhalb der Klassendefinition deklariert werden.
> - In anonymer Funktionsdefinition, um Variablen innerhalb der Funktion zu übergeben.

Wenn Sie Data Provider mit Specify nutzen möchten ist es wichtig das Sie wissen, dass Sie diese nicht beliebig nennen dürfen. Der Data Provider muss example heißen:
['examples' => $this->provider_PatternToPaypalbutton()]

```
<?php
namespace suites\plugins\content\agpaypal;
use \Codeception\Util\Fixtures;
class agpaypalTest extends \Codeception\Test\Unit{
 use \Codeception\Specify;
 protected $tester;
 protected $class;
 protected $row;
 protected $params;
 protected function _before(){
        require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
        $subject = \JeventDispatcher::getInstance();
        Fixtures::add('config', [
                'name' => 'agpaypal',
                'type' => 'content',
                'params' => new \JRegistry
        ]);
        $params = new \JRegistry;
        require_once JINSTALL_PATH . '/plugins/content/agpaypal/agpaypal.php';
        $this->class = new \PlgContentAgpaypal($subject, Fixtures::get('config'), $params);
        $this->row = $this->getMockBuilder(\stdClass::class)->getMock();
        $this->params = $this->getMockBuilder(\JRegistry::class)->getMock();
        $this->params->expects($this->any())
        ->method('get')
```

```
            ->with('aktive')
            ->willReturn(true);
    }
    protected function _after() {}
    public function testOnContentPrepareMethodRunsOneTime()
    {
            $this->specify("Die Method startCreateButtons wird ausgeführt wenn der Beitragstext
als Eigenschaft eines Objektes vorkommt.", function() {
            $myplugin = $this->getMockBuilder(\PlgContentAgpaypal::class)
                    ->disableOriginalConstructor()
                    ->setMethods(['startCreateButtons'])
                    ->getMock();
                    $myplugin->expects($this->once())->method('startCreateButtons');
                    $myplugin->onContentPrepare('', $this->row, $this->params);
            });
            $this->specify("Die Method startCreateButtons wird ausgeführt wenn der Beitragstext
als reiner Text vorkommt.", function() {
                    $myplugin = $this->getMockBuilder(\PlgContentAgpaypal::class)
                    ->disableOriginalConstructor()
                    ->setMethods(['startCreateButtons'])
                    ->getMock();
                    $myplugin->expects($this->once())->method('startCreateButtons');
                    $text = 'Text des Beitrages';
                    $myplugin->onContentPrepare('', $text, $this->params);
            });
    }
    /**
     * @dataProvider provider_PatternToPaypalbutton
     */
    public function testStartCreateButtons($text){
            $this->specify("Wandelt den Text @paypalpaypal@ in eine PayPalschaltfläche um,
wenn er einmal im Beitragstext vorhanden ist.", function($text) {
            $contenttextbefore = $text['contenttextbefore'];
            $contenttextafter = $text['contenttextafter'];
            $hint = $text['hint'];
            $this->class->startCreateButtons($contenttextbefore);
            $this->assertEquals($contenttextafter, $contenttextbefore, $hint);
            }, ['examples' => $this->provider_PatternToPaypalbutton()]);
```

```
}
public function provider_PatternToPaypalbutton() {
…
```

Möchten Sie das Paket Specify verwenden? Ausführlichere Informationen finden Sie in der Dokumentation auf Github.

Verify

Die Funktion Verify überzeugt mich nicht vollends. Das liegt aber vielleicht daran, dass ich mit den Assert-Methoden von PHPUnit vertrauter bin. Ich möchte Ihnen Verify aber nicht vorenthalten. Es ist sicher richtig, dass, wenn Sie `$this->assertEquals` mit `verify()` ersetzen, der Testprogrammcode lesbarer und kürzer wird. Im nachfolgenden Programmcodebeispiel sehen Sie, wie Sie Verify in unseren Beispieltest integrieren können. Ich habe die vorhandene Assert-Methode auskommentiert und an der Stelle die Specify-Methode eingesetzt.

```
<?php
namespace suites\plugins\content\agpaypal;
use \Codeception\Util\Fixtures;
use \Codeception\Verify;
class agpaypalTest extends \Codeception\Test\Unit {
...
/**
 * @dataProvider provider_PatternToPaypalbutton
 */
public function testStartCreateButtons($text){
        $this->specify("Wandelt den Text @paypalpaypal@ in eine PayPalschaltfläche um, wenn er einmal im Beitragstext vorhanden ist.", function($text) {
        $contenttextbefore = $text['contenttextbefore'];
        $contenttextafter = $text['contenttextafter'];
        $hint = $text['hint'];
        $this->class->startCreateButtons($contenttextbefore);
        // $this->assertEquals($contenttextafter, $contenttextbefore, $hint);
        verify($contenttextafter)->equals($contenttextbefore);
        }, ['examples' => $this->provider_PatternToPaypalbutton()]);
}
...
```

Lesen Sie bitte die ausführliche Dokumentation auf Github, wenn Ihnen verify() gefällt und Sie die das Paket einsetzen möchten.

Kurzgefasst

… haben wir Mocks und Stubs und … Todo dies in jedes Kapitel und auch über all ein vorwort. Todo es gibt auch noch Stub in codeception

Im 5. Kapitel geht es um Schnittstellen mit denen das zu testende System interagiert und wie diese simuliert werden können. Thema sind Stubs (also Code, der stellvertretend für den realen Code steht) und Mocks (simulierte Objekte).

Funktionstest

(todo Einleitung)

In diesem Kapitel geht es um Funktionstest. Im vorhergehenden Kapitel haben wir eine selbst erstellte Joomla! Erweiterung als einzelne Einheit getestet. Das die Erweiterung die Schnittstelle zu Joomla! korrekt nutzt, konnten wir so mit Tests belegen. Verarbeiten aber alle anderen Joomla! Units die Daten, die unsere Erweiterung liefert, richtig weiter? Die bisherigen Tests waren unabhängig von anderen Programmcodeteilen. In diesem Kapitel werden wir das Zusammenspiel von mehreren Einheiten genauer unter die Lupe nehmen.

Todo REST

Tauchen Sie mit mir hier nun in das Thema Funktionstests ein. Erstellen Sie mit mir einige einfache Tests bevor wir uns dann die REST-Schnittstelle ansehen.

Wie Sie wissen enthält unsere Website bisher nur einen einzigen Beitrag und der PayPal Jetzt-kaufen-Button wird sofort auf der Startseite angezeigt. (todo eventuell bild).

Ein erstes Beispiel

Den ersten Funktionstest generieren

Im Kapitel Codeception – ein Überblick hatte ich die wichtigsten codecept Befehle erklärt. Eigentlich erklärt sich der Befehl vendor/bin/codecept generate:cept functional /suites/plugins/content/agpaypal/agpaypal, mit dem Sie ein Boilerplate für einen Funktionstest erstellen können, aber selbst. Mit diesem Befehl wird die Datei agpaypalCept.php im Verzeichnis /var/www/html/joomla/tests/functional//suites/plugins/content/agpaypal/ angelegt.