

Bucheinband

Viele Programmierer haben ein ungutes Gefühl bestehenden Programmcode zu erweitern oder zu verändern. Nach einer Änderung ist es oft so, dass irgendwo im Programm ein Problem auftritt. Wer dann einmal in die testgetriebene Softwareentwicklung hinein schnuppert, macht oft die gute Erfahrung, dass er mit dieser Arbeitsweise nach getaner Arbeit entspannt nach Hause gehen kann.

Codeception ist ein beliebtes Tool, das Sie bei der Erstellung von Softwaretests unterstützt – mit seiner Vielzahl an Funktionen allerdings nicht selbsterklärend ist. Dieses Buch hilft Ihnen dabei, sich mühelos und schnell in Codeception zurechtzufinden. Von der Installation mittels Composer bis hin zu fertigen Unittests, Funktionstests und Akzeptanztests wird Ihnen das Vorgehen in leicht verständlichen Anleitungen erklärt.

Fragen

Softwaretests – eine Einstellungssache?

1. Warum sollten Sie Ihre Software testen, wenn Sie diese in guter Qualität anbieten möchten?
 1. Weil es ohne Tests unmöglich ist, qualitativ gute Software zu erstellen. (richtig)
 2. Ursachen fehlerhafter Software sind menschliche Fehlleistungen. Ich muss Software während der Entwicklung nur dann testen, wenn ich Fehler mache. Ich sollte mich eher darauf konzentrieren, die Software fehlerfrei zu erstellen. (falsch)
 3. Ich sollte meine Software nur dann testen, wenn es mir wichtig ist, dass Fehler frühzeitig erkannt werden. Kostengünstiger wäre es aber, auf das Erstellen von Testcode zu verzichten. Bei der Programmausführung tauchen Fehler sowieso automatisch auf und können auch dann noch behoben werden. (falsch)
 4. Weil der Standard ISO 29119 dies fordert. (falsch)
2. Das Magische Dreieck im Projektmanagement beschreibt den Zusammenhang zwischen den Kosten, der benötigten Zeit und der leistbaren Qualität. Dieser Zusammenhang ist zwingend!
 1. Der Zusammenhang aus Zeit, Kosten und Qualität kann langfristig überwunden werden. (richtig)
 2. Ein höherer Kostenaufwand hat positive Auswirkungen auf die Qualität und/oder den Fertigstellungstermin – also die Zeit. Das Magische Dreieck soll anschaulich darstellen, dass immer nur zwei der drei Ziele erreichbar sind – beziehungsweise, dass bei dem Fokus auf zwei Ziele ein drittes vernachlässigt werden muss. Dieser Zusammenhang kann nicht überwunden werden! (falsch)
 3. Dieser Zusammenhang ist zwingend, weil Kosten, benötigte Zeit und leistbaren Qualität untereinander konkurrierende Ziele sind. (falsch)
 4. Ja, der Zusammenhang ist zwingend. Wenn in einem Softwareprojekt Softwaretests erstellt werden, ist dies ein Mehraufwand und das Erstellen der Tests hat negativen Einfluss auf die benötigte Zeit. Dies bedeutet, dass

mit gleichem Kostenaufwand nur eine geringere Qualität erzeugt werden kann. (falsch)

Praxisteil: Die Testumgebung einrichten

1. Wie unterscheiden sich Funktionstests oder Integrationstests von Akzeptanztests?
 1. Ein Akzeptanztest testet nicht nur, ob die Software selbst korrekt funktioniert, sondern auch das Verhalten der Software unter realen Bedingungen. (richtig)
 2. Ein Akzeptanztest wird in der Softwareentwicklung angewendet, um eine kleine Einheit eines Programms zu testen. Ein Integrationstest bezeichnet in der Softwareentwicklung eine aufeinander abgestimmte Reihe von Einzeltests. Diese Einzeltests dienen dazu, verschiedene voneinander abhängige Einheiten eines Systems im Zusammenspiel miteinander zu testen. (falsch)
 3. Integrationstests zählen, anders als die Akzeptanztests, zu den White-Box-Tests. White-Box-Tests sind Software-Tests, die in Kenntnis über die innere Funktionsweise des zu testenden Systems entwickelt werden. (falsch)
 4. Im Grunde genommen bezeichnen beide Testvarianten das Gleiche. Der Begriff Akzeptanztest – oder auch Abnahmetest – wird in der Endphase eines Softwareprojektes eher verwendet. (falsch)
2. Die heute übliche Vorgehensweise beim Erstellen von Tests ist *von unten nach oben* – beziehungsweise von *innen nach außen*. Je weiter sich die verhaltensgetriebene Softwareentwicklung durchsetzt, desto mehr kommt diese Strategie in Wanken. Warum vertreten immer mehr Entwickler die Sichtweise, dass von *oben nach unten* – beziehungsweise von *außen nach innen* – getestet werden sollte.
 1. Nur bei einer Top-down Strategie kann eine falsche Auslegung der Benutzeranforderungen frühzeitig erkannt und korrigiert werden. (richtig)
 2. Nur bei einer Top-down Strategie wird das Ziel in einzelne Komponenten zerlegt. Komplexe Sachverhalte können so schneller von Menschen verstanden und dargestellt werden. (falsch)

3. Nur bei einer Top-down Strategie verfügt man sehr schnell über fertige Softwareteile. (falsch)
4. Ein Nachteil der Bottom-up Strategie ist, dass sehr viel Zeit für die Erstellung von Testduplikaten verwendet werden muss. Noch nicht integrierte Komponenten müssen durch Platzhalter ersetzt werden. Es gibt zu Beginn noch keinen echten Programmcode. Deshalb vertreten immer mehr Entwickler die Sichtweise, dass von *oben nach unten* – beziehungsweise von *außen nach innen* – getestet werden sollte. (falsch)

Codeception – ein Überblick

1. Wer oder was ist Composer und wofür wird Composer gebraucht?
 1. Ein PHP-Projekt ist oft abhängig von anderen Projekten. Diese Abhängigkeiten mussten lange Zeit vom Entwickler manuell verwaltet werden. Mit Composer ist dies zum Glück Vergangenheit, den Composer übernimmt diese lästige und fehleranfällige Arbeit. (richtig)
 2. Composer bündelt bewährte existierende Testprogramme und vereinheitlicht die Anwendung. (falsch)
 3. Composer ist ein Werkzeug zur Messung der Codeabdeckung mit Tests. (falsch)
 4. Composer unterstützt Sie bei der Automatisierung Ihrer Tests und bei der kontinuierlichen Integration Ihrer Softwarebausteine. (falsch)
2. Warum ist Codeception mehr als nur ein weiteres Testwerkzeug?
 1. Weil Codeception eine einheitliche Art Tests zu schreiben bietet. Dabei unterstützt es unterschiedliche Testtypen – nämlich Unittests, Integrationstest/Funktionstests und Akzeptanztests. (richtig)
 2. Weil Sie mit Codeception auch Designelemente testen können. (falsch)
 3. Weil Codeception automatisch Hinweise darauf gibt, ob die Spezifikation richtig umgesetzt wurde. (falsch)
 4. Weil mit Codeception das Testen aller möglichen Eingabeparameter in ein Programm möglich ist. (falsch)

Unittests

1. Was testet ein Unittest?

1. Ein Unittest testet kleinstmögliche Einheiten innerhalb einer Software. (richtig)
2. Ein Unittest prüft und bewertet Software auf Erfüllung der für ihren Einsatz definierten Anforderungen und misst ihre Qualität. (falsch)
3. Ein Unittest testet die Zusammenarbeit voneinander abhängiger Komponenten. (falsch)
4. Der Unittest ist die Teststufe, bei der das gesamte System gegen die gesamten Anforderungen getestet wird. (falsch)

2. Welche Aufgabe hat ein Data Provider innerhalb eines Unittests?

1. Ein Data Provider simuliert mögliche Eingaben in das zu testende Programm während eines Testdurchlaufs. (richtig)
2. Wichtig ist, dass Softwaretests für eine Einheit, unabhängig von anderen Einheiten erfolgen. Zahlreiche Klassen lassen sich aber nicht so ohne Weiteres einzeln und unabhängig testen. Die Aufgabe von Data Provider ist es, Softwaretests einzelner Einheiten unabhängig voneinander ablaufen zu lassen. (falsch)
3. Als Data Provider bezeichnet man in der Softwareentwicklung Objekte, die als Platzhalter für echte Objekte innerhalb von Unittests verwendet werden. (falsch)
4. Ein Data Provider bezeichnet in der Softwareentwicklung einen Programmcode, der anstelle eines anderen Programmcodes steht. (falsch)

Testduplikate

1. Welche Aufgabe haben Testduplikate?

1. Wichtig ist, dass Softwaretests für eine Einheit unabhängig von anderen Einheiten erfolgen. Zahlreiche Klassen lassen sich aber nicht ohne Weiteres einzeln und unabhängig testen. Dafür gibt es eine Lösung, nämlich das Erstellen von Testduplikaten. (richtig)

2. Ein Testduplikat simuliert mögliche Eingaben in das zu testende Programm während eines Testdurchlaufs. (falsch)
 3. Ein Testduplikat liefert, anstelle von vorher zum Testfall passend festgelegten Werte, echte Daten zurück. (falsch)
 4. Ein Testduplikat wird eingesetzt, um die Performance eines Testdurchlaufs positiv zu beeinflussen. (falsch)
2. Wie unterscheiden sich Stubs von Mocks?
1. Stub-Objekte prüfen den Status einer Anwendung und Mock-Objekte das Verhalten der Anwendung. (richtig)
 2. Anders als ein Stub-Objekt liefert ein Mock-Objekt echte Daten. (falsch)
 3. Mocks und Stubs sind genau wie die Begriffe Dummy, Fake und Fixtures eine Bezeichnung für Testduplikate und unterscheiden sich nicht. (falsch)
 4. Mocks werden bei der testgetriebenen Softwareentwicklung eingesetzt und Stubs bei der verhaltensgetriebenen Softwareentwicklung. (falsch)

Funktionstest

1. Wie unterstützt Codeception Sie beim Erstellen von Aktionen in Funktionstests und Akzeptanztests?
 1. Codeception bietet vordefinierte Methoden, die Sie nutzen können. Damit Sie alle Aktionen, die Codeception für ein Modul bietet nutzen können, müssen Sie das entsprechende Modul in der Konfigurationsdatei hinzufügen und danach den Befehl `vendor/bin/codecept build` ausführen. (richtig)
 2. Codeception bietet Ihnen eine benutzerfreundliche und übersichtliche Bedieneroberfläche. (falsch)
 3. Codeception überprüft, ob die ausgewählten Aktionen logisch zusammenpassen. (falsch)
 4. Mit Codeception können Sie einzelne Aktionen in einer Vorschau ansehen. (falsch)
2. Welche Konstrukte sind in Codeception dafür vorgesehen, Programmcode wiederzuverwenden?

1. Akteure, Step-Objekte und Page-Objekte. Den Akteur FunctionalTester finden Sie in Codeception in der Klasse FunctionalTester im Verzeichnis joomla/tests/_support. Step-Objekte kommen ins Spiel, wenn Sie mit einem Akteur verschiedene inhaltlich zusammenhängende Bereiche testen. Page-Objekte unterstützen die Wiederverwendung von Elementen im HTML-Dokumentes. (richtig)
2. Sie können mithilfe von Testduplikaten Programmcode wiederverwenden. (falsch)
3. Die Akteure bieten die einzige Möglichkeit, um Programmcode wiederzuverwenden. (falsch)
4. In Codeception gibt es keine speziellen Konstrukte zur Wiederverwendung von Programmcode. Sie können Codewiederholungen vermeiden, indem Sie die Konzepte der Objektorientierten Programmierung nutzen. (falsch)

Akzeptanztests

1. Wie können Sie Akzeptanztests von Unittests abgrenzen?
 1. Mit Akzeptanztests überprüfen Sie nicht nur die Korrektheit des Programms. Sie überprüfen zusätzlich, ob die Spezifikationen, die ganz zu Beginn des Projektes aufgestellt wurden, erfüllt ist: Tut die Software tatsächlich das, was sie tun soll? Sie prüfen also nicht nur, ob die Software richtig erstellt wurde, sondern auch, ob die richtige Software erstellt wurde! (richtig)
 2. Unittests kommen bei einer testgetriebenen Entwicklung zum Einsatz. (falsch)
 3. Akzeptanztests kommen nur bei einer verhaltensgetriebenen Entwicklung zum Einsatz. (falsch)
 4. Akzeptanztests bauen auf Unittests auf. (falsch)
2. Können Sie Tests mithilfe von Codeception in einem wirklichen Browser mit grafischer Benutzeroberfläche automatisch ablaufen lassen?
 1. Ja, mithilfe von Selenium Standalone Server und WebDriver können Sie Akzeptanztests auf einem wirklichen Browser ausführen. (richtig)
 2. Nein, um ein Programm in einem Browser mit grafischer Benutzeroberfläche ablaufen zu lassen, muss manuell eingegriffen werden. Automatisch kann

ein Programm nur in einem „Headless Browser“, also einem Browser ohne grafischer Benutzeroberfläche, ablaufen. (falsch)

3. Ja. Allerdings ist dies nur im Browser Google Chrome möglich. (falsch)
4. Man kann Tests mithilfe von Codeception nur in einem wirklichen Browser mit grafischer Benutzeroberfläche automatisch ablaufen lassen, wenn es sich beim Browser um eine Open-Source-Software handelt und man Zugriff auf den Quellcode des Browsers hat. (falsch)

Analyse

1. Eine hohe Abdeckung des Programmcodes mit Tests, ist mit einer hohen Qualität der Software gleichzusetzen. Trifft dieser Satz zu?
 1. Eine hohe Codeabdeckung kann auch mit Tests erreicht werden, die nichts Wichtiges überprüfen. Deshalb hat die Codeabdeckung für die Qualität der Tests eine nur eingeschränkte Aussagekraft. (richtig)
 2. Eine hohe Abdeckung des Programmcodes mit Tests ist mit einer hohen Qualität der Software gleichzusetzen. Dieser Satz ist voll und ganz richtig. (falsch)
 3. Programme, die die Abdeckung des Programmcodes mit Tests berechnen, filtern Tests, die nichts Wichtiges überprüfen, automatisch heraus. Deshalb ist eine hohe Abdeckung des Programmcodes mit Tests mit einer hohen Qualität der Software gleichzusetzen. (falsch)
 4. Eine hohe Abdeckung des Programmcodes mit Tests verlangsamt den Programmablauf und hat deshalb negative Auswirkungen auf die Qualität der Software. (falsch)
2. Das Einhalten von Programmiercodestandards bringt viele Vorteile beim Arbeiten im Team – dafür ist die Einhaltung der Standards jedoch auch sehr zeitaufwendig! Ist dieser Satz zutreffend?
 1. Es gibt Softwaretools, mit deren Hilfe man den eigenen Programmcode automatisch auf Programmcodestandards testen und bis auf wenige Ausnahmen automatisch korrigieren lassen kann. (richtig)
 2. Programmiercodestandards sind so unterschiedlich, dass ein automatisches Korrigieren nicht möglich ist. (falsch)

3. Programmiercodestandards sind ein Relikt aus vergangenen Zeiten und es gibt heute keinen Grund mehr für ihren Einsatz. (falsch)
4. Codeception wandelt Programmcode automatisch so um, dass bestimmte Programmiercodestandards eingehalten werden. (falsch)

Automatisierung – Gestatten mein Name ist Jenkins

1. Was genau ist eine kontinuierliche Integration von Software?
 1. Bei der kontinuierlichen Integration werden einzelne Bestandteile der Software permanent integriert. Die Software wird in kleinen Zyklen immer wieder erstellt und getestet. Integrationsprobleme oder fehlerhafte Tests finden sich frühzeitig und nicht erst Tage oder Wochen später. (richtig)
 2. Bei einer kontinuierlichen Integration werden neue Softwareteile sofort in den eigentlichen Programmcode integriert. So arbeiten alle Entwickler immer mit der aktuellsten Programmversion. (falsch)
 3. Bei einer kontinuierlichen Integration werden neue Programmcodebestandteile nach zeitlich genau festgelegten Regeln zum Programm hinzugefügt. (falsch)
 4. Kontinuierliche Integration ist ein anderes Wort für Versionskontrolle. (falsch)
2. Welche Vorteile bietet die kontinuierliche Integration von neuen Softwarebausteinen?
 1. Bei einer kontinuierlichen Integration ist die Fehlerbehebung wesentlich einfacher, weil die Fehler zeitnah zur Programmierung auftauchen und in der Regel nur ein kleiner Programmteil betroffen ist. (richtig)
 2. Da alle Entwickler immer mit dem gleichen Programmcode arbeiten, gibt es weniger Programmcodekonflikte, die bei einer späteren Zusammenführung zu lösen wären. (falsch)
 3. Bei einer kontinuierlichen Integration ist immer klar, welcher Programmcode in welcher Version hinzugefügt wurde. (falsch)
 4. Bei einer kontinuierlichen Integration weiß jeder Entwickler, was er wann zu tun hat, da es einen klaren Fahrplan gibt. (falsch)

Guten Tag,

Bevor ich zuerst auf [Joomla!](#) und dann später auf [Codeception](#) gestoßen bin, konnte ich mir nicht vorstellen, dass die Hindernisse, die mir beim Testen oft im Wege standen, tatsächlich etwas zur Seite gerückt wurden.

Ich habe sehr viel Zeit mit dem Testen von Software – und früher noch mehr mit den Problemen die aufgrund von fehlenden Tests entstanden sind – verbracht!

Nun bin ich davon überzeugt, dass Tests, die

- möglichst zeitnah zur Programmierung,
- automatisch,
- häufig – idealerweise nach jeder Programmänderung

durchgeführt werden mehr einbringen als sie kosten. Und darüber hinaus: Testen kann sogar Spaß machen.

Ich musste mir auf meinem Lernweg alle Informationen an verschiedenen Stellen zusammen suchen und selbst eine Menge nicht immer schöner Erfahrungen sammeln. Deshalb habe ich mich entschieden das Buch zu schreiben, welches ich auf meinem Weg gerne zur Verfügung gehabt hätte. Dabei habe ich nicht den Anspruch, tief in jedes Detail hineinzugehen. Das Buch bietet einen Überblick zu den verschiedenen Testmethoden und nach dem Durcharbeiten haben Sie alle notwendigen Grundlagen um selbst tief einzusteigen.

RANDBEMERKUNG:

Es lohnt sich Test-Methoden zu erlernen! Testmethoden sind nachhaltig, denn diese können nicht nur mit jeder Programmiersprache genutzt werden, sie sind anwendbar auf so gut wie jedes Menschenwerk. Sie sollten fast alles beizeiten einmal testen.

Testmethoden sind unabhängig von bestimmten Softwarewerkzeugen.

Das, was Sie in diesem Buch lesen, ist im Gegensatz zu Programmiertechniken oder Programmiersprachen, die oft Modeerscheinungen sind, zeitlos.

Welche Themen behandelt dieses Buch?

Das Kapitel *Softwaretests – eine Einstellungssache?* behandelt die Frage, warum man Zeit in Softwaretests investieren sollte. Dabei erläutere ich auch die wichtigsten Testmethoden und Testkonzepte.

Im Kapitel *Praxisteil: Die Testumgebung einrichten* richte ich mit Ihnen die Entwicklungsumgebung ein, mit der wir im weiteren Verlauf des Buches arbeiten werden. Danach installieren wir das Content Management System Joomla! und entwickeln eine einfache Erweiterung. Diese Erweiterung soll die Grundlage für die späteren Beispieltests sein. Am Ende des Kapitels sehen wir uns die einzelnen Teile des Softwaresystems an und überlegen welche unterschiedlichen Herangehensweisen in Frage kommen – also, welche Teststrategie wir anwenden möchten.

Es gibt sehr viele Werkzeuge, die Sie beim Testen von Software unterstützen. In diesem Buch zeige ich Ihnen die Software Codeception. Codeception ist kein weiteres Testtool. Vielmehr bündelt Codeception bewährte existierende Testprogramme und vereinheitliche die Anwendung. Im Kapitel *Codeception – ein Überblick* zeige ich Ihnen, wie Sie Codeception auf Ihrem Ubuntu Rechner installieren und sehe mir mit Ihnen die wichtigsten Dateien und Verzeichnisse an.

Wie Sie Unittests mit Codeception schreiben und Ihre Software dabei testgetrieben entwickeln erfahren Sie im Kapitel *Unittests*. Falls Sie Unittests noch nicht kennen ist dies kein Problem. Ich gehe in diesem Kapitel auch auf die Grundfunktionen von PHPUnit ein.

Tests sollten unabhängig ablaufen. Sie sollten keine besonderen Einstellungen oder Daten voraussetzen und auch nicht auf einem anderen Test aufbauen. Deshalb müssen Sie vor einem Test oft eine künstliche Umgebung aufbauen. Wie Sie dies tun erfahren Sie im Kapitel *Testduplikate*.

Mit Funktionstests stellen Sie sicher, dass Ihr System als Ganzes korrekt zusammenarbeitet. Codeception unterstützt Sie beim Erstellen von Funktionstests. Wie Sie Codeception für den Aufbau von Funktionstests nutzen können erfahren Sie im Kapitel *Funktionstest*.

Ihr Programm arbeitet technisch korrekt. Aber tut es auch genau das, was der Kunde erwartet? Diese Frage beantworten Akzeptanztests. Wie Sie Akzeptanztests mit Codeception schreiben und in Ihrem Browser automatisch ausführen erfahren Sie im Kapitel *Akzeptanztests*.

Wenn Sie Tests für Ihre Software schreiben, möchten Sie sicherlich auch immer wissen, welche Programmteile gut mit Tests abgedeckt sind und wo Tests fehlen. Lesen Sie im Kapitel *Analyse*, wie Sie sich einen Überblick über den Stand Ihrer Tests mithilfe von aussagekräftigen Reports verschaffen können.

Damit Ihre Tests nicht in Vergessenheit geraten, zeige ich Ihnen am Ende im Kapitel *Automatisierung – Gestatten mein Name ist Jenkins* ein Werkzeug, mit dem Sie die Ausführung Ihrer Tests automatisieren können.

Was Sie zur Bearbeitung dieses Buchs benötigen

Welche Ausstattung brauchen Sie? Sie müssen nicht sehr viele Voraussetzungen erfüllen, um die Inhalte dieses Buches bearbeiten zu können. Natürlich müssen Sie über einen heute üblichen Computer verfügen. Auf diesem sollte eine [Entwicklungsumgebung](#) und ein lokaler [Webserver](#) installiert oder installierbar sein. Weitere Informationen und Hilfen zur Einrichtung Ihres Arbeitsplatzes finden Sie im Kapitel *Praxisteil: Die Testumgebung einrichten*.

Was sollten Sie persönlich für Kenntnisse mitbringen? Sie sollten grundlegende PHP Programmier Techniken beherrschen. Idealerweise haben Sie bereits eine kleine oder mittlere Webapplikation programmiert. Auf jeden Fall sollten Sie wissen, wo Sie PHP Dateien auf Ihrem Entwicklungsrechner ablegen und wie Sie diese in Ihrem Internetbrowser aufrufen.

Das Allerwichtigste ist aber: Sie sollten Spaß daran haben neue Dinge auszuprobieren.

Wer sollte dieses Buch lesen

Jeder, der der Meinung ist, dass das Testen von Software reine Zeitverschwendung ist, sollte einen Blick in dieses Buch werfen. Insbesondere möchte ich diejenigen Entwickler einladen dieses Buch zu lesen, die schon immer Tests für ihre Software schreiben wollten – dies aber aus den unterschiedlichsten Gründen bisher nie konsequent gemacht haben. **Codeception** könnte ein Weg sein, solche Hemmnisse aus dem Weg zu räumen.

Informationen zu Formatierungen und Verweisen

Bei Verweisen habe ich mich bemüht deutsche Quellen zu finden. Leider war dies nicht immer möglich. Manchmal waren die Inhalte, auf die ich in englischer Sprache verweisen konnte, inhaltlich passender. Deshalb finden Sie an manchen Stellen Verweise auf Internetseiten, die in englischer Sprache verfasst sind.

Ein Buch im Bereich Programmierung enthält **Programmcode**. Um diesen Code vom normalen Fließtext abzugrenzen habe ich ihn in einer anderen Schriftart etwas eingerückt. Relevante Teile sind fett abgedruckt.

```
/**
 * @dataProvider provider_credentials_emptypassword
 */
public function testonUserAuthenticate_EmptyPassword($credentials) {
    require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
    $subject = \JeventDispatcher::getInstance();
    $config = array(
        'name' => 'joomla',
        'type' => 'authentication',
        'params' => new \JRegistry
    );
}
```

Kommandozeilen Ein- und Ausgaben sind ebenfalls in einer anderen Schriftart eingerückt. Zusätzlich habe ich diese grau hinterlegt und eingerahmt.

```
$ tests/codeception/vendor/bin/codecept generate:test unit
/suites/plugins/authentication/joomla/PlgAuthenticationJoomla
Test was created in
/var/www/html/gsoc16_browser-automated-tests/tests/codeception/unit//suites/plugins/authentication/
joomla/PlgAuthenticationJoomlaTest.php
```

Randbemerkungen ergänzen den eigentlichen Inhalt dieses Titels. Zum Verständnis der Inhalte dieses Buches sind diese Texte nicht zwingend notwendig. Ich habe Randbemerkungen eingerückt und mit einem Strich am linken Rand versehen.

RANDBEMERKUNG:

Welche Bedeutung hat das Zeichen & vor der Variablen \$response?
Mithilfe des vorangestellten &-Zeichens können Sie eine Variable an eine Methode per Referenz übergeben, so dass die Methode ihre Argumente modifizieren kann.

Der Text, mit dem wichtige **Merksätze** festgehalten werden, ist zum leichteren Wiederfinden grau hinterlegt und mit einem Rahmen versehen.

WICHTIG!

final, private und static Methoden können nicht als PHPUnit Stub-Objekt oder Mock-Objekt verwendet werden. PHPUnit unterstützt diese Methodentypen nicht.

Softwaretests – eine Einstellungssache?

Unter Testen versteht man den Prozess des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen.

[Koomen und Spillner]

In diesem ersten Kapitel erkläre ich Ihnen theoretische Grundlagen. Wenn Sie lieber praktisch starten, können Sie dieses Kapitel zunächst links liegen lassen und mit dem zweiten Kapitel *Praxisteil: Die Testumgebung einrichten* beginnen. Ich verweise an passender Stelle immer mal wieder auf diesen ersten Theorie-Teil.

Ich habe bewusst ein fertiges System, hier konkret das [Content Management System Joomla!](#), als Beispiel für Erklärungen gewählt. Die Benutzung einer fertigen Anwendung, anstelle von ausschließlich kleinen selbst erstellten Codebeispielen, hat Vorteile und Nachteile. Das Framework Joomla! stellt einen Rahmen zur Verfügung, innerhalb dessen ein Programmierer eine Erweiterung programmieren kann. Aus diesem Rahmen kann ich für dieses Buch Testbeispiele wählen. Ich muss also nicht immer das Rad selbst neu erfinden. Nachteilig ist, dass dieser Rahmen teilweise selbst erklärungsbedürftig ist.

Was ist das Ziel dieses ersten Kapitels? Nach der Lektüre dieses Kapitels sollte Ihnen klar ist, warum Softwaretests in einem Projekt eingeplant werden sollten. Ihnen wird bewusst werden, welchen Einfluss das Integrieren von Tests auf Ihre Arbeit haben kann. Mir ging es so,

- dass ich sicherer in meiner Arbeit wurde,

- Fehler in Spezifikationen eher gefunden und korrigiert habe,
- meine Vorgehensweise im Vorhinein überdacht habe und
- im Ergebnis qualitativ bessere und fehlerfreie Programme erstellt habe.

Um dies mit Beispielen zu veranschaulichen, tauchen wir am Ende dieses Kapitels in die [testgetriebene Softwareentwicklung](#) (Test-Driven-Development, kurz TDD) und die [verhaltensgetriebene Softwareentwicklung](#) (Behaviour-Driven-Development, kurz BDD) ein.

Warum sollten Sie Software testen?

Software zu testen ist auf den ersten Blick nichts Tolles. Viele sind der Meinung, dass dies eine langweilige Tätigkeit ist! Außerdem erscheint es auch nicht wichtig Software zu testen. Schon im Studium war dieser Themenbereich ganz am Schluss eingeordnet und in meinem Fall blieb dafür gar keine Zeit. Prüfungsrelevant waren Testmethoden nicht. Demotiviert hat mich zusätzlich die Tatsache, dass Qualität nicht sicher mit Tests belegt werden kann. Dass der ideale Test nicht berechenbar ist, hat [Howden](#) schon 1977 bewiesen. Andererseits hat mich der Satz

„While it is true that quality cannot be tested in, it is equally evident that without testing it is impossible to develop anything of quality.“
[[James Whittaker](#)]

nachdenklich gestimmt. Obwohl es stimmt, dass Qualität nicht sicher mit Tests belegt werden kann, ist es ebenso offensichtlich, dass es unmöglich ist, ohne Tests etwas qualitativ Gutes zu entwickeln. Es gibt sogar Entwickler, die noch weiter gehen und die Meinung vertreten, dass Softwareentwicklung ohne Tests so ähnlich ist wie Klettern ohne Seil und Haken.

Zu 100 % fehlerfreie Software gibt es nicht!

Ursachen fehlerhafter Software sind menschliche Fehlleistungen. Die meisten Fehler entstehen beim Informationsaustausch – also der Kommunikation.

- Intrapersonale Kommunikation:
Mögliche Fehlerursachen bei der Informationsverarbeitung innerhalb eines Menschen sind Denken und/oder Wahrnehmungsfehler. Beispiel: Ein Programmierer weist den Wert einer Variablen brutto statt netto zu.

- **Interpersonale Kommunikation:**
Beim Informationsaustausch zwischen Personen kann ein Erklärungsirrtum auftreten. Jemand hat A gesagt, er hat aber B gemeint.
- **Irrtum bei der Übermittlung:**
Fehler treten auch bei der Übermittlung von Informationen auf. Zum Beispiel dann, wenn ein Mitarbeiter Informationen aus einem entgegengenommenen Anruf falsch weitergibt.
- **Irrtum beim Entschlüsseln:**
Wenn Informationen falsch gelesen oder gehört werden kann es auch zu Fehlern kommen. Hinzu kommt: Immer mehr Menschen kommunizieren heutzutage nicht in ihrer Muttersprache. Dies kann zu Irrtümern führen. Außerdem gibt es immer mehr Abkürzungen: In manchen Unternehmen versteht man kein Wort, wenn man nicht die dort gängige Fachsprache beherrscht. Auch dies kann zu Missverständnissen führen.
- **Kognitive Einschränkungen:**
Wir Menschen haben zu wenig Arbeitsspeicher! Unser Langzeitgedächtnis kann zwar viele Informationen speichern. Im Kurzzeitgedächtnis ist aber nur wenig Speicherplatz. Oft reicht dieser nicht aus, um zwei Schleifendurchläufe inklusive Kontext zusammenhängend nachzuvollziehen! Ab einer bestimmten Komplexität ist ein Programm für den Menschen nicht mehr begreifbar. Genau wie ein Maurer ein Haus Stein für Stein baut, schreiben wir Programme Zeile für Zeile. Wir betrachten Programme durch ein extrem kleines kognitives Fenster!
- **Nicht-kommunikative Fehlerquellen:**
Fehler entstehen aber nicht nur aufgrund von Kommunikationsproblemen. Andere Fehlerquellen sind Mitarbeiter, die nicht über ein ausreichendes fachliches oder projektspezifisches Wissen verfügen oder zu viel Stress ausgesetzt sind. Zudem machen Menschen Fehler, wenn sie übermüdet, krank oder unmotiviert sind.

Menschliche Fehlleistungen können nie ausgeschlossen werden. Die 100 % fehlerfreie Software kann es somit nicht geben. Beim Testen von Software geht es im Wesentlichen um die Minimierung von Fehlern!

Probieren Sie es selbst aus

Integrieren Sie Tests in Ihr nächstes Projekt. Vielleicht springt der Funke auch bei Ihnen über, wenn Sie das erste Mal hautnah erlebt haben, dass ein Test Ihnen eine

mühsame Fehlersuche erspart hat. Denn: Mit einem Sicherheitsnetz von Tests können Sie mit weniger Stress hochwertige Software entwickeln.

Möchten Sie, dass die Software, die Sie programmieren, qualitativ gut ist und Sie selbst entspannter arbeiten können? In diesem Kapitel habe ich dargelegt, dass dies ohne Tests nicht möglich ist.

Testen sollte aber auch kein Selbstzweck sein! Deshalb stellt sich als Nächstes die Frage, wie intensiv und auf welche Art idealerweise Tests integriert werden. Und dies ist die ideale Überleitung zum Thema Projektmanagement.

Projektmanagement

Das **Magische Dreieck** beschreibt den Zusammenhang zwischen den **Kosten**, der benötigten **Zeit** und der leistbaren **Qualität**. Ursprünglich wurde dieser Zusammenhang im Projektmanagement erkannt und beschrieben. Sie haben aber sicher auch schon in anderen Bereichen von diesem Spannungsverhältnis gehört. Es ist bei fast allen betrieblichen Abläufen in einem Unternehmen ein wichtiges Thema.

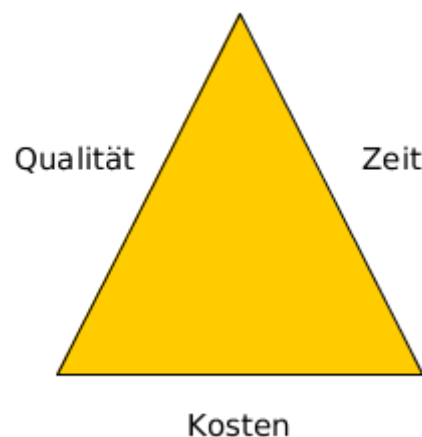


Abbildung 1: Das Magische Dreieck im Projektmanagement
965a.png

Zum Beispiel geht man allgemein davon aus, dass ein höherer Kostenaufwand positive Auswirkungen auf die Qualität und/oder den Fertigstellungstermin – also die Zeit – hat.

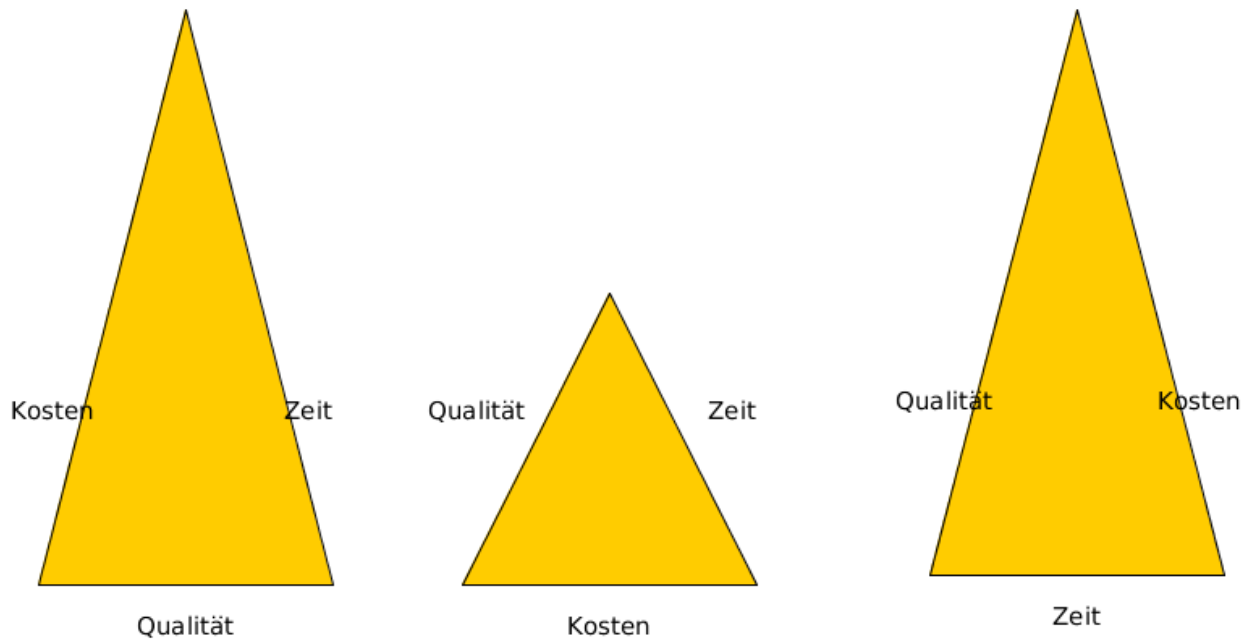


Abbildung 2: Das Magische Dreieck im Projektmanagement – Wenn mehr Geld ins Projekt investiert wird, hat dies positive Auswirkungen auf Qualität oder Zeit.
965b.png

Umgekehrt wird eine Kosteneinsparung zwangsweise die Qualität mindern und/oder die Fertigstellung verzögern.



Abbildung 3: Das Magische Dreieck im Projektmanagement – Wenn weniger Geld ins Projekt investiert wird, hat dies negative Auswirkungen auf Qualität oder Zeit.
965c.png

Nun kommt die Magie ins Spiel: Wir überwinden den Zusammenhang zwischen Zeit, Kosten und Qualität! Denn, auf lange Sicht kann dieser tatsächlich überwunden werden.

WICHTIG!

Der Zusammenhang zwischen Zeit, Kosten und Qualität kann auf lange Sicht überwunden werden.

Vielleicht haben auch Sie schon einmal in der Praxis selbst erlebt, dass eine Qualitätssenkung langfristig keine Kosteneinsparungen zur Folge hat. Die **technische Schuld**, die dadurch entsteht, führt oft sogar zu Kostenerhöhungen und zeitlichem Mehraufwand.

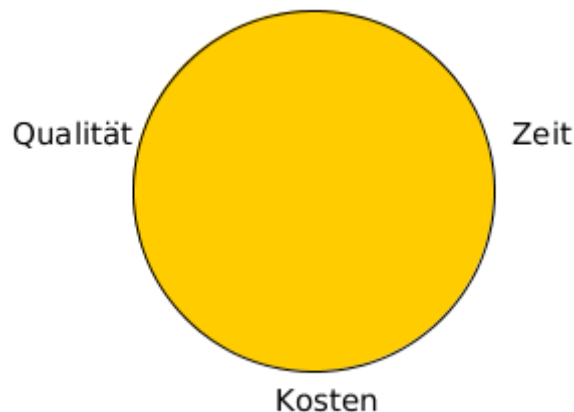


Abbildung 4: Auf lange Sicht kann der Zusammenhang zwischen Kosten, Zeit und Qualität tatsächlich überwunden werden. 965d.png

RANDBEMERKUNG (Technische Schuld):

Unter der technischen Schuld versteht man den Aufwand, den man für Änderungen und Erweiterungen an minderwertig programmierter Software, im Vergleich zu gut geschriebener Software, zusätzlichen einplanen muss. [Martin Fowler](#) unterscheidet folgende [Arten von technischen Schulden](#): Diejenigen, die man bewusst aufgenommen hat und diejenigen, die man ungewollt eingegangen ist. Darüber hinaus unterscheidet er zwischen umsichtigem und risikofreudigem Eingehen von technischer Schuld.

Und nun sind wir bei einem Thema angekommen, das sehr unterschiedlich diskutiert wird: Wie schaffen wir es, Kosten in der Planung genau zu berechnen und darauf aufbauend im zweiten Schritt, Kosten und Nutzen in idealer Weise zu verbinden?

Kosten Nutzen Rechnung

In der Literatur finden Sie immer wieder niederschmetternde Statistiken über die Erfolgsaussichten von Softwareprojekten. Es hat sich wenig an dem negativen Bild geändert, das bereits eine [Untersuchung von A.W. Feyhl](#) in den 90er Jahren aufzeichnete. Hier wurde bei einer Analyse von 162 Projekten in 50 Organisationen die Kostenabweichung gegenüber der ursprünglichen Planung ermittelt: 70 % der Projekte wiesen eine Kostenabweichung von mindestens 50 % auf!

Da stimmt doch etwas nicht! Das kann man doch nicht einfach so hinnehmen, oder?

Ein Lösungsweg wäre, ganz auf Kostenschätzungen zu verzichten und der Argumentation der [#NoEstimates-Bewegung](#) zu folgen. Diese vertritt die Meinung, dass Kostenschätzungen in einem Softwareprojekt unsinnig sind. Ein Softwareprojekt beinhaltet nach Meinung von #NoEstimates immer die Erstellung von etwas Neuem. Das Neue ist nicht mit bereits existierenden Erfahrungen vergleichbar und somit nicht prognostizierbar.

Je länger ich als Entwicklerin arbeite, desto mehr komme ich zu der Überzeugung, dass extreme Sichtweisen nicht gut sind. Die Lösung liegt fast immer in der Mitte. Ich empfehle Ihnen: Vermeiden Sie auch bei Softwareprojekten Extreme und suchen Sie nach einem Mittelweg. Ich bin der Meinung, dass man keinen 100 % sicheren Plan als Ziel haben muss. Man sollte aber auch nicht blauäugig an ein neues Projekt herangehen.

Obwohl das Management von Softwareprojekten und insbesondere die Kostenschätzung ein wichtiges Thema ist, werde ich Sie in diesem Buch nicht länger damit langweilen. Der Schwerpunkt dieses Buches liegt darin aufzuzeigen, wie Softwaretests in den praktischen Arbeitsablauf bei der Entwicklung von Software integriert werden können.

Softwaretests in den Arbeitsablauf integrieren

Sie haben sich dazu entschieden Ihre Software zu testen. Schön! Wann tun Sie dies am besten? Schauen wir uns dazu die Kosten für die Behebung eines Fehlers in den unterschiedlichen Projektphasen an.

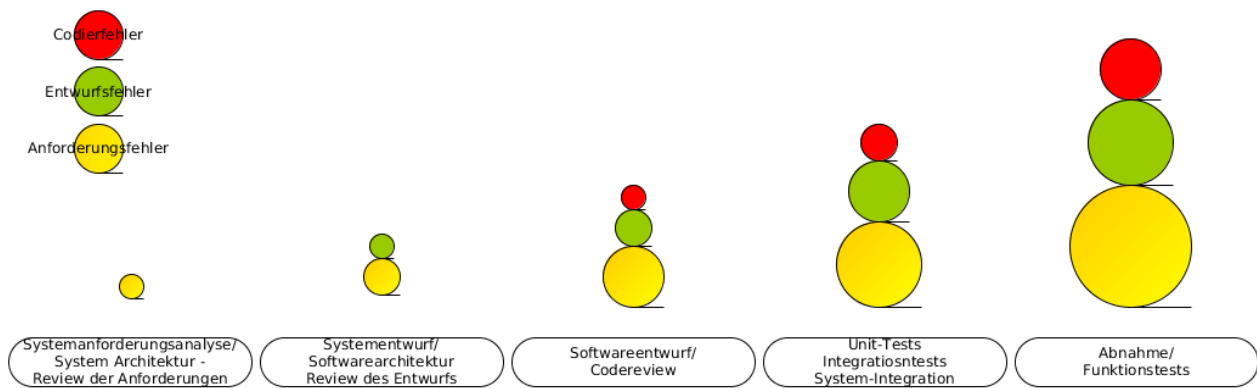


Illustration 5: Relative Kosten für die Fehlerbehebung in den unterschiedlichen Projektphasen 997.png

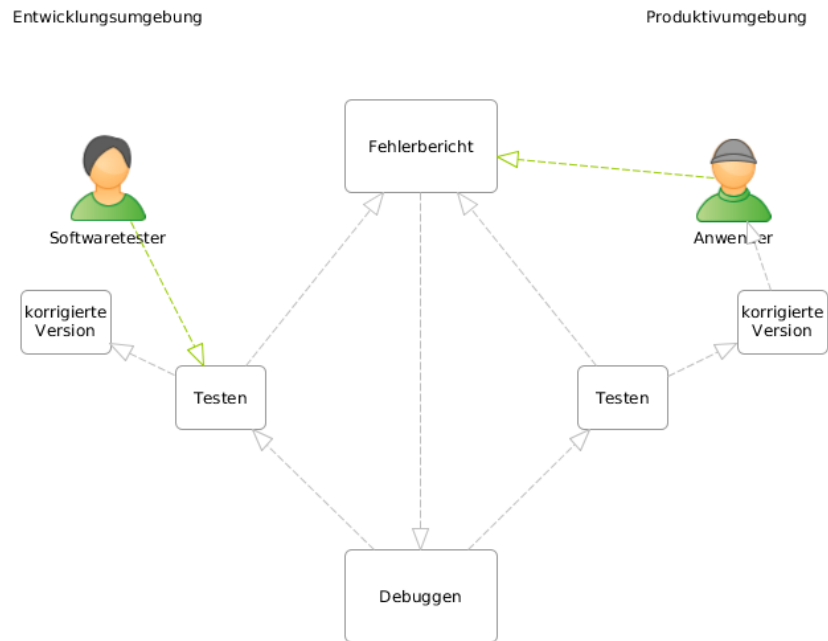
Je früher Sie einen Fehler finden, desto geringer sind die Kosten für die Fehlerkorrektur.

RANDBEMERKUNG (Testen und Debuggen):

Es gibt Worte, die oft in einem Atemzug genannt werden und deren Bedeutung deshalb gleichgesetzt wird. Bei genauer Betrachtung stehen die Begriffe aber für unterschiedliche Auslegungen. Testen und Debuggen gehören zu diesen Worten. Die beiden Begriffe haben gemein, dass Sie Fehlfunktionen aufdecken. Es gibt aber auch Unterschiede in der Bedeutung.

Tests finden unbekannte Fehlfunktionen während der Entwicklung. Dabei ist das Finden der Fehlfunktion aufwendig und teuer, die Lokalisation und Behebung des Fehlers ist hingegen billig.

Debugger beheben Fehlfunktionen, die nach der Fertigstellung des Produktes gefunden werden. Dabei ist das Finden der Fehlfunktion gratis, die Lokalisation und Behebung des Fehlers aber teuer.



Kontinuierliche Testes

Kontinuierliche Integration von Tests

Stellen Sie sich folgendes Szenario vor. Die neue Version eines beliebigen Content Management Systems soll veröffentlicht werden. Alles das, was die Entwickler des Teams seit der letzten Veröffentlichung beigetragen haben, wird nun das erste Mal zusammen eingesetzt. Die Spannung steigt! Wird alles funktionieren? Werden alle Tests erfolgreich sein – falls das Projekt überhaupt Tests integriert. Oder muss die Freigabe der neuen Version doch wieder verschoben werden und es stehen nervenaufreibende Stunden der Fehlerbehebung an? Ganz nebenbei ist das Verschieben des Veröffentlichungszeitpunkts auch nicht gut für das Image des Softwareproduktes!

Das eben beschriebene Szenario mag wohl kein Entwickler gerne miterleben. Viel besser ist es doch, jederzeit zu wissen, in welchem Zustand sich das Softwareprojekt gerade befindet? Weiterentwicklungen, die nicht zum bisherigen Bestand passen, sollten erst integriert werden, nachdem diese „passend“ gemacht wurden. Gerade in Zeiten, in denen es immer häufiger vorkommt, dass eine Sicherheitslücke behoben werden muss, sollte ein Projekt auch stets in der Lage sein, eine Auslieferung erstellen zu können! Und hier kommt das Schlagwort **kontinuierliche Integration** ins Spiel.

Bei der kontinuierlichen Integration werden einzelne Bestandteile der Software permanent integriert. Die Software wird in kleinen Zyklen erstellt und getestet. Auf

Probleme bei der Integration oder fehlerhafte Tests stoßen Sie so frühzeitig und nicht erst Tage oder Wochen später. Bei einer sukzessiven Integration ist die Fehlerbehebung wesentlich einfacher, weil die Fehler zeitnah zur Programmierung entdeckt werden und in der Regel nur ein kleiner Programmteil betroffen ist. Im letzten Kapitel *Automatisierung – Gestatten mein Name ist Jenkins* stelle ich Ihnen kurz die webbasierte Software [Jenkins](#) vor, die Sie bei einer stetigen Integration neuer Softwareteile unterstützen kann.

WICHTIG

Bei einer kontinuierlichen Integration von neuer Software ist die Fehlerbehebung wesentlich einfacher, weil die Fehler zeitnah zur Programmierung entdeckt werden und in der Regel nur ein kleiner Programmteil betroffen ist.

Damit Sie bei einer kontinuierlichen Integration auch jederzeit Tests für alle Programmteile zur Verfügung haben sollten Sie testgetrieben entwickeln.

Testgetriebene Entwicklung (TDD)

Die testgetriebene Entwicklung ist eine Programmiertechnik, bei der in kleinen Schritten entwickelt wird. Dabei schreiben Sie als Erstes den Testcode. Erst danach erstellen Sie den zu testenden Programmcode. Jede Änderung am Programm wird erst vorgenommen, **nachdem** der Testcode für diese Änderung erstellt wurde. Ihre Tests schlagen also unmittelbar nach der Erstellung fehl. Die geforderte Funktion ist ja noch nicht im Programm implementiert. Erst danach erstellen Sie den eigentlichen Programmcode – also den Programmcode der den Test erfüllt. Die Tests helfen Ihnen dabei, das **Programm richtig** zu schreiben. Wenn Sie das erste Mal von dieser Technik hören, können Sie sich vielleicht nicht so recht mit diesem Konzept anfreunden. „Mensch“ will doch immer erst etwas Produktives machen. Und das Schreiben von Tests wirkt auf den ersten Blick nicht produktiv. Sehen Sie sich mein praktisches Beispiel im Kapitel *Unittests* an. Manchmal freundet man sich erst nach dem Kennenlernen mit einer neuen Technik an!

RANDBEMERKUNG (Regressionstest):

Ein Regressionstest ist ein wiederholt durchgeführter Test. Durch die Wiederholung wird sichergestellt, dass Modifikationen in bereits getesteten

Teilen der Software keinen neuen Fehler oder keine neue Regression verursachen. Wofür sind Regressionstests gut? Sie sollten Tests wiederholt durchführen, weil bestimmte Fehlfunktionen manchmal plötzlich wieder auftauchen. Zum Beispiel

- bei der Verwendung von Versionskontrollsoftware beim Zusammenführen mit alten fehlerhaften Versionen.
- aufgrund von Maskierung: Fehlfunktion A tritt aufgrund der unkorrekten Änderung B nicht mehr auf. Der neue Defekt B maskiert die Fehlfunktion A. Nachdem B gefixt ist, tritt Fehlfunktion A wieder auf.

Behaviour-Driven-Development (BDD)

BDD ist keine weitere Programmier- oder Testtechnik, sondern eine Art Best Practices für die Entwicklung von Software. BDD kommt idealerweise gemeinsam mit TDD zum Einsatz. Im Prinzip steht [Behaviour-Driven-Development](#) dafür, nicht die Implementierung des Programmcodes, sondern die Ausführung zu testen – also das Verhalten des Programms. Ein Test prüft, ob die Spezifikation, also die Anforderung des Kunden, erfüllt ist. Wenn Sie Software verhaltensgetrieben entwickeln, helfen Ihnen Tests nicht nur dabei, **das Programm richtig zu schreiben**. Die Tests unterstützen Sie auch dabei, **das richtige Programm zu schreiben**. Beim Behaviour Driven Development werden die Anforderungen an die Software durch Beispiele, sogenannten Szenarien oder [User Storys](#), beschrieben.

Merkmale des Behaviour Driven Development sind

- eine starke Einbeziehung des Endnutzers in den Entstehungsprozess der Software,
- die Dokumentation aller Projektphasen mit Fallbeispielen in Textform – üblicherweise in der Beschreibungssprache [Gherkin](#),
- das automatische Testen dieser Fallbeispiele,
- eine sukzessive Implementierung.

So kann jederzeit auf eine Beschreibung der umzusetzenden Software zugegriffen werden. Mithilfe dieser Beschreibung können Sie fortlaufend die Korrektheit des bereits implementierten Programmcodes sicherstellen. Eine Anleitung dazu, wie Sie Gherkin Texte in Akzeptanztests verwenden, finden Sie im Kapitel *Akzeptanztests*.

RANDBEMERKUNG (Grundlegende Teststrategien):

Grundsätzlich können Sie **spezifikationsorientierte** und **implementierungsorientierte** Testverfahren unterscheiden. In beiden Verfahren gibt es **statische** und **dynamische** Prüfverfahren. Statische Prüfverfahren prüfen das Programm, ohne es auszuführen. Dynamische prüfen das Programm während es ausgeführt wird.

Bei **spezifikationsorientierten Tests** werden die Testfälle durch Analyse der Spezifikation gewonnen. Sicherlich haben Sie schon einmal den Begriff **Black-Box-Tests** gehört. Unter diesem Namen ist diese Testvariante bekannter. Black-Box-Tests werden bewusst in Unkenntnis der Programminterna durchgeführt. Ein Vorteil von spezifikationsorientierten Tests ist, dass fehlende Programmteile entdeckt werden – vorausgesetzt die Spezifikation ist vollständig.

Implementierungsorientierte Tests gewinnen Testfälle durch die strukturelle Analyse des Programms. Diese Testvariante kennen Sie vielleicht unter dem Namen **White-Box-Tests**. Implementierungsorientierte Tests sind auch bei fehlender Spezifikation möglich, dabei können aber vergessene Funktionen nicht entdeckt werden.

	implementierungsbezogen	spezifikationsbezogen
statisch	Statische Code-Analyse	Techniken in der Entwurfsphase
dynamisch	Profiler	Diese Tests sind unser Thema.

Tests planen

Alles beginnt mit einem Plan. Sollte es zumindest. Ein Testplan sollte allen Projektbeteiligten Klarheit darüber verschaffen, was wie intensiv getestet werden soll. Der Testplanungsprozess kann sehr komplex sein. Gemäß [ISO 29119-2](#) umfasst er neun umfangreiche Aktivitäten.

Es gibt aber auch andere Vorgehensweisen. Whittaker beschreibt in seinem Buch [How Google Tests Software](#) die praxisnahe und leicht auf dem aktuellen Stand zu haltende Methode **Attributes-Componentes-Capabilities**, kurz ACC. ACC ordnet jeder Komponente verschiedene Attribute wie Benutzerfreundlichkeit, Geschwindigkeit oder Sicherheit zu. Diese Komponente-Attribut-Kombination wird in einer Matrix zusammen mit einem Wert für die Wichtigkeit dieser Komponente-Attribut-Kombination aufgenommen.

	Benutzerfreundlichkeit	Sicherheit	Schnelligkeit
Suche	mittel	niedrig	hoch
Plugin Agpaypal	mittel	hoch	niedrig
Kontaktformular	hoch	mittel	niedrig
...			

Abbildung 6: Eine Beispiel für eine ganz einfache ACC-Matrix 948.png

Egal wie Sie Ihren Testplan erstellen. Wichtig ist meiner Meinung nach das Klarheit darüber herrscht, welche Programmbestandteile wie wichtig sind. Daraus ergibt sich dann, was wie intensiv getestet werden sollte. Wie Sie überprüfen können, ob Ihre Tests tatsächlich so wie geplant implementiert sind, erfahren Sie am Ende dieses Buches im Kapitel *Analyse*. Dieses Kapitel enthält einen Teil, in dem es um die Messung der Testabdeckung geht. Denn: Das Testen aller möglichen Eingabeparameter ist in der Realität unmöglich. Ein systematisches stichprobenartiges Testen ist die einzig praktikable Lösung.

RANDBEMERKUNG:

Warum ist das Testen aller möglichen Eingabeparameter in der Realität unmöglich? Nehmen wir an, die Menge der möglichen Testfälle ist D . Um ein Beispiel zu nennen: Beim Durchsuchen eines Textes nach einem bestimmten Muster, das in einen anderen Text umgewandelt werden soll, kann dieses Muster genau einmal, keinmal oder mehrmals im Text vorkommen.

D = Eingabebereich.

Nehmen wir weiter an, dass es eine Menge an möglichen Ausgabemöglichkeiten gibt und nennen diese R . Soll ein Muster in einem Text in einen anderen Text umgewandelt werden, könnte eine Ausgabemöglichkeit der Eingabetext mit korrekt umgewandeltem Muster sein. Eine andere Ausgabemöglichkeit ist der Eingabetext mit Muster, das fehlerhaft nicht umgewandelt wurde.

R = Ausgabemöglichkeiten.

Während der Programmausführung wandelt das System die Menge D in die Menge R um. Im Folgenden beschreibe ich die Programmausführung formal mit P .

Programmausführung P : $D \rightarrow R$

Ein Spezialfall von P liegt vor, wenn das Programm die Daten so verarbeitet, wie es in der Spezifikation festgelegt wurde. Im folgenden nenne ich diesen Spezialfall F .

F: D → R (Ausgabe-Anforderung für P ist erfüllt)

Das Programm arbeitet also formal gesehen korrekt und fehlerfrei, wenn für alle möglichen Eingaben P gleich F gilt.

$d \in D : P(d) = F(d)$

Eine endliche Teilmenge $T \subseteq D$ ist eine Menge von Testfällen. Dies kann eine Stichprobe sein.

Der ideale Test: Wenn ein Programm an einer Stelle nicht korrekt ist, dann gibt es einen Testfall der dieses unkorrekte Verhalten erzeugt. Ideal ist dieser Testfall, wenn man ihn findet, ohne alle möglichen Testfälle durchprobieren zu müssen. Den idealen Test t in einem fehlerhaften Programm $P(d) \neq F(d)$ findet man, indem man $P(t) \neq F(t)$ bestimmt.

$\exists d \in D: P(d) \neq F(d)$

$\exists t \in T: P(t) \neq F(t)$

Die gute Nachricht: Für jedes Programm gibt es einen idealen Test, der sogar nur einen Testfall enthält:

1. Wenn die Programmausführung fehlerhaft ist, gilt: $\exists d \in D: P(d) \neq F(d)$ – für den idealen Test gilt $T = \{d\}$
2. Wenn die Programmausführung fehlerfrei ist, gilt: $T = \{\}$ – die Menge der idealen Tests ist leer. Es gibt keinen Fehler.

Leider ist die Menge T der idealen Tests nicht einfach zu bestimmen. Wenn dies so wäre, gäbe es sicherlich ausschließlich korrekte Programme. Da der ideale Test nicht berechenbar ist, hat [Howden](#) schon 1977 bewiesen.

Was sollten Sie beim Generieren von Tests beachten?

Sie haben nun sehr viel Theorie zum Thema Softwaretests gelesen. In diesem Buch werde ich ihnen einige Werkzeuge erklären, die Sie in der Praxis beim Generieren von Tests unterstützen. Wenn Sie diese Tools einsetzen, werden Sie selbst erfahren, wie Sie Ihre Tests am besten schreiben. Da jeder Entwickler seine individuelle Vorgehensweise hat, gibt es viele Dinge, die man nicht allgemein als Regel mitgeben kann. Es gibt aber drei Regeln, die sich allgemein durchgesetzt haben:

1. Ein Test sollte **wiederholbar** sein.
2. Ein Test sollte **einfach** gehalten sein.

3. Ein Test sollte **unabhängig** von anderen Tests sein.

RANDBEMERKUNG:

Neben den später beschriebenen Methoden können Sie auch mit [Docker](#) Tests unabhängig voneinander ablaufen lassen. Eine kurze Beschreibung dazu, wie Sie Docker mit Codeception zusammen nutzen können, finden Sie in der [Dokumentation von Codeception](#).

Kurzgefasst

Wenn Sie dieses Kapitel gelesen haben wissen Sie, warum Softwaretests wichtig sind und welche Vorteile eine testgetriebene Entwicklung bringt. Die können erklären, warum eine verhaltensgetriebene Entwicklung keine weitere Testtechnik ist, sondern eine Art Best Practice. Außerdem ist Ihnen nun bewusst, warum Sie kontinuierlich neue Codeteile in Ihr Programm integrieren sollten.

Praxisteil: Die Testumgebung einrichten

Program testing can be used to show the presence of bugs, but never show their absence!

[Edsger W. Dijkstra]

In diesem Kapitel arbeiten wir endlich praktisch. Da dieses Buch die Anwendung von Software zum Thema hat, ist es nicht dazu geeignet auf dem Sofa oder am Strand gelesen zu werden. Das Erlernen von Software funktioniert meiner Meinung nach am besten, wenn alle Beispiele am Computer selbst nachvollzogen werden. Am Ende dieses Kapitels werden Sie Joomla! und eine erste kleine eigene Erweiterung auf Ihrem lokalen Webserver installiert haben. Diese Erweiterung wird dann auch die Grundlage für den Aufbau der Beispieltests in diesem Buch sein.

Joomla! ist ein Content Management System, mit dem Sie eine Website erstellen und deren Inhalte mithilfe eines [WYSIWYG](#) Editors pflegen können. Die Erweiterung, die wir beispielhaft erstellen, soll einen Benutzer dabei unterstützen einen Jetzt-kaufen-Button von PayPal in einen Beitrag zu integrieren. Der Benutzer muss hierzu nur ein bestimmtes Textmuster kennen und dieses in einen Beitrag einfügen. Wenn dieser Beitrag dann von Joomla! für die Anzeige auf der Internetseite präpariert wird, kommt

die Erweiterung zum Einsatz. Sobald diese während der Beitragserstellung auf das definierte Textmuster stößt, wandelt sie dieses in ein HTML-Element um, welches einen PayPal Jetzt-kaufen-Button anzeigt.

Das Erstellen dieser Erweiterung ist aber nicht das eigentliche Ziel dieses Kapitels. Am Ende dieses Kapitels sollten Sie wissen, mit welchen Testtypen Sie Ihre Software testen können und welche Teststrategien sie anwenden können.

Entwicklungsumgebung und Arbeitsweise

Entwickeln Sie bereits Software? Dann arbeiten Sie sicherlich in Ihrer persönlichen Entwicklungsumgebung, in der Sie sich sicher und wohl fühlen. Falls dies nicht so ist, sollten Sie sich diese Entwicklungsumgebung aufbauen. Ein Computer, auf dem Werkzeuge installiert sind, die Sie bei der Arbeit unterstützen ist eine Voraussetzung für die Erstellung guter Software. Ohne eine solche Umgebung werden Sie sicherlich keinen Spaß an Ihrer Arbeit haben.

Als Leitfaden beschreibe ich Ihnen kurz die – für das Mitmachen der Übungen hier im Buch – wichtigsten Teile meiner Entwicklungsumgebung.

- Ich arbeite mit dem Betriebssystem [Ubuntu](#). Aktuell verwende ich die Version 16.04 LTS
- Zum Bearbeiten des Programmcodes verwende ich die integrierte Entwicklungsumgebung (IDE) [Netbeans](#). Theoretisch können Sie einen einfachen Texteditor verwenden. Eine IDE bietet Ihnen jedoch eine Menge mehr Komfort.
- Um mir die Installation und die Konfiguration des [Webserver Apache](#) mit der [Datenbank MySQL](#) und der [Skriptsprachen PHP](#) so einfach wie möglich zu machen, verwende ich die Programmkombination [LAMP](#). LAMP ist ein Akronym. Die einzelnen Buchstaben des Akronyms stehen für die verwendeten Komponenten Linux, Apache, MySQL und PHP. Eine Anleitung, die die Installation von LAMP unter Ubuntu beschreibt, finden Sie unter der Adresse <https://wiki.ubuntuusers.de/LAMP/> im Internet.

Sie können natürlich die Beispiele im Buch auch mit alternativer Software durchführen. In diesem Fall kann es sein, dass etwas nicht so wie beschrieben läuft und Sie selbst Anpassungen vornehmen müssen.

Joomla! herunterladen und auf einem Webserver installieren

Sie werden feststellen, dass die Installation von Joomla! sehr einfach und intuitiv ist. Zumindest dann, wenn alle Systemvoraussetzungen erfüllt sind.

Ich beschreibe hier die Installation unter Ubuntu Linux mit einer Standard LAMP Installation. Arbeiten Sie mit einem anderen System? Dann passen Sie die Beschreibung bitte an Ihre Systemumgebung an.

Was ist Joomla! eigentlich genau?

Bevor Sie [Joomla!](#) installieren, möchten Sie sicherlich erfahren, worauf Sie sich mit dieser Installation möglicherweise einlassen? Joomla! ist ein [Content Management System](#), kurz CMS, mit dem Sie nicht nur eine Website erstellen und pflegen können. Sie können mit Joomla! leistungsstarke Webanwendungen programmieren.

Wenn Sie Joomla! nutzen möchten, müssen Sie kein Geld dafür bezahlen. Außerdem können Sie den Quellcode einsehen. Joomla! ist eine [Open Source](#) Software die unter der Lizenz [General Public License Version 2 or later](#) (GNU) veröffentlicht ist.

GNU ist die am weitesten verbreitetste Softwarelizenz. Diese erlaubt Ihnen, Joomla! auszuführen, für Lernzwecke zu verwenden, zu ändern und zu kopieren. Software, die diese Freiheitsrechte gewährt, wird auch [Freie Software](#) genannt. Joomla! unterliegt weiterhin einem [Copyleft](#). Das heißt, Sie müssen die eben beschriebenen Rechte bei einer eventuellen Weitergabe beibehalten. Auch dann, wenn Sie Ihrer Meinung nach verbessernde Änderungen vorgenommen haben. Sie dürfen für Ihre Verbesserungen aber Geld berechnen. Freie Software kann – wie im Falle von Joomla! – kostenlos sein. Freie Software muss aber nicht kostenlos sein!

Voraussetzungen

Die Anforderungen von Joomla! bezüglich PHP-Version, unterstützter Datenbanken und unterstütztem Web-Server sind nicht sehr hoch. Wahrscheinlich werden Sie keine Probleme haben. Da sich die Mindestanforderungen von Version zu Version ändern, gebe ich hier nur einen Link an. Die aktuellen Systemvoraussetzungen können Sie unter der Adresse <https://downloads.joomla.org/de/technical-requirements-de> einsehen.

Download des Joomla! Installationspaketes

Besorgen Sie sich nun das aktuelle Joomla! Installationspaket. Die neueste Version finden Sie immer unter der Adresse <https://downloads.joomla.org>. Ich habe die Version 3.6.5 – also die Datei Joomla_3.6.5-Stable-Full_Package.zip – installiert.

Upload der Joomla! Installationsdateien auf den lokalen Webserver

Verschieben Sie das heruntergeladene Installationspaket in das Stammverzeichnis Ihres lokalen Webserver und entpacken Sie es dort. Wenn Sie wie ich die Standardinstallation von LAMP nutzen, sollten Sie das Installationspaket in das Verzeichnis `/var/www/html` kopieren. Nach dem Entpacken sehen Sie dann das Unterverzeichnis `/var/www/html/Joomla_3.6.5-Stable-Full_Package`. Der Einfachheit halber benennen Sie dieses Verzeichnis bitte in `/var/www/html/joomla` um.

Nun können Sie die Installationsroutine von Joomla! in Ihrem Internetbrowser über die URL `http://localhost/joomla` aufrufen. Probieren Sie es aus! Wenn alles richtig läuft, werden Sie mit der Hauptkonfigurationsseite begrüßt und können sofort mit dem nächsten Kapitel fortfahren.

Hauptkonfiguration



Abbildung 7: Joomla! Hauptkonfiguration 992

Der weitere Ablauf der Konfiguration ist meiner Meinung nach sehr intuitiv und selbsterklärend. Falls Sie doch Fragen haben, hilft Ihnen vielleicht die ausführlichere [Installationsanleitung](#) in der Joomla! eigenen Dokumentation weiter. Gerne werden auch Fragen im [deutschen Joomla! Forum](#) beantwortet.

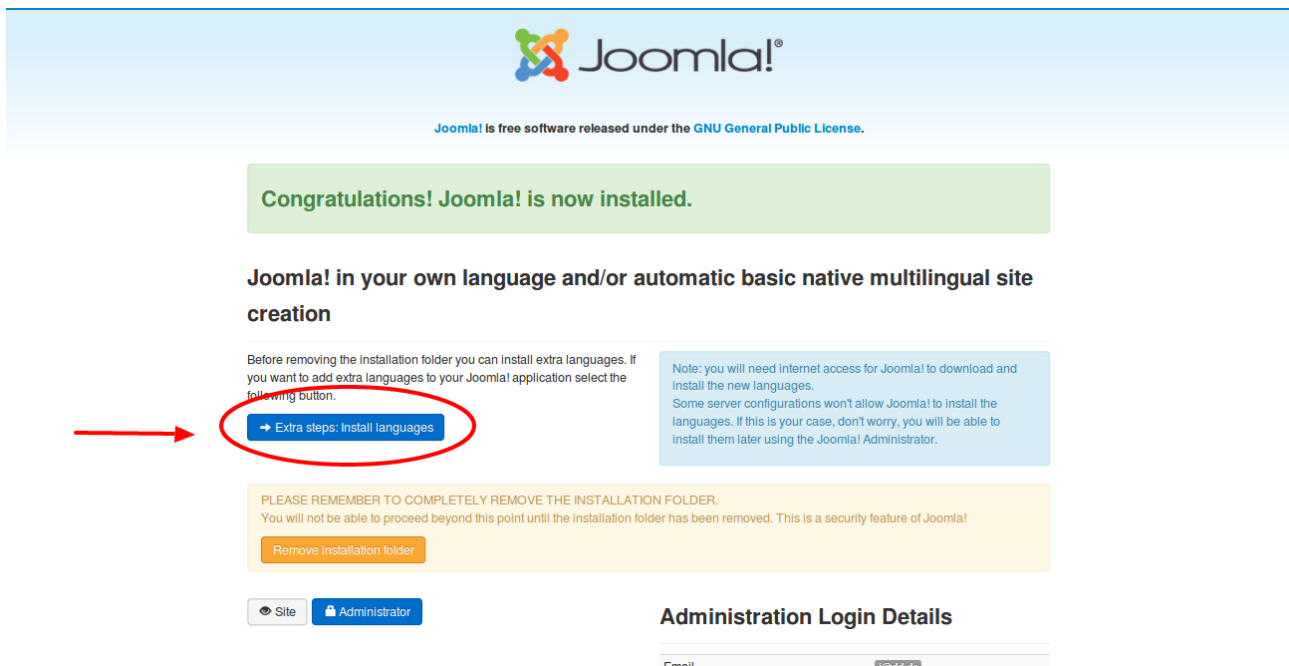
Damit wir gleiche Voraussetzungen haben wäre es gut, wenn Sie im letzten Schritt der Installation auf Beispieldaten verzichten und die deutsche Sprache als

Standardsprache installieren. Andernfalls müssen Sie später beim Erstellen der Tests Anpassungen vornehmen.



The screenshot shows the Joomla! installation summary screen. At the top, the Joomla! logo and the text "Joomla! ist freie Software. Veröffentlicht unter der GNU General Public License." are visible. Below this, there are three tabs: "1 Konfiguration", "2 Datenbank", and "3 Überblick". The "3 Überblick" tab is selected. The main heading is "Zusammenfassung". Below it, there are two buttons: "← Zurück" and "→ Installieren". The section "Beispieldaten installieren" has a red circle around the "Keine" option, which is selected. Below this, there are four radio button options for English sample data: "Englische (GB) Beispieldaten: Bloginhalte", "Englische (GB) Beispieldaten: Prospektinhalte", "Englische (GB) Beispieldaten: Standardinhalte", and "Englische (GB) Beispieldaten: Joomla! erlernen". A red arrow points to the "Keine" option. Below the radio buttons, there is a paragraph of text: "Anfängern wird dringend empfohlen diese Daten zu installieren. Hiermit werden die Beispieldaten eingefügt, die dem Installationspaket von Joomla! beiliegen." Below this, there is a section "Überblick" with a "Konfiguration senden" button and a "Ja" button. Below the "Ja" button, there is a text field with "admin@example.de" and a "Nein" button. Below this, there are two sections: "Hauptkonfiguration" and "Konfiguration der Datenbank".

Abbildung 8: Die Zusammenfassung am Ende der Joomla! Installation - Hier bitte auf Beispieldaten verzichten 991.png



The screenshot shows the Joomla! installation completion screen. At the top, the Joomla! logo and the text "Joomla! is free software released under the GNU General Public License." are visible. Below this, there is a green box with the text "Congratulations! Joomla! is now installed." Below this, there is a section "Joomla! in your own language and/or automatic basic native multilingual site creation". Below this, there is a paragraph of text: "Before removing the installation folder you can install extra languages. If you want to add extra languages to your Joomla! application select the following button." Below this, there is a red circle around the "→ Extra steps: Install languages" button. A red arrow points to this button. To the right of this button, there is a blue box with the text: "Note: you will need internet access for Joomla! to download and install the new languages. Some server configurations won't allow Joomla! to install the languages. If this is your case, don't worry, you will be able to install them later using the Joomla! Administrator." Below this, there is a yellow box with the text: "PLEASE REMEMBER TO COMPLETELY REMOVE THE INSTALLATION FOLDER. You will not be able to proceed beyond this point until the installation folder has been removed. This is a security feature of Joomla!" Below this, there is a button "Remove installation folder". Below this, there are two buttons: "Site" and "Administrator". Below this, there is a section "Administration Login Details" with a text field for "Email" and a "Log in" button.

Abbildung 9: Extraschritt am Ende der Joomla! Installation - Hier bitte die deutschen Sprachdateien installieren und als Standard auswählen. 991a.png

Ich bin mir sicher, dass Sie Joomla! erfolgreich installiert und konfiguriert haben und den Administrationsbereich in Ihrem Browser nun über die Adresse

<http://localhost/joomla/administrator/> und das Frontend in Ihrem Browser über die Adresse <http://localhost/joomla/> aufrufen können.

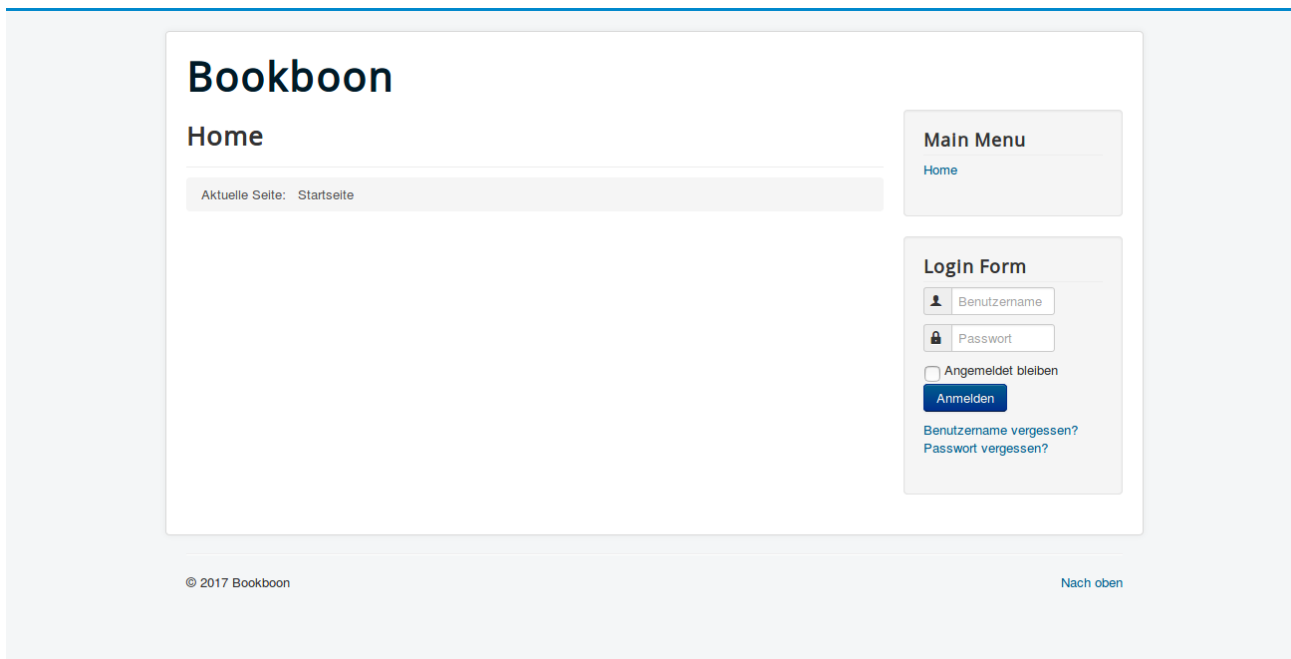


Abbildung 10: Joomla! Frontend unmittelbar nach der Installation – erreichbar über die Adresse <http://localhost/joomla/> 988.png



Abbildung 11: Anmeldemaske zum Joomla! Administrationsbereich – erreichbar über die Adresse <http://localhost/joomla/administrator/> 987.png

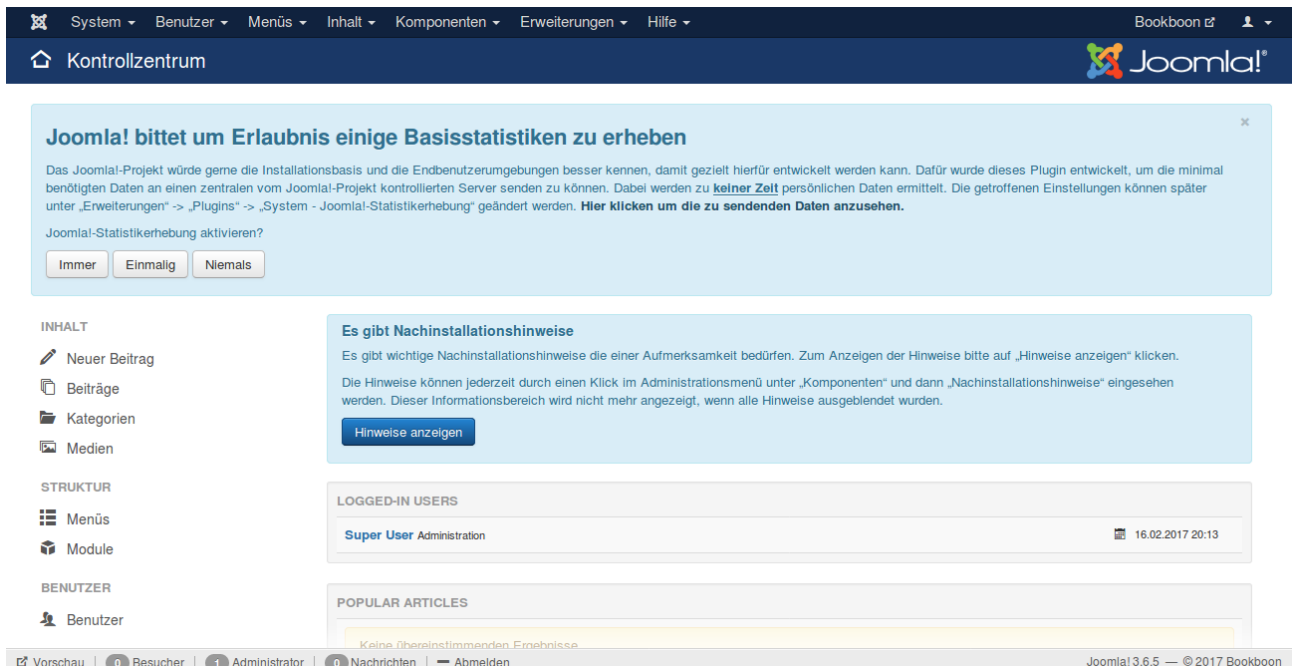


Abbildung 12: Joomla!: Die erste Anmeldung im Administrationsbereich 986.png

In Ihrem Dateisystem finden Sie die Joomla!-Dateien im Verzeichnis

/var/www/html/joomla. Genau finden Sie hier folgende Verzeichnisstruktur vor:

```
/var/www/html/joomla$
```

- administrator
- bin
- cache
- cli
- components
- images
- includes
- language
- layouts
- libraries
- media
- modules
- **plugins**
- templates

```
- tmp
LICENSE.txt
README.txt
configuration.php
htaccess.txt
index.php
robots.txt
web.config.txt
```

RANDBEMERKUNG:

Bitte haben Sie im weiteren Verlauf des Buches immer im Hinterkopf, dass Joomla! ein aktives und lebendes Projekt ist. Es wird ständig weiterentwickelt und verbessert. Das ist auch sehr gut so. Nachteilig ist nur, dass ich nicht sicherstellen kann, dass in allen zukünftigen Versionen alles genauso ist, wie ich es Ihnen hier erkläre. Sie sollten aber trotzdem immer die neueste Version von Joomla! verwenden – schon alleine aus Sicherheitsgründen.

Die Joomla! Architektur verstehen

Wie jedes System besteht Joomla! aus mehreren Elementen. Und wie jedes System ist es mehr als die Summe der einzelnen Elemente!

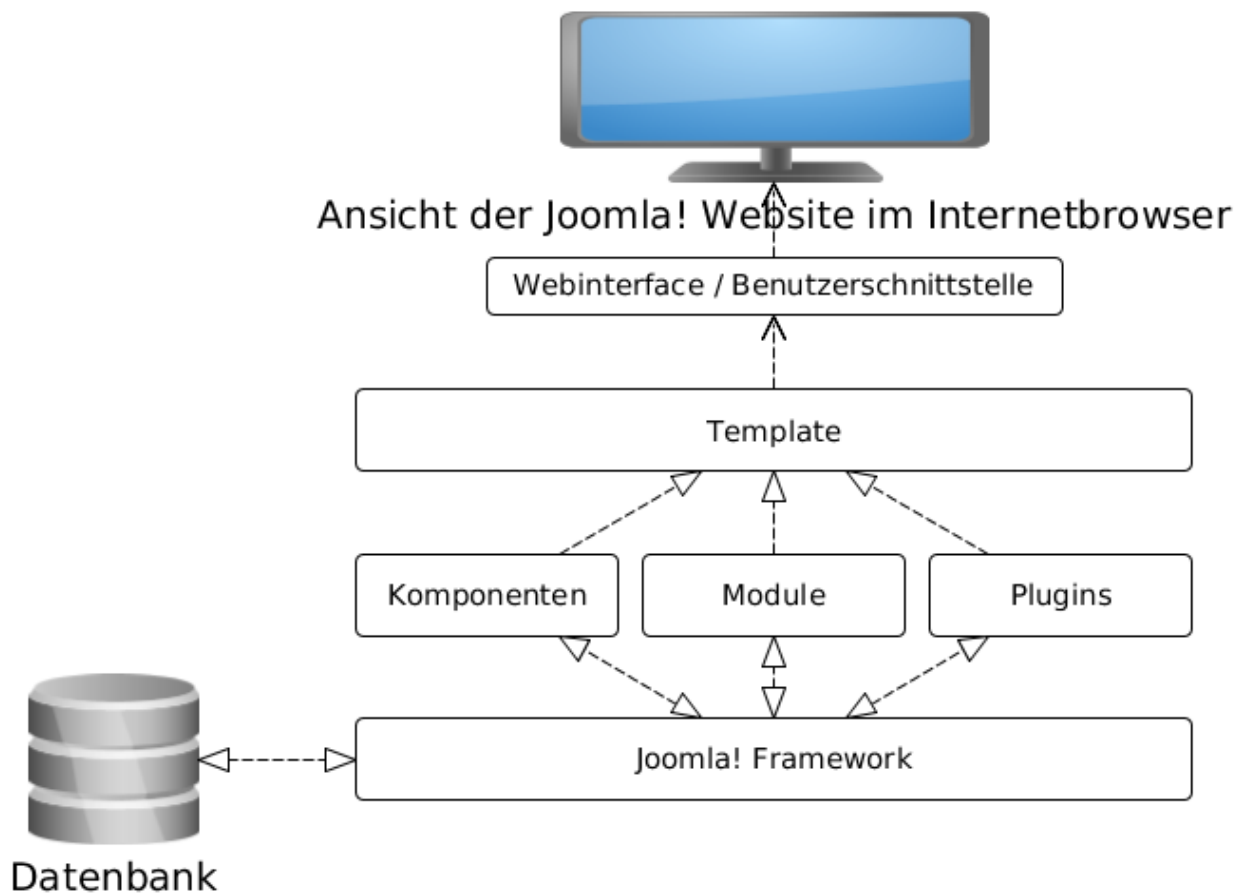


Abbildung 13: Joomla! Architektur 996.png

Joomla! ist ein modular aufgebautes System, das bereits in der Basisinstallation zahlreiche, nützliche Erweiterungen mitbringt.

Datenbank

Alle Inhalte Ihrer Joomla! Website, mit Ausnahme von Bildern und Dateien im Unterverzeichnis `/images`, werden in einer Datenbank gespeichert.

Joomla! Framework

Das Joomla! Framework ist eine Sammlung von Open Source Software, auf der das Joomla! CMS aufbaut.

Erweiterungen

Erweiterungen erweitern, wie der Name schon sagt, die Basis von Joomla! – also das Joomla! Framework. Sie können zwei Arten und vier Typen von Erweiterungen unterscheiden:

Erweiterungsarten

Joomla! unterscheidet die geschützten Erweiterungen und Erweiterungen von Drittanbietern.

- Geschützte Erweiterungen.

Geschützte Erweiterungen haben Sie bei der Standardinstallation von Joomla! mit installiert. Joomla! bringt schon in der Basisinstallation zahlreiche nützliche Komponenten, Module und Plugins mit.

- Erweiterungen von Drittanbietern.

Sollte die Basisinstallation für Ihren Bedarf nicht ausreichen, lässt sich das System um beinahe jede erdenkliche Funktion ergänzen. In einem [offiziellen Verzeichnis](#) stehen Ihnen zahlreiche Erweiterungen von Drittanbietern zur Verfügung.

Erweiterungstypen

Für jede Erweiterungsart gibt es verschiedene Erweiterungstypen.

- Komponenten

Unter einer [Komponente](#) können Sie sich eine kleine Anwendung vorstellen. Diese Anwendung erfordert das Joomla! Framework als Grundlage, ansonsten können Sie die Komponente aber eigenständig nutzen und mit ihr interagieren. Ein Beispiel für eine geschützte Komponente ist der [Benutzermanager](#).

- Module

Ein [Modul](#) ist weniger komplex als eine Komponente. Es stellt keinen eigenständigen Inhalt dar, sondern wird beim Aufbau der Seite auf einer festgelegten Position angezeigt. Mit dem [Modulmanager](#) konfigurieren Sie ein Modul. Das wohl bekannteste Modul ist das mit dem Namen [Benutzerdefiniertes Modul \(Custom HTML\)](#). Mit diesem Modul können Sie individuelle Texte mittels der [Hypertext-Auszeichnungssprache HTML](#) an einer bestimmten Position auf Ihrer Website anzeigen.

- Plugins

[Plugins](#) sind relativ kleine Programmcode Teile, die bei Auslösung eines bestimmten Ereignisses ausgeführt werden. Ein Beispiel für ein Ereignis ist die erfolgreiche Anmeldung eines Benutzers. Im Kapitel *Joomla! mit einem eigenen Plugin erweitern* werden wir das einfache Plugin *Agpaypal* schreiben. Das Ereignis, das wir in diesem Plugin ausnutzen werden, ist [onContentPrepare](#). Genau

beschreibt das Ereignis `onContentPrepare` den ersten Schritt bei der Vorbereitung eines Beitrags für die Anzeige im Frontend. Ein Plugin ist eine einfache Art das Joomla! Framework zu erweitern.

- Templates

Das [Template](#) bestimmt das Aussehen Ihrer Joomla! Website.

Joomla! mit einem eigenen Plugin erweitern

Zu Beginn dieses Kapitels haben Sie Joomla! installiert. Danach habe ich Ihnen die wichtigsten Bestandteile von Joomla! erläutert. In diesem Abschnitt werden wir nun eine einfache Erweiterung schreiben. Genau erstellen wir ein [Content Plugin](#). Aufgabe dieser Erweiterung ist es, einen bestimmten Text in einen PayPal Jetzt-kaufen-Button umzuwandeln. Wir werden dazu das Ereignis `onContentPrepare` nutzen. Dieses Ereignis wird in Joomla! beim Vorbereiten eines Beitrags für die Anzeige im Browser ausgelöst. Das Plugin soll dann im weiteren Verlauf die Grundlage für unsere Tests sein.

Ein Joomla! Plugin muss nur aus [zwei Dateien](#) bestehen. Der XML-Installationsdatei oder Manifest Datei und dem eigentlichen Programmcode. Die Dateien müssen in einem bestimmten Verzeichnis abgelegt sein. Plugins, die Inhalte von Beiträgen manipulieren, gehören in ein Unterverzeichnis des Verzeichnisses `plugins\content`. Legen Sie also als Erstes im Verzeichnis `plugins\content` den Ordner `agpaypal` an. In diesem Ordner erstellen Sie als Nächstes die Datei `agpaypal.php`. In meiner Entwicklungsumgebung lege ich also konkret die Datei

`/var/www/html/joomla/plugins/content/agpaypal/agpaypal.php` mit folgendem Inhalt an.

```
<?php
defined('_JEXEC') or die;

class plgContentAgpaypal extends JPlugin {
    public function onContentPrepare($context, &$row, $params, $page = 0){
        $search = "@paypalpaypal@";
        $replace = '<form name="_xclick" action="https://www.paypal.com/de/cgi-
            bin/webscr" method="post">
                <input type="hidden" name="cmd" value="_xclick">
                <input type="hidden" name="business" value="me@mybusiness.com">
                <input type="hidden" name="currency_code" value="EUR">
                <input type="hidden" name="item_name" value="Teddybär">
                <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" alt="Zahlen Sie mit PayPal – schnell,
        kostenlos und sicher!">
    </form>;
    if (is_object($row)){
        $row->text = str_replace($search, $replace, $row->text);
    }
    else{
        $row = str_replace($search, $replace, $row);
    }
    return true;
}
}

```

Was tut das Plugin genau? Zunächst einmal werden die Variablen `$search` und `$replace` erstellt. Die Variable `$search` wird mit einem Suchtext belegt und der Variable `$replace` wird ein Text, der in einem HTML-Dokument einen [PayPal Jetzt-kaufen-Button](#) anzeigt, zugeordnet. Als Nächstes ist es wichtig zu wissen, dass die Variable `$row` den Text des Beitrags, der gerade von Joomla! angezeigt werden soll, enthält. Die Variable `$row` ist entweder ein Objekt, das den Beitragstext in der Eigenschaft `text` enthält, oder eine einfache Variable. In jedem Fall ersetzen wir mithilfe der Funktion `str_replace()` den Suchtext mit dem Text für die Anzeige des PayPal Jetzt-kaufen-Buttons mit `str_replace($search, $replace, $row);` – beziehungsweise `str_replace($search, $replace, $row->text);`.

RANDBEMERKUNG:

Was bedeutet das Zeichen `&` vor dem Parameter `$row` in der Methode `onContentPrepare()`? Mithilfe des vorangestellten `&`-Zeichens können Sie eine Variable an eine Methode per Referenz übergeben, so dass die Methode die Variable modifizieren kann. Beispiel:

```

<?php
function foo(&$var) {
    $var++;
}
$a=5;
foo($a);
// $a ist 6

```



```

<?php
function foo($var) {
    $var++;
}
$a=5;
foo($a);
// $a ist 5

```

Die Datei, die die eigentliche Arbeit erledigt, ist fertig. Nun erstellen Sie im gleichen Verzeichnis die Manifest-Datei agpaypal.xml. Diese Datei ist für die Installation, also das Bekanntmachen des Plugins in Joomla!, wichtig.

```

<?xml version="1.0" encoding="utf-8"?>
<extension version="3.6" type="plugin" group="content">
    <name>Paypal Schaltfläche</name>
    <creationDate>[DATE]</creationDate>
    <author>[AUTHOR]</author>
    <authorEmail>[AUTHOR_EMAIL]</authorEmail>
    <authorUrl>[AUTHOR_URL]</authorUrl>
    <copyright>[COPYRIGHT]</copyright>
    <license>GNU General Public License version 2 or later; see LICENSE.txt</license>
    <version>1.0</version>
    <description>Das Plugin erzeugt eine PayPal "Kaufe jetzt" Schaltfläche.</description>
    <files>
        <filename plugin="agpaypal">agpaypal.php</filename>
    </files>
</extension>

```

Dies ist kein Lehrbuch zum Thema Plugin Programmierung für Joomla!. Deshalb hier nur das Wesentliche ganz kurz. Wichtig ist, dass die XML-Datei [wohlgeformt](#) ist und genauso heißt wie die PHP Datei. Am Anfang der XML-Datei steht immer das XML-Tag `<?xml version="1.0" encoding="utf-8"?>`. In der zweiten Zeile `<extension version="3.6" type="plugin" group="content">` geben Sie wichtige Informationen zur Erweiterung an. In unserem Fall handelt sich um ein Content Plugin für die Joomla! Version 3.6. Im Anschluss können Sie weitere Informationen zu Ihrem Plugin angeben. Zuletzt geben

Sie im Tag <files> alle Dateien, die zum Plugin gehören, an. Das ist in unserem Fall einfach. Unser Plugin besteht bisher – neben der Manifest-Datei – nur aus einer einzigen Datei.

Als Nächstes werden wir die Erweiterung in Joomla! ausprobieren. Wenn wir sicher sind, dass alles klappt, bearbeiten wir die Erweiterung testgetrieben weiter.

Irgendwann möchten Sie sicherlich einmal etwas anderes als einen Teddy für 12,99 Euro verkaufen, oder?

Damit Joomla! Ihre Erweiterung kennenlernt, muss das Content Management System diese noch identifizieren. Öffnen Sie dazu bitte im Administrationsbereich das Menü Erweiterungen | Verwalten | Überprüfen und klicken dann links oben auf die Schaltfläche *Überprüfen*.

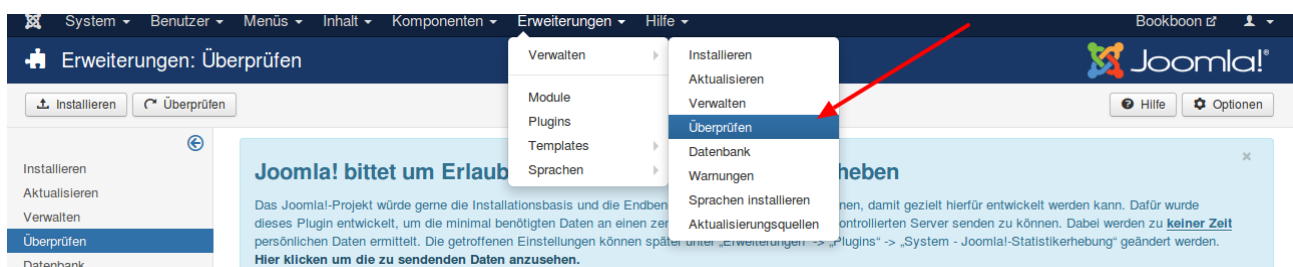


Abbildung 14: Damit Joomla! Ihre Erweiterung kennenlernt, muss das Content Management System diese noch identifizieren. 985.png

Wenn Sie die Plugin Dateien genauso wie von mir beschrieben erstellt haben, sehen Sie nun im Hauptbereich einen Eintrag, der Ihr neu erstelltes Plugin enthält. Wählen Sie diesen Eintrag aus und klicken danach links oben auf die Schaltfläche Installieren.

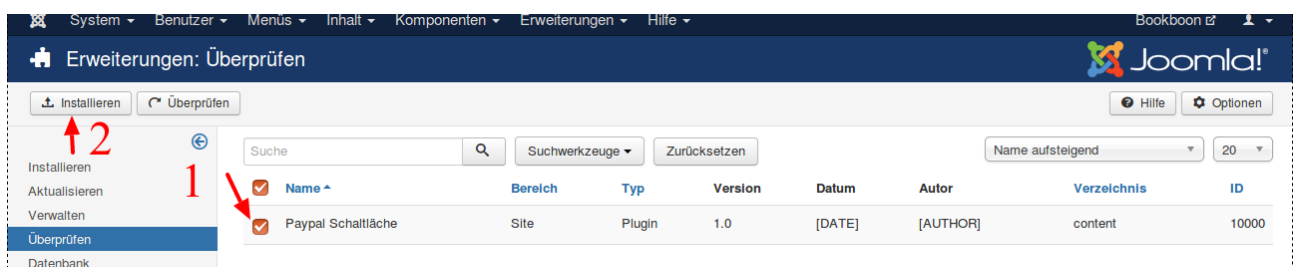


Abbildung 15: Wenn Sie die Plugin Dateien genauso wie von mir beschrieben erstellt haben, sehen Sie nun im Hauptbereich einen Eintrag, der Ihr neu erstelltes Plugin enthält. 984.png

Wenn alles richtig läuft, wird Ihnen nun gemeldet, dass das Installationspaket installiert wurde. Überprüfen Sie über das Menü Erweiterungen | Plugins ob Joomla! Ihr Plugin nun wirklich kennt und aktivieren Sie es im nächsten Schritt. Zum Aktivieren

des Plugins selektieren Sie die Auswahlbox links vor dem Plugin Eintrag und klicken dann in der Werkzeugleiste auf die Schaltfläche Aktivieren.



Abbildung 16: Der Bereich Plugins im Administrationsbereich wird über das Menü Erweiterungen | Plugins geöffnet. 983.png

Ihr Plugin ist nun aktiv und wenn Sie einen neuen Beitrag mit dem Text @paypalpaypal@ erstellen, erscheint im Frontend anstelle des Textes @paypalpaypal@ ein PayPal Jetzt-kaufen-Button. Probieren Sie es aus. Erstellen Sie als Erstes einen Beitrag, indem Sie im Administrationsbereich das Menü Inhalt | Beiträge | Neuer Beitrag öffnen.

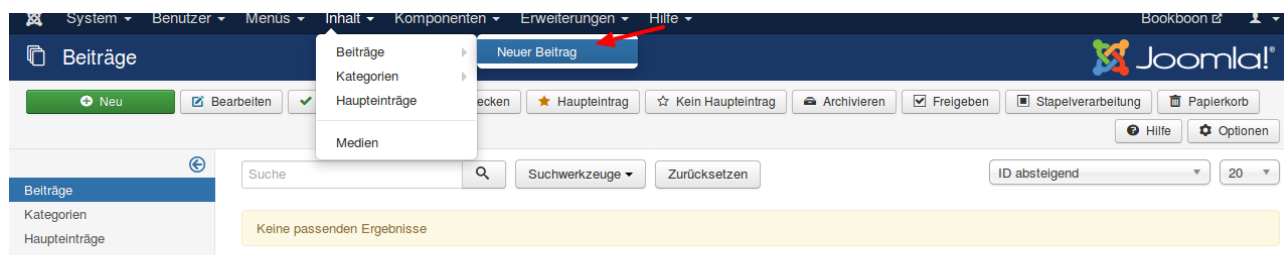


Abbildung 17: Der Bereich Beiträge im Administrationsbereich wird über das Menü Inhalt | Beiträge | Neuer Beitrag geöffnet. 982.png

Geben Sie hier nun einen Text ein, der das Muster "@paypalpaypal@" enthält. Geben Sie dem Beitrag einen Titel und setzen Sie den Parameter Haupteintrag auf Ja, damit der Beitrag als Haupteintrag auf der Startseite angezeigt wird. Zu guter Letzt speichern Sie den Beitrag.

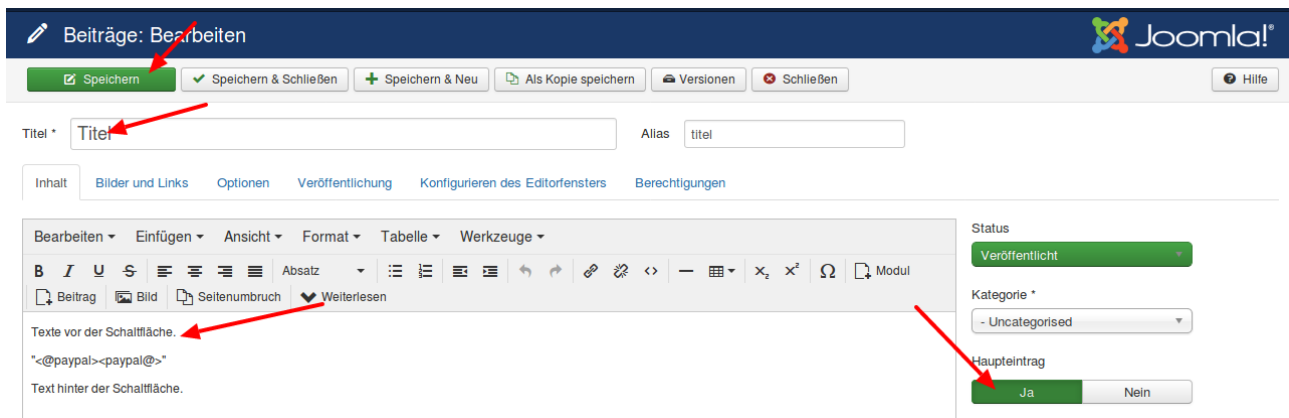


Abbildung 18: In Joomla! einen neuen Beitrag erstellen. 981.png

Öffnen Sie im Browser nun das Frontend – in unserer Beispielinstallation öffnen Sie das Frontend über die Adresse <http://localhost/joomla/>. Sehen Sie den PayPal Jetzt-kaufen-Button? Ich denke ja. Falls nicht sollten Sie den Programmcode Ihres Plugins noch einmal mit der von mir beschriebenen Datei vergleichen.

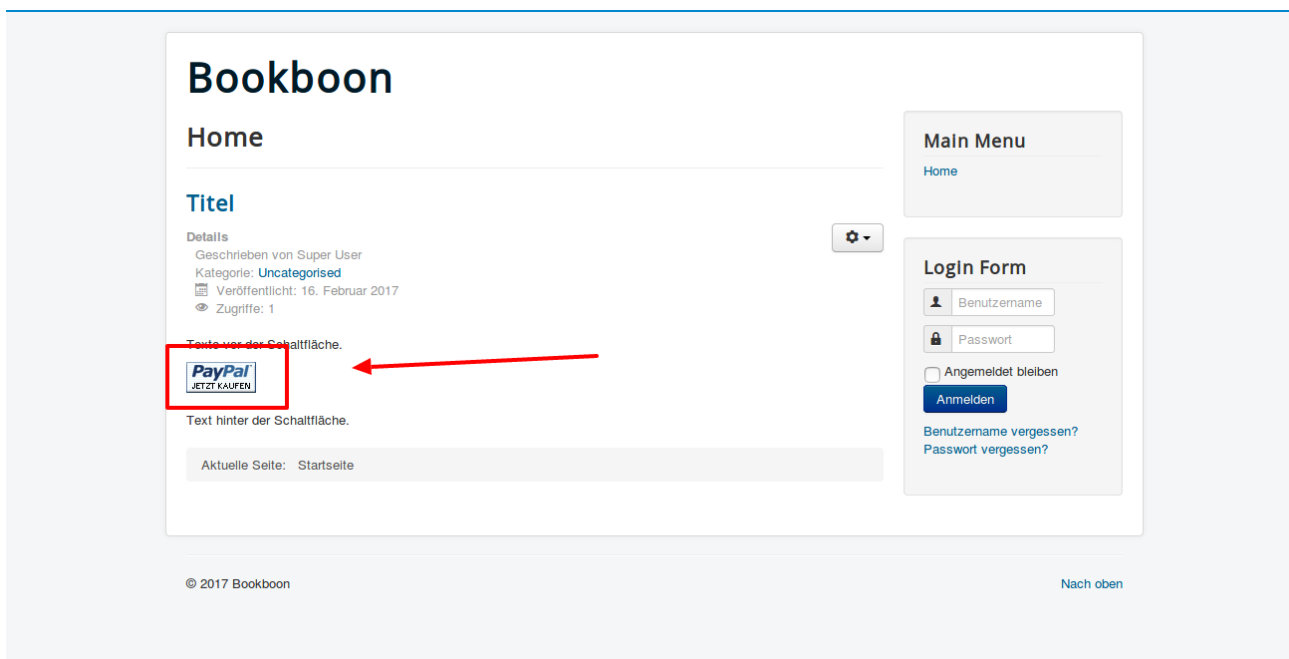


Abbildung 19: Die Ansicht des neu erstellen Beitrags im Frontend. 980.png

So, das wäre geschafft. Die Erweiterung funktioniert.

Unsere Tests mit Codeception planen

Die Spezifikation für das Plugin, das wir fertigstellen wollen, ist definiert. In Kurzform lautet diese: Ein Textmuster soll in einen PayPal Jetzt-kaufen-Button umgewandelt werden. Wie integrieren wir nun am besten welche Tests? Welche Möglichkeiten bietet uns Codeception?

Testtypen

Codeception unterstützt Sie beim Erstellen von

- **Unittests:**
Ein Unittest ist ein Test, der kleinste Programmeinheiten unabhängig voneinander testet.
- **Integrationstests oder Funktionstests:**
Ein Integrationstest ist ein Test, der das Zusammenspiel der einzelnen Einheiten testet. In der Codeception Terminologie heißt dieser Testtyp Funktionstest.
- **Akzeptanztests:**
Ein Akzeptanztest überprüft, ob das Programm seine, zu Beginn festgelegte, Aufgabe erfüllt.

Die Bausteine des Testsystems

Da zu Beginn eines Softwareprojektes noch nicht sicher ist, wie das Programm am Ende genau aussieht, fällt das Planen von Tests schwer. Man tappt sozusagen im Dunkeln. Aus diesem Grund ist es sinnvoll, das zu testende System in einzelne Bausteine zu unterteilen.

Für das Plugin, dass wir in diesem Buch als Beispieltestobjekt verwenden, könnten wir folgende Bausteine unterscheiden:

- Das Content Management System Joomla!, das überwiegend nach dem Entwurfsmuster [Model View Controller](#), kurz MVC, aufbaut ist und in das unser Plugin integriert ist.
- Unser Plugin selbst, als kleinste Einheit innerhalb der Programmsteuerung – also innerhalb eines Controllers.
- Das Webinterface – beziehungsweise die Schnittstelle zum Internetbrowser, über den der Benutzer mit dem System agiert.
- Die Datenbank, in der unter anderem die Beitragstexte, die unser Plugin umwandeln soll, gespeichert sind.
- Der Internetbrowser, über den der Benutzer die Webanwendung Joomla! aufruft.

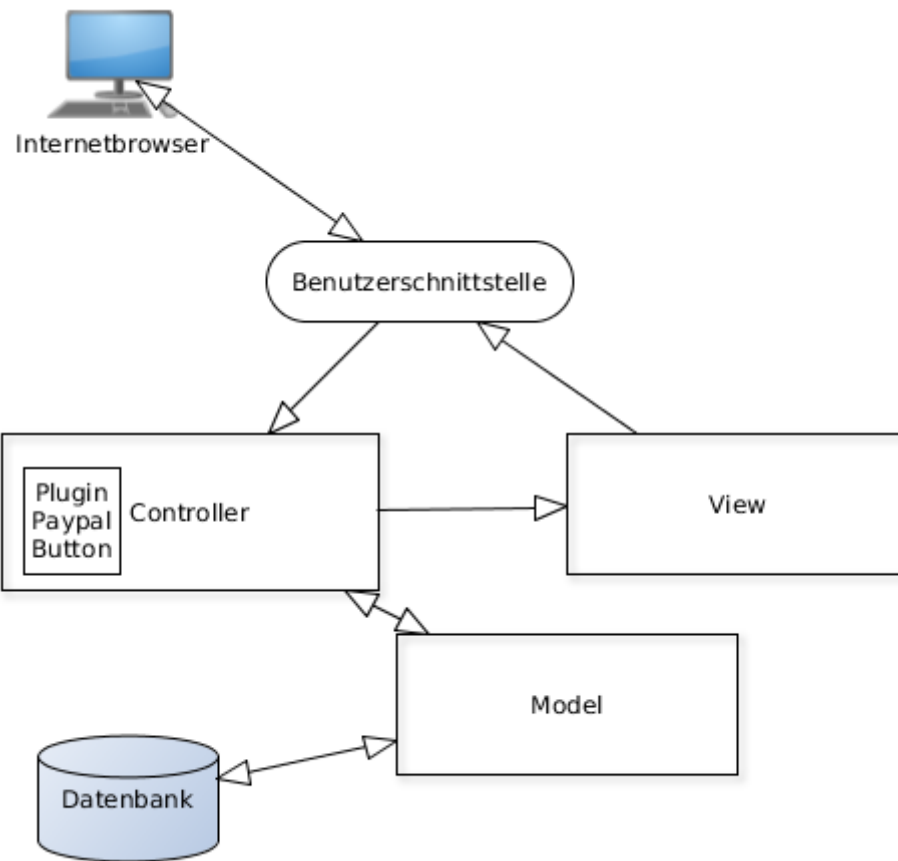


Abbildung 20: Eine Webanwendung mit Datenbank und Browerausgabe 990.png

Unittests testen einen Baustein

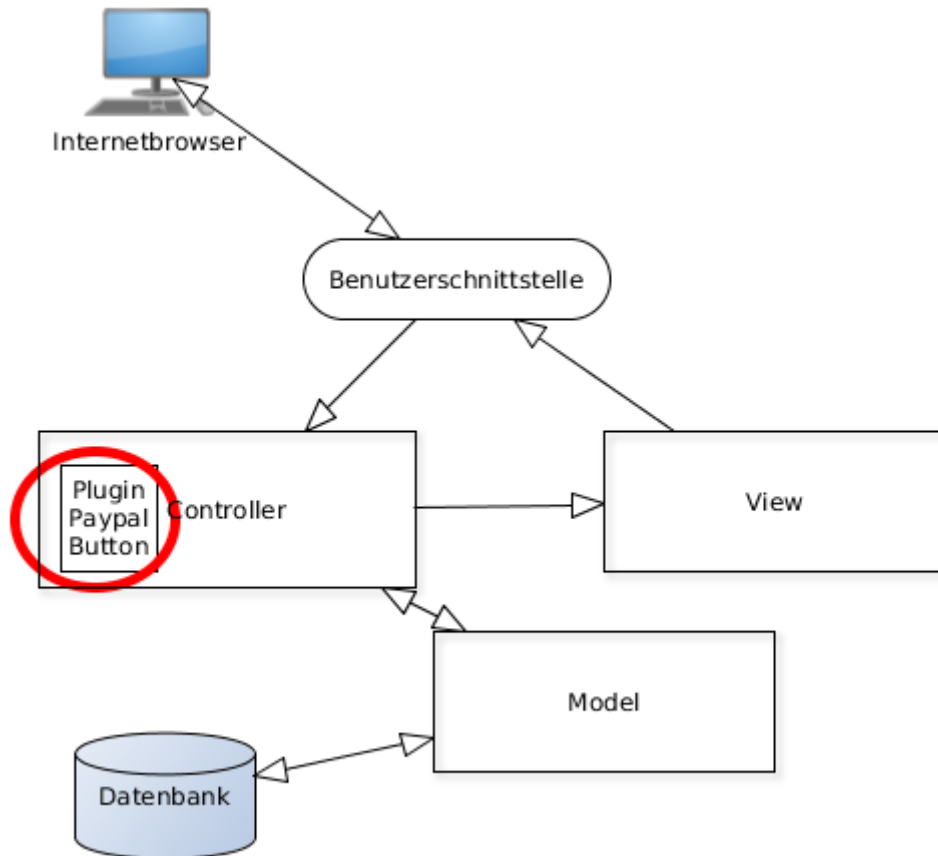


Abbildung 21: Ein Unittest testet eine kleine Einheit innerhalb eines Systems. 990aUnittest.png

Ein Unittest testet kleinste Programmeinheiten unabhängig voneinander. In unserem Beispiel testet ein Unittest die Plugin Klasse. Abhängigkeiten zu anderen Programmteilen müssen dabei aufgelöst werden. Im Beispiel prüfen wir beim Testen nicht, ob der Plugin Klasse auch der richtige Text zur Umwandlung übergeben wird – also, ob die anderen Klassen im System richtig arbeiten. Hierfür setzen wir einfach ein Testduplikat ein. Mit Testduplikat meine ich in diesem Fall einen extra für den Test erstellten Text.

Integrationstests testen das Zusammenspiel der Bausteine

Ein Integrationstest testet das Zusammenspiel der einzelnen Bausteine. Im nachfolgenden Bild sehen Sie einen Integrationstest, der das gesamte System zum Gegenstand hat. Wobei der Browser, mit dem der Endbenutzer arbeitet, ausgenommen ist. Um sicher zu stellen, dass unsere Webanwendung richtig arbeitet, reicht es aus, dass alle Daten richtig an die Schnittstelle zum Browser übergeben werden. Den Browser und die Funktionstüchtigkeit in der realen Umgebung testen wir mit einem Integrationstest nicht. Diese Bereiche sind nicht Teil unserer Anwendung.

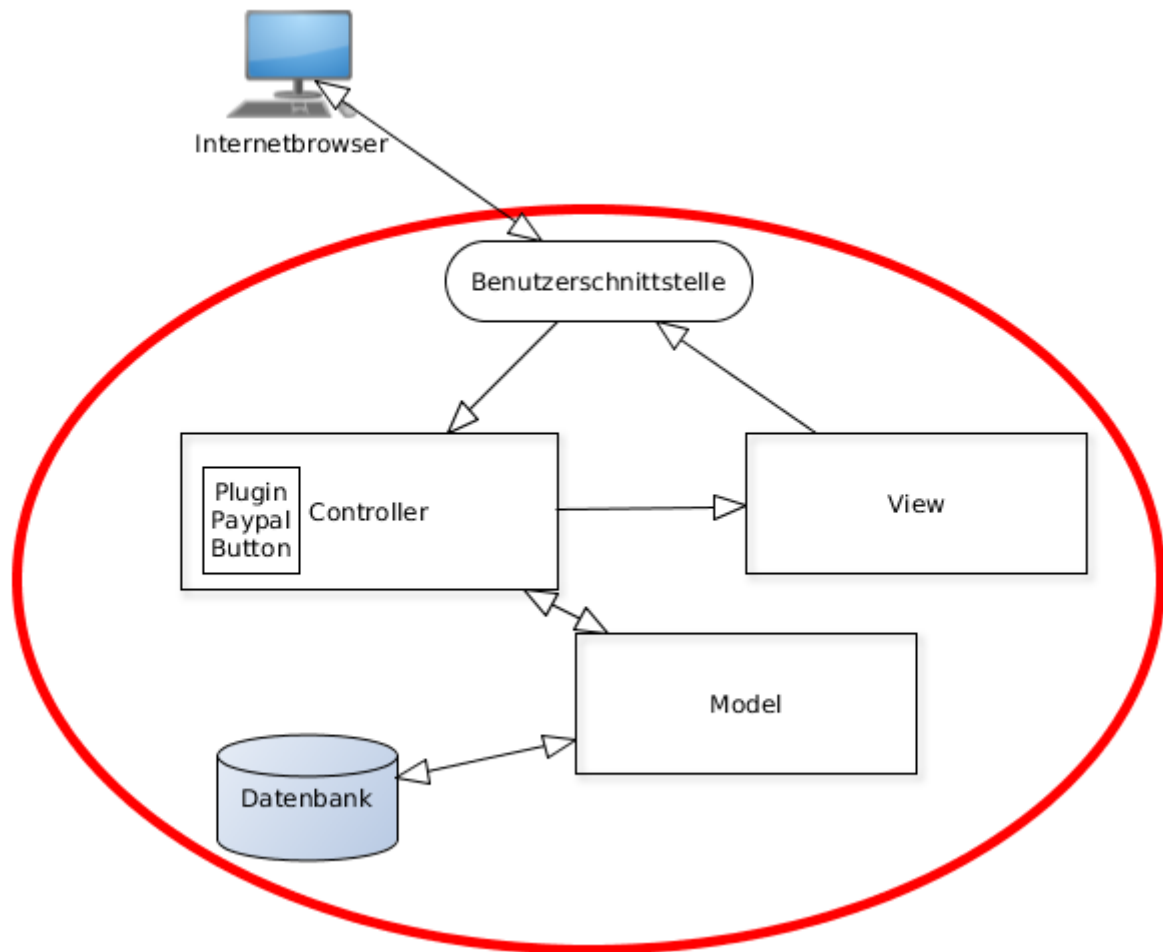


Abbildung 22: Funktionstests oder Integrationstests testen das Zusammenspiel der einzelnen Einheiten.

Akzeptanztests testen das System mit wirklichen Anwendungsfällen

Ein Akzeptanztest testet, ob das Programm seine zu Beginn festgelegte Aufgabe erfüllt. In unserem Beispiel werden nun also alle Bausteine mit in den Test einbezogen. Wir testen nicht nur, ob die Umwandlung unseres Suchtextes in einen PayPal Jetzt-kaufen-Button korrekt erfolgt. Bei einem Akzeptanztest ist es auch wichtig, dass der Button vom Benutzer wie gewollt verwendet werden kann.

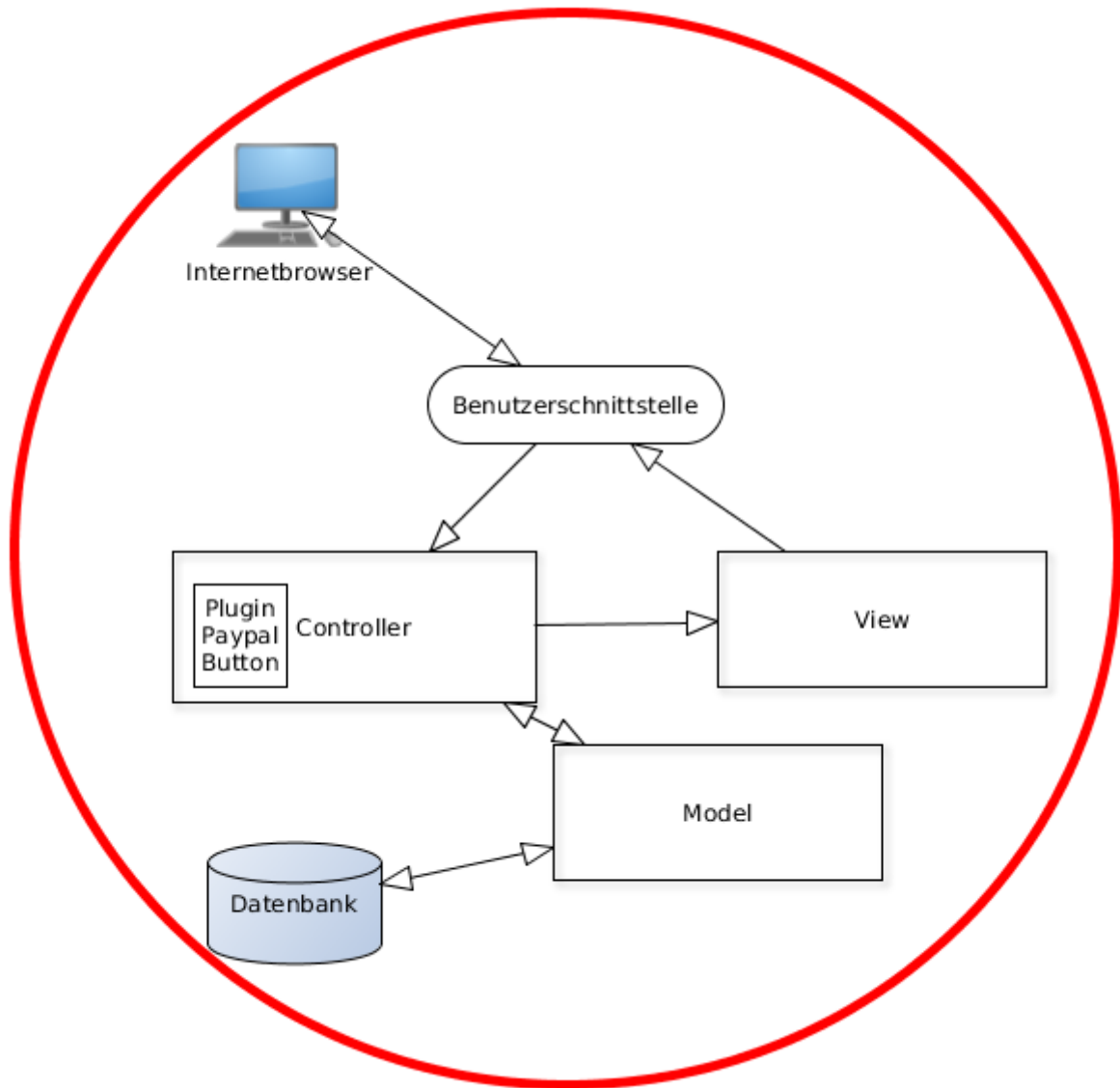


Abbildung 23: Ein Akzeptanztest testet wirkliche Anwendungsfälle anhand von automatisierten Benutzereingaben.

Teststrategie

Top-down-Testen und Bottom-up-Testen

Top-down und Bottom-up sind zwei grundsätzlich verschiedene Vorgehensweisen, um komplexe Sachverhalte zu verstehen und darzustellen. Dabei geht Top-down vom Abstrakten und Allgemeinen schrittweise hin zum Konkreten und Speziellen. Um dies an einem Beispiel zu verdeutlichen: Ganz allgemein stellt ein Content Management System wie Joomla! Websites in einem Browser dar. Konkret betrachtet gibt es aber bei diesem Vorgang eine Vielzahl von kleinen Unteraufgaben. Eine von diesen ist die Aufgabe, die unser Beispiel-Plugin übernimmt – nämlich die Darstellung eines Jetzt-kaufen-Button von PayPal. Bottom-up beschreibt die umgekehrte Richtung.

An dieser Stelle ist es wichtig, sich noch einmal in Erinnerung zu rufen, dass ein Element des Behaviour Driven Development die Erstellung einer textuellen Beschreibung des Verhaltens der Software ist. Diese Beschreibung der Akzeptanzkriterien erleichtert gleichzeitig die Erstellung von Tests – insbesondere das Erstellen der Akzeptanztests auf der obersten Ebene.

Die heute übliche Vorgehensweise beim Erstellen von Tests ist *von unten nach oben* – beziehungsweise von *innen nach außen*. Je weiter sich die verhaltensgetriebene Softwareentwicklung durchsetzt, desto mehr kommt diese Strategie in Wanken. Immer mehr Entwickler vertreten die Sichtweise, dass von *oben nach unten* – beziehungsweise von *außen nach innen* – getestet werden sollte. Grund hierfür ist: Nur bei einer Top-down Strategie kann ein Missverständnis in der Entwurfsphase frühzeitig erkannt werden.

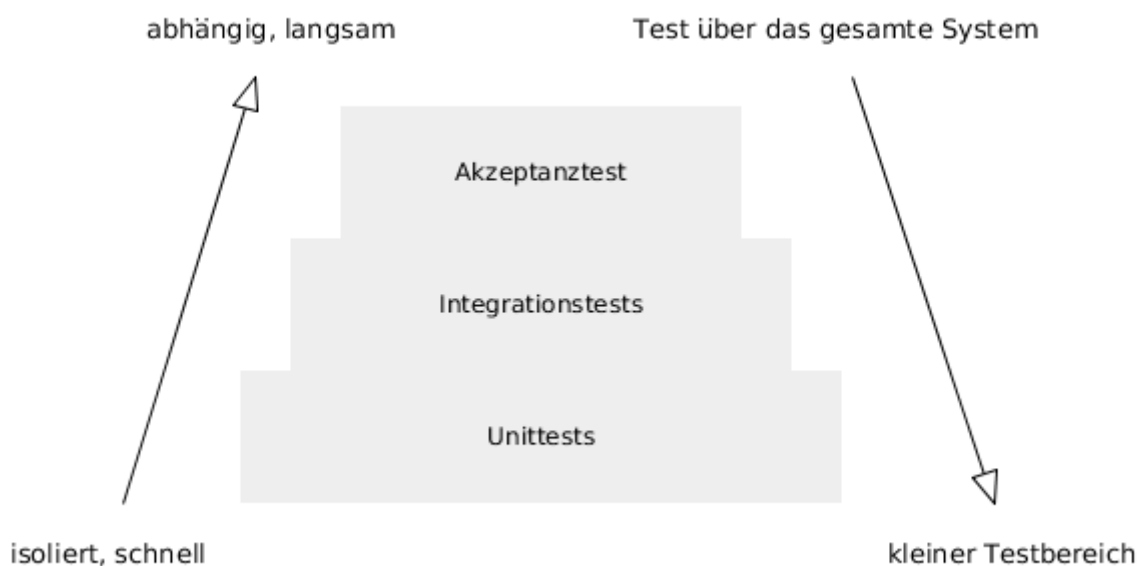


Abbildung 24: Teststrategien: Top-down-Testen und Bottom-up-Testen 989.png

Bottom-up-Testen

Verfolgt man die Bottom-up Strategie beginnt man mit den Unittests. Der Entwickler hat zu Beginn den Zielzustand vor Augen. Dieses Ziel zerlegt er dann aber zunächst in einzelne Komponenten. Das Problem beim Bottom-up-Ansatz ist, dass es schwierig ist zu testen, wie eine Komponente später verwendet wird. Der Vorteil von Bottom-up-Tests ist, dass wir sehr schnell fertige Softwareteile haben. Diese Teile sind aber mit Vorsicht zu genießen. Sie arbeiten zwar korrekt. Dies stellen die Unittests sicher. Ob

das Endergebnis aber wirklich so ist, wie der Kunde sich die Software vorstellt, ist nicht sichergestellt.

Top-down-Testen

Bei Anwendung der Top-down Strategie beginnt man mit den Akzeptanztests – also mit dem Teil des Systems, der am engsten mit den Benutzeranforderungen verknüpft ist. Bei Software, die für menschliche Benutzer geschrieben wurde, ist dies in der Regel die Benutzerschnittstelle. Schwerpunktmäßig wird getestet, wie ein Benutzer mit dem System interagiert. Ein Nachteil von Top-down-Tests ist, dass sehr viel Zeit für die Erstellung von Testduplikaten verwendet werden muss. Noch nicht integrierte Komponenten müssen durch Platzhalter ersetzt werden. Es gibt zu Beginn noch keinen echten Programmcode. Deshalb müssen fehlende Teile künstlich erstellt werden. Nach und nach werden diese künstlichen Daten dann durch wirklich berechnete Daten ersetzt.

Kurzgefasst

Sie haben in diesem Kapitel Ihre Entwicklungsumgebung am Beispiel des Content Management Systems Joomla! eingerichtet und eine eigene Erweiterung erstellt. Außerdem haben Sie sich Gedanken darüber gemacht, wie Sie diese Erweiterung idealerweise testen. Im nächsten Kapitel werden wir nun Codeception installieren, damit wir diese Tests in die Tat umsetzen können.

Codeception – ein Überblick

Testing is the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspects of the system or component. [[ANSI/IEEE Std. 610.12-1990](#), S. 76]

Wir möchten Tests für eine Joomla! Erweiterung mit [Codeception](#) schreiben. Dafür installieren wir Codeception im Joomla! Projekt. Alternativ könnte Codeception auch global für alle Anwendungen auf Ihrem Rechner installiert werden. Ich empfehle und zeige Ihnen hier aber die projektspezifische Installation. Um Codeception zu installieren, benötigen Sie den Paketmanager [Composer](#).

Ziel dieses Kapitels ist es, Codeception und Composer auf Ihrem Rechner zu installieren und Ihnen die notwendigen Grundlagen für die Nutzung dieser beiden Werkzeuge zur Verfügung zu stellen.

Composer

Wer oder was ist Composer und wofür wird Composer gebraucht? Um diese Frage zu beantworten, ist es wichtig, sich die folgende Tatsache in Erinnerung zu rufen: Es gibt eine unübersehbare Menge von PHP-Bibliotheken, Frameworks und Bausteinen. Wenn Sie schön länger mit PHP arbeiten, werden Sie sicherlich in Ihren Projekten die eine oder andere externe Software einsetzen. Ihr PHP-Projekt ist somit abhängig von einem anderen Projekt. Diese Abhängigkeiten mussten lange Zeit manuell verwaltet werden. Zusätzlich mussten Sie mithilfe von [Autoloading](#) sicherstellen, dass die verschiedenen Bausteine sich auch gegenseitig kennen. Mit Composer ist dieses zum Glück Vergangenheit.

Vielleicht kennen Sie **PEAR** und fragen Sie sich nun, ob dieser Paketmanager ein Pendant zu Composer ist. Nicht ganz. Verwenden Sie [PEAR](#) für die Verwaltung von Abhängigkeiten in einem **Gesamtsystem** wie Ihrem Computer und [Composer](#) für die Verwaltung von Abhängigkeiten in einem **einzelnen Projekt**.

Sie führen Composer Befehle über die Kommandozeile aus. In der Regel werden mithilfe von Composer andere PHP Programme, zu denen das aktuell zu installierende Pakete in einer Abhängigkeitsbeziehung steht, automatisch installiert. Welche PHP Anwendungen über Composer verfügbar sind, können Sie über die Plattform [Packagist](#) herausfinden. Composer ist noch recht neu. Die erste Version wurde im März 2012 veröffentlicht.

Installation

Wie gesagt: Sie können Composer lokal in Ihrem aktuellen Joomla!-Projekt installieren – oder global, zum Beispiel im Verzeichnis `/usr/local/bin`. Dieses Verzeichnis ist von Haus aus in der Umgebungsvariable `$PATH` hinterlegt und wird bei einem Programmaufruf nach dem entsprechenden Programm durchsucht.

Ich habe Composer unter Ubuntu 16.04 im Verzeichnis `/usr/local/bin` installiert. Dazu habe ich die Datei `composer.phar` mit dem Befehl `wget https://getcomposer.org/composer.phar` heruntergeladen.

```
$ wget https://getcomposer.org/composer.phar
```

RANDBEMERKUNG:

Falls `wget` noch nicht auf Ihrem Rechner installiert ist, können Sie das Programm mit dem Befehl `sudo apt-get install wget` installieren.

Danach habe ich die Datei `composer.phar` in `composer` umbenannt und ausführbar gemacht. Die dazu notwendigen Befehle stehen im nächsten Kasten.

```
$ mv composer.phar composer
```

```
$ chmod +x composer
```

Nun kann ich `composer` lokal, also in dem Verzeichnis, in dem die Datei abgelegt ist, ausführen. Um auch global auf den Paketmanager zugreifen zu können, habe ich die Datei mit dem Befehl `sudo mv composer /usr/local/bin` in das Verzeichnis `/usr/local/bin` verschoben.

```
$ sudo mv composer /usr/local/bin
```

Composer ist nun auf meinem Rechner überall verfügbar. Probieren Sie es aus. Die Eingabe des Befehls `composer` zeigt Ihnen, egal in welchem Verzeichnis Sie sich gerade befinden, eine Liste mit allen möglichen Befehlen an.

```
$ composer
Composer version 1.2.4 2016-12-06 22:00:51
Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
  ...
```

RANDBEMERKUNG:

Composer selbst erklärt auf der eigenen Website die [Installation](#) für unterschiedliche Plattformen.

Die Dateien Composer.json und Composer.lock

Wenn sie den Befehl `composer install` ausführen, sucht der Paketmanager Composer in dem Verzeichnis, in dem Sie den Befehl aufgerufen haben, nach einer Datei die `composer.json` heißt. Falls es diese Datei nicht gibt, sieht die Ausgabe wie folgt aus. Ich habe mich in meinem [Homeverzeichnis](#) `/home/meinName` befunden, als ich den Befehl abgesetzt habe. In diesem Verzeichnis gibt es keine Datei mit dem Namen `composer.json`.

```
~$ composer install
Composer could not find a composer.json file in /home/meinName
To initialize a project, please create a composer.json file as described in the https://getcomposer.org/
"Getting Started" section
```

Im nächsten Kapitel werden wir die Datei `composer.json`, die die Installation von Codeception bewirkt, anlegen. Sie könnten diese Datei selbst manuell erstellen. Der Inhalt der Datei sollte wie im folgenden Codeschnipsel aussehen.

```
{
  "require": {
    "codeception/codeception": "*"
  }
}
```

Sie schützen sich vor Problemen aufgrund von Tippfehlern, wenn Sie den folgenden Befehl verwenden. Dieser Befehl erstellt die Datei `composer.json` automatisch in der richtigen Syntax.

```
composer require codeception/codeception
```

Praktisch werden wir diesen Befehl im nächsten Kapitel anhand der Installation von Codeception anwenden. Im nächsten Kapitel werden Sie dann auch praktisch sehen was passiert, wenn Sie den Befehl `composer install` in einem Verzeichnis, in dem eine Datei `composer.json` vorhanden ist, ausführen. Nur so viel vorab: In diesem Fall werden alle Pakete, die zur Installation der in der Datei `composer.json` enthaltenen Programme notwendig sind, heruntergeladen und im Unterverzeichnis `/vendor` installiert. Außerdem wird die Datei `composer.lock` im Stammverzeichnis, hier ist dies gleichzeitig das Stammverzeichnis von Joomla!, angelegt. Die Datei `composer.lock` dokumentiert die heruntergeladenen Versionsstände der einzelnen Pakete. Wenn Sie Ihr Projekt mit anderen teilen möchten, können Sie mithilfe der Datei `composer.lock` immer sicherstellen, dass alle Projektbeteiligten mit den gleichen Versionsständen arbeiten. Alle im Projekt enthaltenen Pakete werden immer in der Version, die in der Datei `composer.lock` festgehalten ist, zum Projekt hinzugefügt. Und das auch dann, wenn ein Paket in der Zwischenzeit vom Entwickler in einer aktuelleren Version angeboten wird.

RANDBEMERKUNG:

Wenn Sie in Ihrem eigenen Projekt die neuere Version eines zwischenzeitlich aktualisierten Paketes einsetzen möchten, laden Sie diese neuere Version zunächst in Ihr lokales Projektverzeichnis. Sehen Sie sich dann in Ruhe an, ob die Änderungen im aktualisierten Paket negative Auswirkungen auf Ihr Projekt haben. Falls dies nicht so ist, können Sie die Versionsnummer in der `composer.lock` und der Datei `composer.json` herauf setzen. Dies bewirkt, dass in Ihrem Projekt zukünftig die neuere Version automatisch über Composer installiert wird.

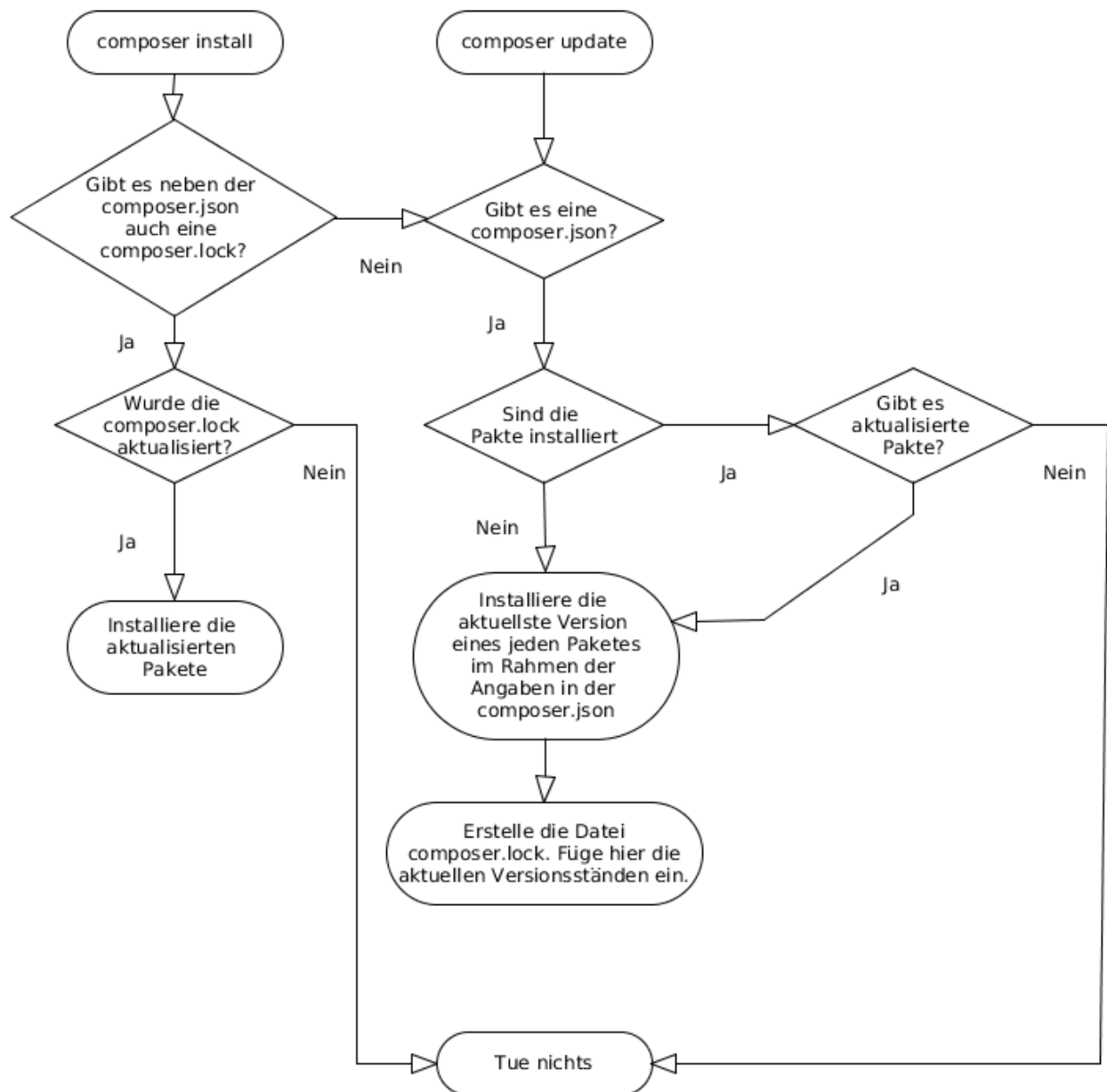


Abbildung 25: Die Dateien Composer.json und Composer.lock im Zusammenspiel mit den Befehlen composer install und composer update 960.png

Paketverwaltung

Die Datei composer.json, die ich im letzten Kapitel als Beispiel genannt habe, „hat es sich mit der Versionsnummer sehr einfach gemacht“. Ein Sternchen – also das Symbol * – fügt die neueste mögliche Version des Paketes zu Ihrem Projekt hinzu.

```

{
  "require": {
    "codeception/codeception": "*"
  }
}

```



```
| }
| }
```

Sie können die Version eines anzufordernden Pakets genauer eingrenzen. Neben dem [Sternchen \(*, englisch Wildcard\)](#), gibt es noch die [Tilde \(~\)](#), den [Zirkumflex \(^, englisch Caret\)](#) und die [Vergleichszeichen \(englisch Hyphen/Range\)](#). Die genaue Bedeutung der Zeichen ist leichter mit Beispielen zu erklären. In der nachfolgenden Tabelle und in der [Dokumentation von Composer](#) finden Sie Beispiele für unterschiedliche Anwendungsfälle.

Format	Trifft zu auf	Beispiel
<code>^1.0</code>	<code>>= 1.0.0 < 2.0.0</code>	1.0.0, 1.4.1, 1.9.9
<code>^1.1.0</code>	<code>>= 1.1.0 < 2.0.0</code>	1.1.0, 1.5.6, 1.9.9
<code>^1.2.3</code>	<code>>= 1.2.3 < 2.0.0</code>	1.2.3, 1.4.5, 1.9.9
<code>~1.0</code>	<code>>= 1.0 < 2.0</code>	1.0, 1.4, 1.9
<code>~1.1.0</code>	<code>>= 1.1.0 < 1.2.0</code>	1.1.0, 1.1.4, 1.1.9
<code>~1.2.3</code>	<code>>= 1.2.3 < 1.3.0</code>	1.2.3, 1.2.5, 1.2.9
<code>1.2.1</code>	<code>1.2.1</code>	1.2.1
<code>1.*</code>	<code>>= 1.0.0 < 2.0.0</code>	1.0.0, 1.4.5, 1.9.9
<code>1.2.*</code>	<code>>= 1.2.0 < 1.3.0</code>	1.2.0, 1.2.3, 1.2.9

Der Unterschied zwischen Tilde und Zirkumflex ist nicht auf den ersten Blick offensichtlich. Die mögliche Version bei Verwendung der Tilde ist abhängig von der Anzahl der Ziffern in der Versionsnummer. Bei Verwendung der Tilde kann nur die letzte Ziffer variieren.

Codeception

Nachdem Sie nun die Grundlagen von Composer kennen, können wir die Installation von Codeception in Angriff nehmen.

Installation

In diesem Kapitel zeige ich Ihnen, wie Sie Codeception installieren können. Grundlage ist die Vorgehensweise, die Codeception selbst auf der eigenen Website [vorschlägt](#).

Besorgen wir uns also zunächst einmal mit dem Befehl `wget`

`http://codeception.com/codecept.phar` die Datei `codecept.phar`. Wechseln Sie vor der Eingabe des

Befehls in das Verzeichnis, in dem Sie Codeception installieren möchten. In unserem Falle ist dies `/var/www/html/joomla`.

```
/var/www/html/joomla$ wget http://codeception.com/codecept.phar
--2017-02-17 22:06:39-- http://codeception.com/codecept.phar
Auflösen des Hostnamen »codeception.com (codeception.com)«... 192.30.252.154, 192.30.252.153
Verbindungsaufbau zu codeception.com (codeception.com)|192.30.252.154|:80... verbunden.
HTTP-Anforderung gesendet, warte auf Antwort... 200 OK
Länge: 10345974 (9,9M) [application/octet-stream]
In »»codecept.phar«« speichern.
codecept.phar 100%[=====>] 9,87M 122KB/s in 56s
2017-02-17 22:07:36 (181 KB/s) - »codecept.phar« gespeichert [10345974/10345974]
```

Mit dem Befehl `php codecept.phar bootstrap` installiert Codeception sich quasi selbst. Probieren Sie es aus.

```
/var/www/html/joomla$ php codecept.phar bootstrap
Initializing Codeception in /var/www/html/joomla
File codeception.yml created <- global configuration
tests/unit created <- unit tests
tests/unit.suite.yml written <- unit tests suite configuration
tests/functional created <- functional tests
tests/functional.suite.yml written <- functional tests suite configuration
tests/acceptance created <- acceptance tests
tests/acceptance.suite.yml written <- acceptance tests suite configuration
---
tests/_bootstrap.php written <- global bootstrap file
Building initial Tester classes
Building Actor classes for suites: unit, acceptance, functional
-> UnitTesterActions.php generated successfully. 0 methods added
\UnitTester includes modules: Asserts, \Helper\Unit
UnitTester.php created.
-> AcceptanceTesterActions.php generated successfully. 0 methods added
\AcceptanceTester includes modules: PhpBrowser, \Helper\Acceptance
AcceptanceTester.php created.
-> FunctionalTesterActions.php generated successfully. 0 methods added
```

\FunctionalTester includes modules: \Helper\Functional

FunctionalTester.php created.

Bootstrap is done. Check out /var/www/html/joomla/tests directory

Wenn alles richtig gelaufen ist, sehen Sie nun ein zusätzliches Verzeichnis in Ihrem Projekt. Dieses Verzeichnis heißt tests.

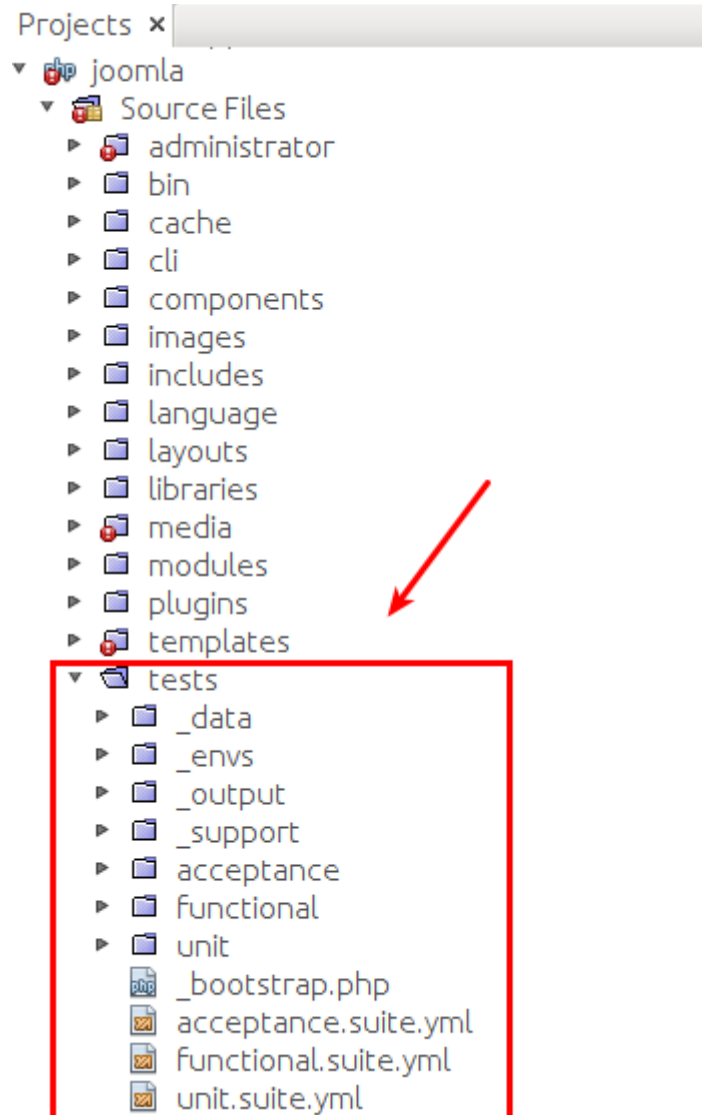


Abbildung 26: Nach der Installation finden Sie Codeception im Verzeichnis /tests.
977_tests.png

Das Grundgerüst von Codeception ist aufgebaut. Besorgen wir uns nun alle Pakete, die in einem Abhängigkeitsverhältnis zu Codeception stehen. Geben Sie dazu den

Befehl `composer require codeception/codeception` ein. Was dieser genau bewirkt, habe ich im vorausgehenden Kapitel erklärt.

```
/var/www/html/joomla$ composer require codeception/codeception
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing symfony/yaml (v3.2.4)
  Downloading: 100%
- Installing symfony/finder (v3.2.4)
  Loading from cache
- Installing symfony/event-dispatcher (v3.2.4)
  Loading from cache
- Installing symfony/polyfill-mbstring (v1.3.0)
  Loading from cache
- Installing symfony/dom-crawler (v3.2.4)
  Loading from cache
- Installing symfony/css-selector (v3.2.4)
  Loading from cache
...
- Installing facebook/webdriver (1.3.0)
  Loading from cache
- Installing behat/gherkin (v4.4.5)
  Loading from cache
- Installing codeception/codeception (2.2.9)
  Loading from cache
symfony/event-dispatcher suggests installing symfony/dependency-injection ()
symfony/event-dispatcher suggests installing symfony/http-kernel ()
symfony/console suggests installing symfony/filesystem ()
sebastian/global-state suggests installing ext-uopz (*)
phpunit/phpunit-mock-objects suggests installing ext-soap (*)
phpunit/phpunit suggests installing phpunit/php-invoker (~1.1)
facebook/webdriver suggests installing phpdocumentor/phpdocumentor (2.*)
codeception/codeception suggests installing codeception/specify (BDD-style code blocks)
codeception/codeception suggests installing codeception/verify (BDD-style assertions)
codeception/codeception suggests installing flow/jsonpath (For using JSONPath in REST module)
codeception/codeception suggests installing phpseclib/phpseclib (for SFTP option in FTP Module)
codeception/codeception suggests installing league/factory-muffin (For DataFactory module)
```

```
codeception/codeception suggests installing league/factory-muffin-faker (For Faker support in  
DataFactory module)  
codeception/codeception suggests installing symfony/phpunit-bridge (For phpunit-bridge support)  
Writing lock file  
Generating autoload files
```

Das war Ihnen sicherlich soweit klar. Ich schreibe es aber der Vollständigkeit halber trotzdem noch einmal. Im Projektordner hat Composer nun das Verzeichnis `/vendor` angelegt. In diesem Verzeichnis finden Sie alle Softwarepakete, die Codeception benötigt.

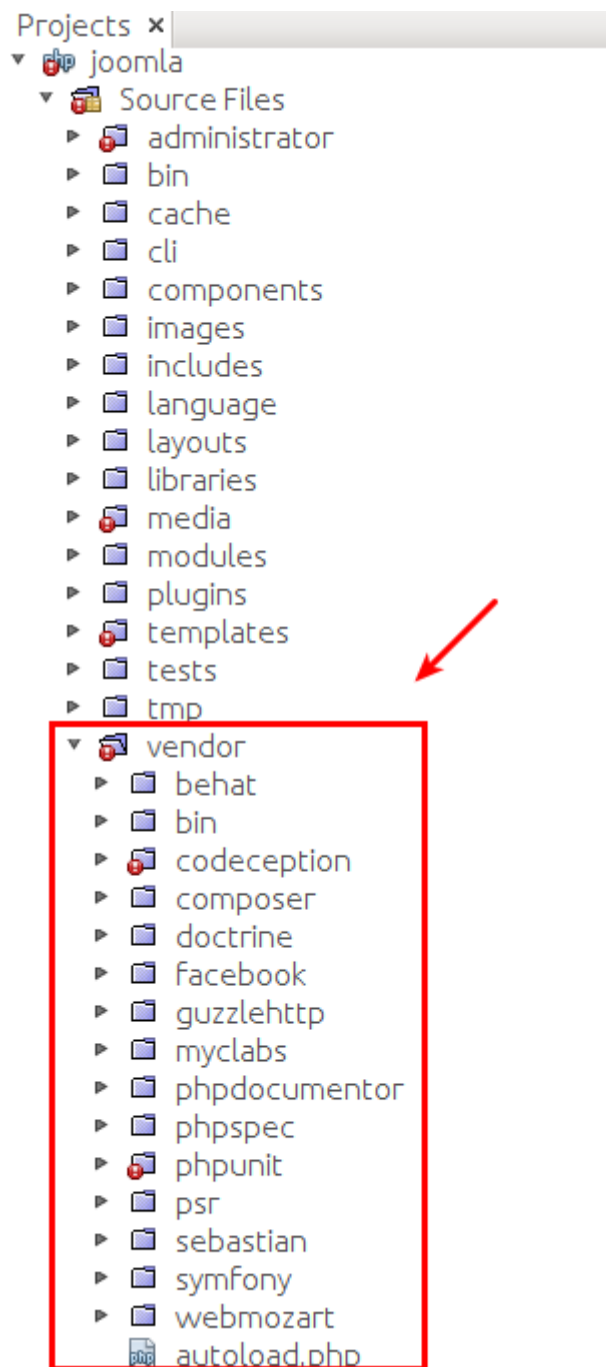


Abbildung 27: Im Verzeichnis /vendor finden Sie alle Softwarepakete, die Codeception benötigt. 978_vendor.png

Sie können nun einen Testlauf starten. Wir haben bisher zwar noch keine Tests erstellt. Es kann also auch nichts wirklich getestet werden. Starten Sie trotzdem einmal mit dem Befehl `vendor/bin/codecept run unit` einen Testlauf. Wenn Ihnen daraufhin kein Fehler gemeldet wird, können Sie sicher sein, dass Codeception richtig installiert und konfiguriert ist.

```
/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 (0) -----
-----
Time: 95 ms, Memory: 8.00MB
No tests executed!
```

Codeception läuft fehlerfrei. Wir können den ersten Test erstellen. Am einfachsten erledigen Sie dies mithilfe des Testgenerators. Geben Sie dazu den Befehl `vendor/bin/codecept generate:test unit` in der Konsole ein. Sie werden daraufhin die Datei `agpaypalTest.php` im Verzeichnis `/var/www/html/joomla/tests/unit/suites/plugins/content/agpaypal` vorfinden.

```
/var/www/html/joomla$ vendor/bin/codecept generate:test unit
/suites/plugins/content/agpaypal/agpaypal
Test was created in
/var/www/html/joomla/tests/unit/suites/plugins/content/agpaypal/agpaypalTest.php
```

In der automatisch generierten Version enthält die Datei ausschließlich Methoden, die nicht mit Programmcode gefüllt sind. Codeception kann Ihnen nur die Routinearbeiten abnehmen, indem es das Codefragmente (das [Boilerplate](#)) erstellt. Im weiteren Verlauf dieses Buches werden wir diese Methoden mit Inhalt füllen.

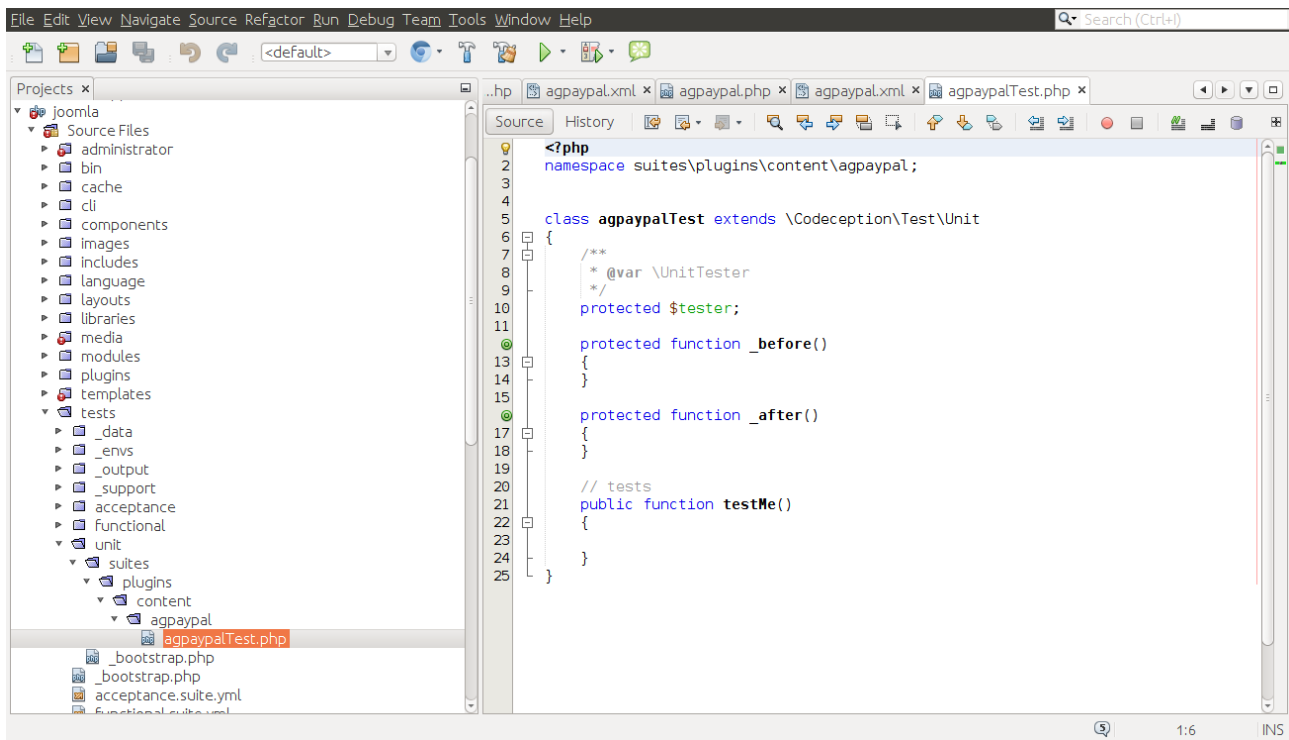


Abbildung 28: In der automatisch generierten Version enthält die Datei ausschließlich Methoden, die nicht mit Programmcode gefüllt sind. Sie sehen die Datei hier in der Entwicklungsumgebung Netbeans geöffnet. 979.png

RANDBEMERKUNG:

Die Testklassen, im Beispiel hier die Klasse `agpaypalTest`, werden bewusst getrennt von den Produktionsklassen `plgContentAgpaypal` in einem separaten Verzeichnis abgelegt.

Codeception – ein erster Rundgang

Nachdem nun alles installiert und lauffähig ist, werden wir uns mit der Implementierung der ersten realen Tests befassen. Da eine gute Vorbereitung die halbe Miete ist, sehen wir uns zunächst das, was Codeception uns bietet, genauer an. Ich führe Sie als Erstes durch die Verzeichnisstruktur von Codeception. Wenn Sie sich in dem Programm zurechtfinden, ist der zweite Teil ein Kinderspiel. Sie wissen dann, wie Codeception intern Daten verarbeitet, welche Erweiterungen es gibt und kennen die grundlegende Syntax.

Was ist das Ziel von Codeception?

Falls Sie bereits Tests mit einem Testwerkzeug geschrieben haben, kennen Sie die Problematik sicherlich. Jedes Werkzeug hat seine Vorteile, seine Nachteile und seine Besonderheiten. Das eine Tool bietet Ihnen einen guten Support, dafür können Sie die

Tests nicht automatisieren. Bietet ein Testtool effizientes und komfortables Arbeiten, ist die Lernkurve vielleicht sehr steil – oder umgekehrt. Es gibt nur wenige Unternehmen, die intensiv Tests einsetzen. Diese Firmen haben in der Regel Wissen im Bereich Softwaretests selbst aufgebaut oder verfügen über die notwendigen finanziellen Mittel, um das Wissen einzukaufen. Mit Codeception können auch Freiberufler Softwaretests intensiv einsetzen.

Für die Programmiersprache PHP ist das bekannteste Werkzeug zweifelsfrei [PHPUnit](#). PHPUnit ist ein PHP-Framework, mit dem Sie PHP-Skripte testen können. Es eignet sich besonders für automatisierte Unittests. PHPUnit basiert auf Konzepten, die zuvor in dem Java Pendant [JUnit](#) umgesetzt wurden. Aber auch PHPUnit hat seine Grenzen. Wenn Sie Integrationstests oder Akzeptanztests schreiben möchten, werden Sie an diese Grenzen stoßen. Wie der Name schon vermuten lässt, ist das Framework für diese Aufgaben nicht gemacht. Hierfür gibt es andere Werkzeuge. Jedes Tool für sich hat seine Berechtigung. Als Anwender müssen Sie sich aber immer wieder für jedes Programm neu einrichten. Die Bedienung und die Konfiguration der Software, die Syntax und nicht zuletzt die Regeln beim Erstellen der Tests müssen neu gelernt werden. Codeception will hier Abhilfe schaffen.

RANDBEMERKUNG:

Möchten Sie sich einen Überblick über die verschiedenen Testframeworks – nur im Bereich Unittests – verschaffen? [Wikipedia](#) bietet eine umfangreiche Liste.

Codeception ist mehr als nur ein weiteres Werkzeug

Codeception ist kein weiteres Werkzeug. Warum sollte auch ein weiteres Werkzeug geschaffen werden? Es gibt ja genug Hilfsmittel. Am Ende würde ein Tester wieder vor dem gleichen Problem stehen: Er müsste unterschiedliche Programme nutzen, aufeinander abstimmen und erlernen. Denn auch ein neues Werkzeug wäre höchstwahrscheinlich nicht die Eierlegende Wollmilchsau die alles gleich gut kann. Deshalb bündelt Codeception verschiedene Werkzeuge. Es ist somit ein *Framework für Frameworks*.

Codeception bietet eine einheitliche Art Tests zu schreiben. Dabei unterstützt es unterschiedliche Testtypen – nämlich Unittests, Integrationstests/Funktionstests und Akzeptanztests. Die Logik und die Vorgehensweise beim Erstellen der unterschiedlichen Tests ist mithilfe von Codeception für alle Testtypen einheitlich.

Codeception stellt sich vor

Codeception schreibt auf der eigenen Website [über sich](#): Codeception wurde entwickelt, um ein Werkzeug zu bieten, das möglichst einfach zu bedienen ist. Schon die Installation ist einfach. Das haben Sie im vorausgehenden Kapitel selbst erfahren.

Codeception legt Wert darauf, dass alle Tests **leicht zu lesen** sind. Ziel ist es, dass alle Projektbeteiligten – auch diejenigen die nicht selbst programmieren – einen Test lesen können und den Testgegenstand verstehen.

Ich hatte weiter vorne schon geschrieben, dass bei der Verwendung unterschiedlicher Testtools die unterschiedliche Syntax ein Problem ist. Jedes Programm hat seine eigenen Syntax-Regeln. Codeception unterstützt Sie beim Erstellen von Tests durch die Vereinheitlichung der Syntax. Tests sollen **einfach zu schreiben** sein!

Ein weiteres Ziel von Codeception ist es, Fehler leichter auffindbar zu machen. Dafür ist es wichtig, jederzeit die aktuelle Variablenbelegung im Testcode ablesen zu können. Mit Codeception sind Tests **leicht zu debuggen**.

Weiterhin ist Codeception modular aufgebaut, so dass es **leicht zu erweitern** ist.

Zusätzlich macht Codeception die **Wiederverwendbarkeit** des Programmcodes einfach.

Sind Sie nun neugierig geworden? Dann tauchen wir doch tiefer in die Details ein.

Testtypen in Codeception

Ich habe im Kapitel *Praxisteil: Die Testumgebung einrichten | Unsere Tests mit Codeception planen | Testtypen* die wichtigsten Testtypen in Codeception, nämlich Unittests, Funktionstests (Integrationstests) und Akzeptanztests, erläutert.

Jeder Testtyp hat in Codeception seinen eigenen Ordner.

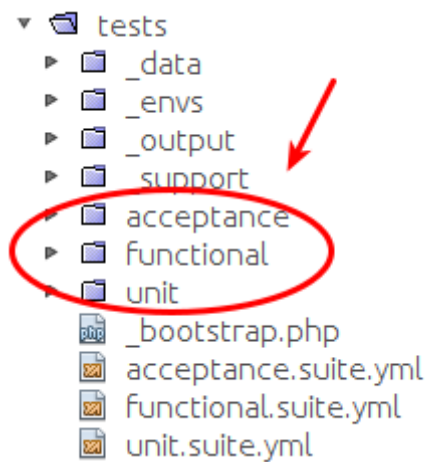


Abbildung 29: Verzeichnisse für die Testtypen innerhalb von Codeception. 976.png

Bevor Sie mit dem Schreiben von Tests beginnen, sollten Sie sicherstellen, dass alle notwendigen Codeception Klassen vorhanden sind. Anlegen können Sie diese Klassen jederzeit mithilfe des Befehls `vendor/bin/codecept build`. Geben Sie diesen Befehl nun ein.

```
/var/www/html/joomla$vendor/bin/codecept build
Building Actor classes for suites: unit, acceptance, functional
-> UnitTesterActions.php generated successfully. 0 methods added
\UnitTester includes modules: Asserts, \Helper\Unit
-> AcceptanceTesterActions.php generated successfully. 0 methods added
\AcceptanceTester includes modules: PhpBrowser, \Helper\Acceptance
-> FunctionalTesterActions.php generated successfully. 0 methods added
\FunctionalTester includes modules: \Helper\Functional
```

Was ist genau passiert? Unter anderem hat Codeception drei Dateien, nämlich die Dateien `UnitTesterActions.php`, `AcceptanceTesterActions.php` und `FunctionalTesterActions.php`, erstellt. Sehen Sie sich die Klassen in diesen Dateien genauer an. Sie sind prall gefüllt mit hilfreichen Funktionen. Diese Funktionen können Sie in Ihren Tests später verwenden. Sie finden die Dateien im Verzeichnis

`/var/www/html/joomla/tests/_support/_generated`.

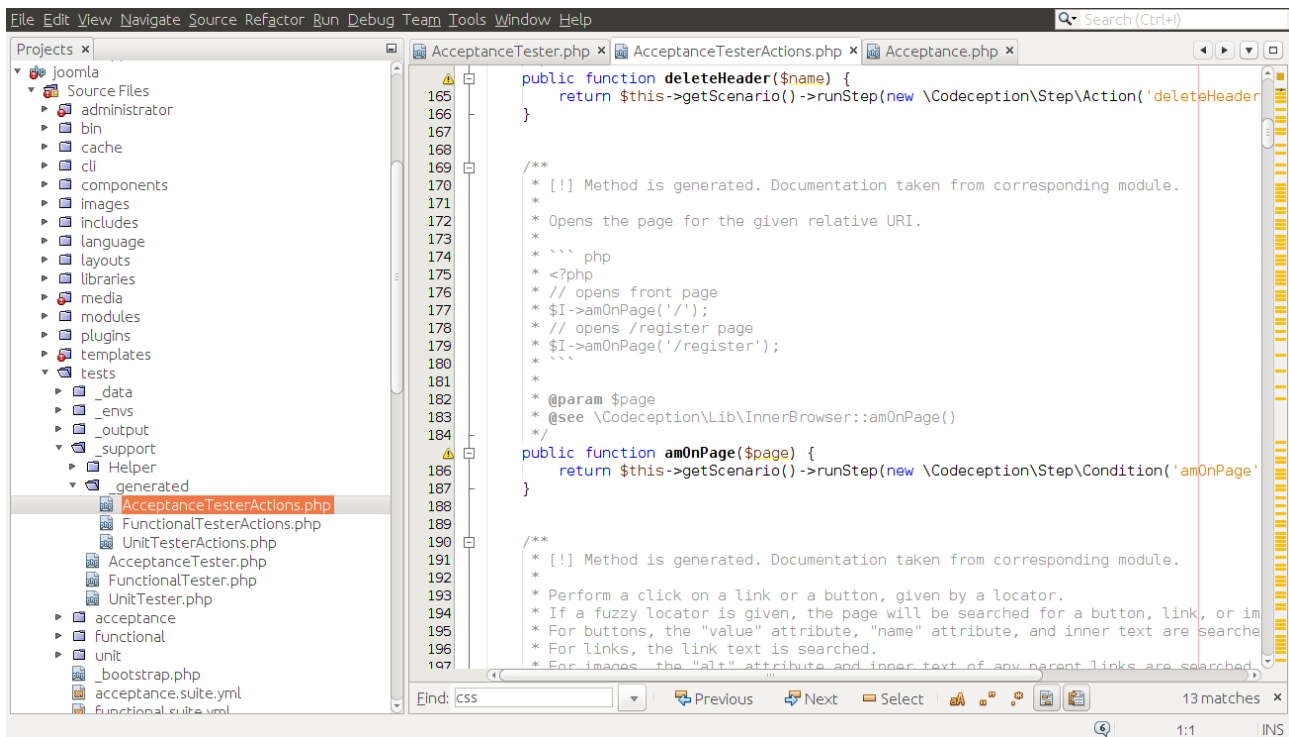


Abbildung 30: Die Klasse AcceptanceTesterAction.php unmittelbar nach der automatischen Generierung. 975.png

WICHTIG

Das Kommando `vendor/bin/codecept build` sollten Sie nach Änderungen in der Konfiguration von Codeception immer ausführen. Sie können sich merken: Immer dann, wenn Sie eine Datei mit der Endung `.suite.yml` innerhalb des Ordners `/tests` ändern, sollten Sie Codeception neu „builden“ – also neu aufbauen. Sie müssen den `build`-Befehl auf jeden Fall dann ausführen, wenn Sie aufgrund der Änderungen neue Module in Codeception laden und nutzen möchten.

Sind Sie beim Stöbern innerhalb von Codeception über die Klassen `UnitTester`, `AcceptanceTester` und `FunctionalTester` gestolpert? Falls nicht: Sie finden diese Klassen im Verzeichnis `/var/www/html/joomla/tests/_support/`. Ich musste schmunzeln, als ich diese Namen zum ersten Mal gelesen habe. Codeception erreicht mit dieser Namensgebung ganz nebenbei, dass man beim Schreiben der Tests einen wirklichen Tester, also einen Menschen, der das Programm ausführt, vor Augen hat. Die verhaltensgetriebene Softwareentwicklung (Behaviour Driven Development; BDD) wird so ideal ergänzt.

BDD hält während der Entwurfsphase die geforderten Funktionen der Software in Textform fest. Die Funktionen werden dabei meist in „Wenn-dann“-Sätzen beschrieben. Diese „Wenn-dann“-Sätze können als Tests programmiert werden. **Wenn** der **Tester** dies tut, **dann** sollte jenes passieren. So kann die Software auf ihre korrekte Implementierung getestet werden.

Getestet wird nicht nur, ob **das Programm richtig funktioniert**. Der Tester stellt auch sicher, dass das **richtige Programm erstellt wurde**. Ich wiederhole das an dieser Stelle, weil das von zentraler Bedeutung ist.

Nachfolgend stelle ich Ihnen die Tester kurz vor, bevor Sie diese in der Praxis kennenlernen.

AcceptanceTester

Unter dem Akzeptanztester können Sie sich eine Person vorstellen, die technisch nicht sehr interessiert ist. Der Akzeptanztester möchte von einem System unterstützt werden. Dabei soll alles so funktionieren, wie es in der Spezifikation festgelegt wurde.

RANDBEMERKUNG:

Zum Beispiel könnte ein Akzeptanzkriterium des Content Management Systems Joomla! sinngemäß so lauten: Wenn ich die Installationsdatei auf meinen Server kopiere und die Fragen der Installationsroutine richtig beantworte, dann kann ich auf den Administrationsbereich von Joomla! zugreifen und hier Einstellungen vornehmen. Sehen Sie sich das 24-sekündige [Video](#) zu diesem Szenario auf YouTube an. So bekommen Sie einen Eindruck und verstehen, warum ich an anderer Stelle schrieb, dass Akzeptanztests Spaß machen können.

Beim Behaviour Driven Development wurde die Spezifikation in der Entwurfsphase mittels Beispielen, sogenannten Szenarios, beschrieben. Üblicherweise wird für die Beschreibung dieser Szenarios ein bestimmtes Format verwendet. Eines dieser Formate, welches auch von Codeception unterstützt wird, ist die Beschreibungssprache [Gherkin](#). Sie können Gherkin sowohl mit den englischen Schlüsselwörtern Given, When, Then und And oder den deutschen Schlüsselwörtern Gegeben, Wenn, Dann und Und nutzen. Ich habe mich hier im Buch für die englischen entschieden. Jedes Szenario beschreibt das Verhalten der Software in einem Teilbereich. Im Englischen werden die Szenarien auch [User Storys](#) genannt.

Beispielsweise könnte die Anforderung „Website als Administrator verwalten“ mit folgenden Szenarios beschrieben werden:

Feature: administrator login

In order to manage my web application

As an administrator

I need to have a control panel

Scenario: Login in Administrator

Given There is an administrator control panel and I am on login page

When I enter correct credentials on login page

Then I should see the administrator control panel

Dieses Szenario könnte wie folgt implementiert werden.

```
$I->amOnPage(LoginPage::$url);  
$I->fillField(LoginPage::$usernameField, $username);  
$I->fillField(LoginPage::$passwordField, $password);  
$I->click(LoginPage::$loginButton);  
$I->see('Your are logged in');
```

Sie sehen, der Test – zumindest der in Gherkin geschriebene Teil – ist tatsächlich einfach zu lesen. Die Implementierung greift auf Methoden zurück, die in der Klasse `AcceptanceTesterActions` von Codeception generiert wurden. Diese Klasse hatten Sie sich zu Beginn dieses Kapitels angesehen. Im Kapitel *Akzeptanztests* werden wir Akzeptanztests praktisch ausprobieren.

RANDBEMERKUNG:

Falls Ihnen der Name `AcceptanceTester` nicht gefällt, können Sie diesen in der Konfigurationsdatei `acceptance.suite.yml` ändern. Sie finden diese Datei im Verzeichnis `/var/www/html/joomla/tests/`. Wenn Sie den Tester Kunde nennen möchten, dann ändern Sie einfach die erste Zeile in der Datei `acceptance.suite.yml`. Ändern Sie also `class_name: AcceptanceTester` in `class_name: Kunde`. Denken Sie daran, danach mit dem Befehl `vendor/bin/codecept build` den geänderten Namen in allen Codeception

Dateien zu aktualisieren. Die Namen FunctionalTester und UnitTester können Sie auf die gleiche Weise ändern.

Akzeptanztests sind Tests, die das Benutzerverhalten am realistischsten reproduzieren. Idealerweise laufen sie unter den gleichen Bedingungen wie das Produktivprogramm ab.

FunctionalTester

Die Bezeichnung Funktionstest ist eine Eigenart in Codeception. Im Grunde genommen sind Funktionstests das gleiche wie Integrationstests. Funktionstests laufen schneller als Akzeptanztests. Grund hierfür ist, dass nicht die gleichen Bedingungen wie beim Produktivsystem gefordert sind. Funktionstests für eine Webapplikation benötigen nicht zwingend einen Webserver. Diese Tests können auch mit einem Browser, der nicht über eine grafische Benutzeroberfläche verfügt und somit schneller abläuft, durchgeführt werden. Browser ohne grafische Benutzeroberfläche nennt man [Headless Browser](#).

Funktionstests nutzen unter anderem die einheitliche Schnittstelle des Architekturstils [Representational State Transfer \(REST\)](#). Was können Sie sich genau unter einer REST-Schnittstelle vorstellen? Systeme können über eine REST-Schnittstelle automatisch miteinander kommunizieren. Das bedeutet, dass auch Tests automatisch ablaufen können. Im Internet ist ein Großteil der für REST nötigen Infrastruktur bereits vorhanden. So ist eine [Webanwendung](#), die statische Seiteninhalte über das Hypertext-Übertragungsprotokoll [HTTP](#) anbietet, REST-konform. Ein Content Management System wie Joomla! erzeugt seine Inhalte dynamisch. Die gleiche [HTTP-Anfrage](#) - also der gleiche HTTP-Request - kann, je nach Datenbankinhalt, unterschiedliche Informationen anzeigen. Joomla! erfüllt das REST Paradigma somit nicht vollständig. Zur Veranschaulichung erstelle ich mit Ihnen im praktischen Teil – im Unterkapitel *REST-Schnittstelle* des Kapitels *Funktionstest* – einen Funktionstest, der HTTP-Zugriffe nutzt.

Wird über das Hypertext-Übertragungsprotokoll HTTP auf eine Anwendung zugegriffen, so gibt die verwendete [HTTP-Methode](#) an, welche Operation des Dienstes gewünscht ist. Die wichtigsten HTTP-Methoden sind

- GET:
Mit der GET-Methode können Sie eine Ressource, zum Beispiel eine Datei, vom Server anfordern.
- POST:
Mit der POST-Methode können Sie Daten wie Bilder oder HTML-Formular-Inhalte zur weiteren Verarbeitung zu einem Server senden.
- PUT:
Die PUT-Methode dient dazu eine Ressource, zum Beispiel eine Datei, unter Angabe der Zieladresse auf einen Webserver zu kopieren. Dabei muss die Ressource nicht wie bei POST durch ein Skript verarbeitet werden, sondern wird an einer festgelegten Stelle platziert.
- DELETE:
Die DELETE-Methode löscht die angegebene Ressource auf dem Server.

UnitTester

Codeception nutzt das Framework PHPUnit für die Erstellung der Unittests. Das ein Unittest kleinste Programmeinheiten unabhängig voneinander testet, hatte ich bereits erwähnt. An dieser Stelle gibt es nicht viel mehr dazu zu sagen. Wenn Sie PHPUnit schon verwenden, können Sie das nächste Kapitel rasch durcharbeiten. Andernfalls müssen Sie vielleicht das ein oder andere Mal etwas in der [Dokumentation von PHPUnit](#) nachlesen.

Mit Codeception Tests organisieren und erweitern

CEPT und CEST

Codeception selbst bietet Ihnen zwei verschiedene Formate. Das [CEPT-Format](#) ist ein [Szenario basiertes](#) Format. Das [CEST-Format](#) basiert auf einer [Klasse](#).

Mithilfe des Befehls `vendor/bin/codecept generate:cest acceptance TestCest` erstellen Sie eine **CEST**-Datei – hier im Beispiel für einen Akzeptanztest.

```
/var/www/html/joomla$ vendor/bin/codecept generate:cest acceptance TestCest
Test was created in /var/www/html/joomla/tests/acceptance/TestCest.php
```


Diese neu erstellte Datei beinhaltet zunächst nur Methoden, die keinen Programmcode enthalten.

```
<?php
class TestCest{
    public function _before(AcceptanceTester $I) { }
    public function _after(AcceptanceTester $I) { }
    // tests
    public function tryToTest(AcceptanceTester $I) { }
}
```

Jede Methode in einer CEST-Datei, deren Name nicht mit einem Unterstrich beginnt, wird von Codeception als Test interpretiert. Die Methoden `_before()` und `_after()` sind spezielle Methoden. `_before()` wird vor jedem Test und `_after()` wird nach jedem Test ausgeführt. Zusätzlich können Sie noch die spezielle Methode `_fail()` verwenden. Sie wird im Falle eines Fehlers ausgeführt und eignet sich deshalb zum Aufräumen im nach einem Fehler.

Mithilfe des Befehls `vendor/bin/codecept generate:cept acceptance TestCest` erstellen Sie eine **CEPT**-Datei.

```
/var/www/html/joomla$ vendor/bin/codecept generate:cept acceptance TestCest
Test was created in /var/www/html/joomla/tests/acceptance/TestCept.php
```

Diese Datei enthält unmittelbar nach der automatischen Generierung die folgenden drei Zeilen.

```
<?php
$I = new AcceptanceTester($scenario);
$I->wantTo('perform actions and see result');
```

Nicht-Entwickler bevorzugen das CEPT-Format. Entwickler entscheiden sich in der Regel für das CEST-Format. Das CEST-Format unterstützt mehrere Tests innerhalb einer Datei. Außerdem können Sie Programmcode leichter wiederverwenden.

RANDBEMERKUNG:

Meiner Meinung nach eignet sich das CEST-Format nur für Akzeptanztests und Funktionstests. Wenn Sie Unittests schreiben, verwenden Sie besser das [hierfür vorgesehene Format](#). Das CEST-Format bietet von Haus aus keine [Assertion](#) und keine Methoden zum Erstellen von Testduplikaten.

Annotationen

Mit [Annotationen](#) können Sie bestimmte Anmerkungen in Ihren Programm Quelltexten verwenden. Mit Hilfe von Zusatzwerkzeugen lassen sich diese Anmerkungen, die im Grunde genommen nichts anderes als Metainformationen sind, in den PHP Programmcode einbetten und auswerten.

Dies ist insbesondere dann sinnvoll, wenn PHP die Funktionen, die die Annotationen bieten, nicht selbst bereitstellt. Die Metadaten werden in Kommentaren vor dem PHP-Compiler verborgen, da dieser die Syntax nicht unterstützt. [Codeception](#), beziehungsweise [PHPUnit](#), bieten Zusatzwerkzeuge, die Annotationen auswerten und richtig weiterverarbeiten können. Sie werden später praktische Anwendungsfälle kennenlernen.

Page-Objekte und Step-Objekte

Stellen Sie sich vor, Sie haben viele unterschiedliche Tests geschrieben, in denen auf der Startseite das HTML-Element mit der ID *username* vorkam. Nun muss der Name dieser ID geändert werden. Ganz egal warum. Sie müssen nun viele unterschiedliche Tests korrigieren und Ihnen ist sicherlich nun schon klar, worauf ich hinaus will.

Sinnvoll ist es, Elemente des [Document Objekt Modell \(DOM\)](#) in Tests nur einmal „hart“ zu codieren und dann immer wieder zu verwenden: Wenn beispielsweise der Name eines Elementes nur an einer Stelle im Programmcode steht, muss nur diese eine Stelle angepasst werden, falls der Name sich einmal ändert. Codeception hat hierfür **Page-Objekte** vorgesehen. Dabei repräsentiert das Page-Objekt eine Website und die DOM-Elemente sind Eigenschaften des Page-Objekts. Je nach Komplexität der Website kann das Page-Objekt aber auch nur einen Teil der Website, zum Beispiel eine Toolbar, abbilden. Bei den Funktionstests, genauer im Kapitel *Funktionstest | Wiederverwendbare Tests*, werden wir praktisch mit Page-Objekten und Step-Objekten arbeiten.

Step-Objekte stellen eine andere Art zum Wiederwenden von Programmcode dar. Codeception hat Step-Objekte dazu vorgesehen, den Programmcode von Aktionen zur

wiederholten Nutzung zur Verfügung zu stellen. Wenn Sie eine Website testen, die aus vielen Formularen besteht, könnten Sie beispielsweise ein Step-Objekt schreiben, das die Aktionen in den Formularen simuliert. Hier könnte eine Methode mit zwei Parametern implementiert sein, die die Auswahl in einem Listefeld testet. Nennen wir die Methode `selectListValue(listenfeld $l, wert $w)`. Immer dann, wenn Sie die Auswahl in dem Listefeld testen möchten, müssen Sie nur diese eine Methode implementieren. Zum Beispiel reicht `selectListValue(„tiere“, „hund“)` aus, um die Belegung der Liste `tiere` mit der Option `hund` zu testen.

Codeception unter die Lupe genommen

Sehen wir uns die einzelnen Verzeichnisse von Codeception einmal kurz an. Nach der Installation von Codeception zu Beginn dieses Kapitels haben Sie den Ordner `tests` mit folgendem Inhalt vorgefunden.

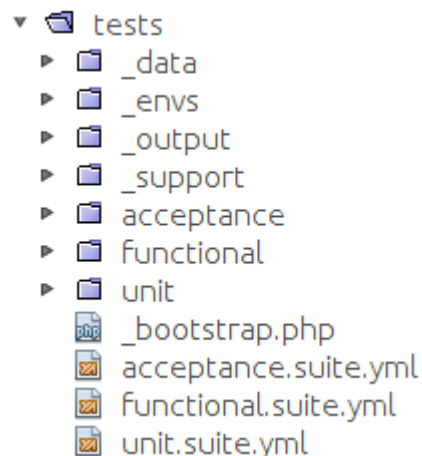


Abbildung 31: Die Verzeichnisse von Codeception. 974.png

Ordnerstruktur

`_data`

Im Verzeichnis `_data` können Sie einen Export aus einer Datenbank ablegen. Wenn Sie möchten, können Sie Codeception über die Konfiguration so einstellen, dass vor dem Ausführen eines Tests eine Datenbank mit den Daten des Exports angelegt wird und die Tests so in einem definierten Zustand ausgeführt werden können.

RANDBEMERKUNG:

Um einen Datenbankexport automatisch vor jedem Testdurchgang in Ihre Datenbank einlesen zu lassen, müssen Sie das `Db` Modul aktivieren. Wie Sie das tun können, steht unter der Überschrift *Db* in der [Dokumentation von](#)

[Codeception](#). Wollen Sie verstehen, wie dieses Modul genau arbeitet? Dann sehen Sie sich die Datei `joomla/vendor/codeception/codeception/src/Codeception/Module/Db.php` an. Sie können das Modul an Ihre Umgebung anpassen, falls es bei Ihnen Besonderheiten gibt. Im Joomla! Projekt gibt es zum Beispiel ein Präfix für jede Datenbanktabelle. Deshalb wurde von der Joomla!-Community das Modul [JoomlaDb](#) kreiert. JoomlaDb erweitert das Codeception Modul Db so, dass genau dieses Präfix unterstützt wird.

`_env`

Möchten Sie Tests in unterschiedlichen Installationsumgebungen, die verschiedene Konfigurationen voraussetzen, ausführen? Ein typischer Anwendungsfall ist ein Akzeptanztest, der in verschiedenen Browsern durchgeführt werden soll. Es kann aber auch sein, dass Ihre Webanwendung mehrere Datenbanken unterstützen soll. Wenn Sie also mit verschiedenen Konfigurationen gleichzeitig arbeiten möchten, können Sie Codeception dafür konfigurieren. Das Verzeichnis `_env` ist für die verschiedenen Konfigurationsdateien vorgesehen. Sehen Sie sich aber vorher auch einmal [Docker](#) an. Mit Docker können Sie unterschiedliche Testumgebungen sehr gut isoliert voneinander und sogar parallel ausführen. In der Codeception Dokumentation können Sie mehr zum Thema [Codeception und Docker](#) lesen.

`_output`

Diesen Ordner werden Sie noch zu schätzen wissen. Wenn Sie später Tests ablaufen lassen und einer dieser Tests fehlschlägt, finden Sie hier Informationen zur Fehlerursache.

`_support`

In diesem Ordner finden Sie selbst erstellte oder automatisch generierte Dateien, die Programmcode zur Wiederverwendung enthalten. Die meisten dieser Datei wurden erstellt, als wir im Kapitel *Testtypen in Codeception* den `build`-Befehl ausgeführt haben.

`acceptance`

Das Verzeichnis `acceptance` einhält die in Gherkin geschriebenen Feature Dateien. Also die Dateien, die für die Ausführung der Akzeptanztests in Codeception zuständig sind. In diesem Verzeichnis ist zusätzlich eine Datei abgelegt, die `_bootstrap.php` heißt. In der Datei `_bootstrap.php` können Sie Programmcode hinterlegen, der vor jedem Akzeptanztest automatisch ausgeführt werden soll.

functional

Das Verzeichnis `functional` enthält alle Dateien zu Ihren Funktionstests und die Datei `_bootstrap.php`.

unit

Das Verzeichnis `unit` enthält die Dateien zu Ihren Unittests und die Datei `_bootstrap.php`.

Konfiguration

`_bootstrap.php`

In Codeception gibt es mehrere Dateien, die `_bootstrap.php` heißen. Die drei `_bootstrap.php` Dateien in den Unterverzeichnissen zu den drei Testtypen hatte ich Ihnen gerade eben schon genannt. Diese drei Dateien führen Programmcode vor jedem Testlauf aus – speziell für den einen Testtyp, in deren Verzeichnis sie sich befinden. Sicherlich gibt es aber bestimmte Initialisierungen, die für alle Testtypen gleichermaßen gelten. Alles, was für jeden Testtyp gilt, können Sie der Einfachheit halber in die Datei `_bootstrap.php`, die direkt im Verzeichnis `tests` liegt, einfügen.

Kommen wir noch einmal auf unser Beispiel Plugin *Agpaypal* zurück. Für diese Joomla! Erweiterung gibt es ein paar Variablenbelegungen, auf die wir später in allen Tests zugreifen möchten. Bitte fügen Sie den nachfolgenden Programmcode in die Datei `/var/www/html/joomla/tests/_bootstrap.php` ein und speichern die Datei dann.

```
<?php
define("_JEXEC", 'true');
error_reporting(E_ALL);
ini_set('display_errors', 1);
define('JINSTALL_PATH', '/var/www/html/joomla');
define('JPATH_LIBRARIES', JINSTALL_PATH . '/libraries');
require_once JPATH_LIBRARIES . '/import.legacy.php';
require_once JPATH_LIBRARIES . '/cms.php';
```

Was bewirken die Einträge genau? Die Zeile `define("_JEXEC", 'true');` setzt die Konstante `JEXEC`. Nur wenn diese Konstante definiert ist, kann Joomla! ausgeführt und somit auch getestet werden.

RANDBEMERKUNG:

Alle Skripte innerhalb von Joomla! beginnen mit der Zeile `defined('_JEXEC')` oder die; Auch in unser Plugin *Agpaypal* haben wir diese Zeile integriert. Dieser Eintrag bewirkt, dass der Joomla! Prozess nicht weiter ausgeführt wird, wenn die Konstante `_JEXEC` nicht gesetzt ist. Dies ist eine Sicherheitsfunktion innerhalb von Joomla!. Kein Skript innerhalb von Joomla! kann somit von außen so manipuliert werden, dass es etwas anderes macht, als die Joomla! Entwickler mit ihm beabsichtigen.

Mit [`error_reporting\(E_ALL\);`](#) und [`ini_set\('display_errors', 1\);`](#) bewirken wir, dass Fehler nicht unterdrückt, sondern angezeigt werden. Das Unterdrücken von Fehlern ist in der Produktivumgebung beim Kunden gewollt. In unserer Entwicklungsumgebung möchten wir aber auf alle Fehler aufmerksam gemacht werden.

Und last but not least bewirkt das Einbinden der Dateien [`import.legacy.php`](#) und [`cms.php`](#), dass Sie in den Tests auf alle Joomla! Bibliotheken zugreifen können.

/codeception.yml

Die Datei `joomla/codeception.yml` enthält die Standardwerte für die notwendigen Konfigurationsangaben für alle drei Testtypen. Den bootstrap-Befehl hatten wir zu Beginn des Kapitels im Abschnitt *Codeception | Installation* ausgeführt. Ich drucke Ihnen hier die automatisch erstellte Version dieser Konfigurationsdatei ab. Wir werden diese Datei im späteren Verlauf erweitern. Im Kapitel *Analyse* werden wir hier die Parameter für die Messung der Testcodeabdeckung, die für alle Tests gleichzeitig gelten sollen, eintragen.

Weitere Informationen zu den einzelnen Konfigurationsparametern finden Sie auf der Website von Codeception in der [Dokumentation](#).

```
actor: Tester
paths:
  tests: tests
  log: tests/_output
  data: tests/_data
  support: tests/_support
  envs: tests/_envs
settings:
```

```
bootstrap: _bootstrap.php
colors: true
memory_limit: 1024M
extensions:
  enabled:
    - Codeception\Extension\RunFailed
modules:
  config:
    Db:
      dsn: "
      user: "
      password: "
      dump: tests/_data/dump.sql
```

/tests/acceptance.suite.yml, /tests/functional.suite.yml und /tests/unit.suit.yml

Die Dateien joomla/tests/acceptance.suite.yml, joomla/tests/functional.suite.yml und joomla/tests/unit.suite.yml enthalten die Standardwerte für die notwendigen Konfigurationsangaben des jeweiligen Testtyps. Den bootstrap-Befehl hatten wir zu Beginn dieses Kapitels im Abschnitt *Codeception | Installation* ausgeführt.

Ich drucke Ihnen hier die automatisch erstellten Versionen dieser Konfigurationsdateien ab. Wir werden auch diese Datei im späteren Verlauf erweitern.

acceptance.suite.yml

```
class_name: AcceptanceTester
modules:
  enabled:
    - PhpBrowser:
        url: http://localhost/myapp
    - \Helper\Acceptance
```

functional.suite.yml

```
class_name: FunctionalTester
modules:
  enabled:
    # add framework module here
    - \Helper\Functional
```

unit.suite.yml

```
class_name: UnitTester
modules:
  enabled:
    - Asserts
    - \Helper\Unit
```

Arbeiten mit Codeception

Dass Sie Codeception über den Befehl `codecept` ausführen, haben Sie schon in Beispielen kennengelernt. Sie haben auch schon Steuerbefehle kennen gelernt, nämlich `build`, `generate` und `run`.

Die ausführbare Datei `codecept.php` finden Sie im Verzeichnis `vendor/bin/`. Führen Sie die Datei doch einfach einmal mit der Option `-v` aus. Dann werden Ihnen alle möglichen Optionen und Argumente angezeigt. Ich beschränke mich hier auf `build`, `generate` und `run`.

- Mit dem Befehl *build* veranlassen Sie Codeception dazu, automatisch Programmcode anhand der Eintragungen in den Konfigurationsdateien zu generieren.
- Mit dem Befehl *run* führt Codeception Tests aus. Wenn Sie `codecept run` starten, können Sie weitere Argumente mitgeben:

```
/var/www/html/joomla$ vendor/bin/codecept run [optionen] [suite] [test].
```

Der Befehl

```
/var/www/html/joomla$ vendor/bin/codecept run unit
```

```
tests/unit/suites/plugins/content/agpaypal/agpaypalTest
```

führt beispielsweise genau den einen Unittest **agpaypalTest** aus.

Und die Eingabe

```
/var/www/html/joomla$ vendor/bin/codecept run -q unit
```

unterdrückt die Ausgabe in der Konsole. Die Option `q` steht für das englische Wort `quiet`.

RANDBEMERKUNG:

Sie können alle Unittests in einer Suite gleichzeitig ausführen. Verwenden Sie dazu einfach den Befehl

```
tests/codeception/vendor/bin/codecept run unit
```

ohne dabei eine spezielle Datei als Parameter mitzugeben.

Wenn Sie später auch Funktionstests und Akzeptanztests geschrieben haben, können Sie mit dem Befehl `tests/codeception/vendor/bin/codecept run` alle drei Testarten auf einen Schlag ausführen.

- Mit dem Befehl *generate* erstellen Sie die Testdateien. Beim Befehl `generate` müssen Sie den Namen einer Suite und einen Namen für den Test, der erstellt werden soll, mitgeben.
 - `generate:cept` erstellt einen Test im CEPT-Format
 - `generate:cest` erstellt einen Test im CEST-Format
 - `generate:phpunit` erstellt einen PHPUnittest ohne die Erweiterungen, die Codeception bietet. Der Test erbt von der Klasse `PHPUnit_Framework_TestCase` und enthält die Methoden `_setUp()` und `tearDown()` automatisch.
 - `generate:test` erstellt einen PHPUnittest der von der Klasse `Codeception\Test\Unit` erbt und zusätzlich zu den Methoden `_setUp()` und `tearDown()` die Methoden `_before()` und `_after()` enthält.

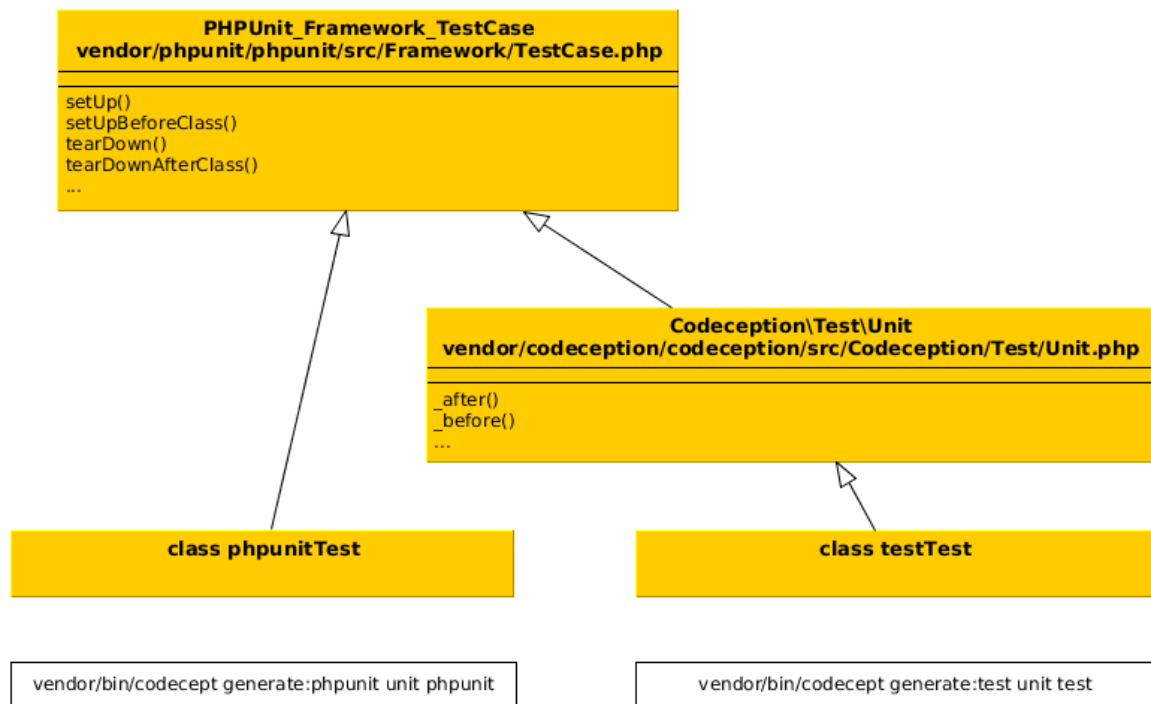


Abbildung 32: Die Klasse Unit erweitert das PHPUnit Framework 959.png

Dies war nun sehr viel Theorie. Sie werden alle diese Befehle im weiteren Verlauf dieses Buches, insbesondere im Kapitel *Unittests | Das erste Testbeispiel verbessern | Vor dem Test den Testkontext herstellen und hinterher aufräumen*, auch praktisch kennenlernen.

Kurzgefasst

Codeception und Composer laufen nun auf Ihrem Rechner. Außerdem wissen Sie, welche Strategien diese beide Werkzeuge verfolgen und wie Sie die beiden Programme konfigurieren können. Wir können nun die ersten Tests erstellen!

Unittests

The fewer tests you write,
the less productive you are
and the less stable your code becomes.
[[Erich Gamma](#)]

Unittests sind Tests, die Sie gleichzeitig mit der Programmierung des eigentlichen Programms erstellen können. Wenn Sie testgetrieben entwickeln, sollten Sie die Tests

vor dem eigentlichen Programmcode erstellen. Jeder Test erfolgt, wie der Name schon vermuten lässt, isoliert von anderen Programmteilen. Unittests testen kleinste Einheiten. Im Deutschen nennt man diese Tests auch Modultest oder Komponenten-Test. Beim Erstellen eines Unittests kennen Sie die Implementierung der zu prüfenden Unit oder haben zumindest eine Vorstellung davon, wie Sie die Unit programmieren möchten. Unittests zählen aber trotzdem zu den spezifikationsorientierten Tests, also den Black-Box-Tests: Das Ziel von Unittests ist es nicht, den Programmcode systematisch zu überprüfen und möglichst alle Programmcodeabschnitte zu testen. Dies tun die implementierungsorientierten White-Box-Tests. Ziel von Unittests ist es, alle Testfälle abzudecken, egal ob dabei alle Programmcodeabschnitte durchlaufen werden oder nicht.

RANDBEMERKUNGEN:

Am Ende dieses Buches werden wir uns im Kapitel *Analyse* ein Werkzeug ansehen, das die [Testabdeckung](#) berechnet. Hier wird der Programmcode systematisch durchsucht. Es handelt sich aber auch nicht um einen White-Box-Test im eigentlichen Sinne. Ziel der Tools zur Messung der Codeabdeckung ist es, sicherzustellen, dass kein wichtiger Testfall vergessen wurde. Da eine hohe Codeabdeckung auch mit Tests erreicht werden kann, die nichts Wichtiges überprüfen, hat die Codeabdeckung für die Qualität der Tests eine nur eingeschränkte Aussagekraft.

Black-Box-Tests und White-Box-Tests hatte ich in einer Randbemerkung im Kapitel *Softwaretests – eine Einstellungssache? | Kontinuierliche Testes | Behaviour-Driven-Development (BDD)* ausführlicher beschrieben.

Ziel dieses Kapitels ist es, Ihnen einen Einblick in die testgetriebene Entwicklung zu geben. Nebenbei erfahren Sie, wie Sie Unittests mithilfe von PHPUnit und Codeception erstellen können.

Einen ersten Überblick verschaffen

Ich zeige Ihnen hier am Beispiel einer Erweiterung für Joomla! wie Sie Unittest erstellen können. Sie lesen dieses Buch sicherlich, weil Sie sich überlegen, Tests in ihre eigene Software einzubauen. Falls Sie noch nicht testgetrieben entwickeln, kommt Ihnen diese Methode zunächst vielleicht sehr umständlich vor. Mir ging es anfangs zumindest so. Eine neue Funktionalität einfach einzubauen erscheint viel unkomplizierter. Der Mehraufwand für den Test kommt einem unnötig vor. Außerdem

kann man diesen Test ja, wenn nötig, nachträglich noch hinzufügen. Ich empfehle Ihnen die testgetriebene Vorgehensweise einmal auszuprobieren. Vielleicht geht es Ihnen ja so wie mir und Sie möchten nicht mehr darauf verzichten.

Endlich: Ein erstes Testbeispiel

Codeception führt Unittests mit PHPUnit aus. Arbeiten Sie schon mit PHPUnit? Dann müssen Sie nicht umlernen. Sie können jeden schon fertig geschriebenen PHPUnit-Test in die Codeception Testsuite integrieren und hier ausführen. Ich empfehle Ihnen aber einmal über den PHPUnit Tellerrand auf Codeception zu blicken, denn Codeception bietet Ihnen eine Menge nützlicher Zusatzfunktionen.

Die Vorlage für den ersten Test

Eine dieser Zusatzfunktionen von Codeception haben Sie schon benutzt, als Sie den ersten Test während der Installation von Codeception erstellten – nämlich den Testcode-Generator. Mit dem Befehl `vendor/bin/codecept generate:test unit`

```
/var/www/html/joomla$ vendor/bin/codecept generate:test unit
/suites/plugins/content/agpaypal/agpaypal
```

haben Sie den ersten Test erstellt. Sie finden diesen Test in der Datei `agpaypalTest.php` im Verzeichnis `/joomla/tests/unit/suites/plugins/content/agpaypal/`

Ausführen können Sie den Test in der Datei `agpaypalTest.php` über den Befehl `codecept run unit tests/unit/suites/plugins/content/agpaypal/agpaypalTest`.

```
/var/www/html/joomla$ vendor/bin/codecept run unit
tests/unit/suites/plugins/content/agpaypal/agpaypalTest
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.13 by Sebastian Bergmann and contributors.
Unit Tests (1) -----
✓ agpaypalTest: Me (0.00s)
-----
Time: 149 ms, Memory: 8.00MB
OK (1 test, 0 assertions)
```

Der gerade neu generierte Test verläuft erfolgreich. Alles andere wäre auch verwunderlich. Bisher gibt es ja noch keinen wirklichen Testcode. Die automatisch erstellte Testdatei enthält ausschließlich leere Methoden.

```
<?php
namespace suites\plugins\content\agpaypal;
class agpaypalTest extends \Codeception\Test\Unit {
protected $tester;
protected function _before() {}
protected function _after() {}
public function testMe() {}
}
```

Damit die Testklasse selbst Funktionen von Codeception nutzen kann, lassen wir sie von der Framework-Klasse `\Codeception\Test\Unit` erben. Im nächsten Abschnitt werden wir den ersten Test programmieren. Wichtig ist, dass dessen Signatur, analog des automatisch erstellten Beispieltests `testMe()`, der Konvention für PHPUnit Testfallmethoden folgt. Hiernach muss der Methodenname mit dem Wort *test* beginnen.

RANDBEMERKUNG:

Falls Sie aus irgendeinem Grund den Namen einer Testmethode nicht mit dem Präfix `test` versehen möchten, können Sie auch die Annotation [`@test`](#) verwenden.

Der erste selbst programmierte Test

Was wollen wir testen?

Für unser Plugin *Agpaypal* hatten wir im Kapitel *Tests planen* grob folgende Testfälle festgehalten: Bei der Prüfung eines Textes auf ein bestimmtes Muster, das in einen anderen Text umgewandelt werden soll, kann dieses Muster

- genau einmal,
- keinmal oder
- mehrmals im Text

vorkommen.

Wie integrieren wir am besten Tests, die diese Testfälle erfüllen? Um diese Frage zu beantworten, sehen wir uns den Programmcode unseres Plugins noch einmal an.

```
/var/www/html/joomla/plugins/content/agpaypal/agpaypal.php
<?php
defined('_JEXEC') or die;
class plgContentAgpaypal extends JPlugin{
    public function onContentPrepare($context, &$row, $params, $page = 0)    {
        $search = "@paypalpaypal@";
        $replace = '<form name="_xclick" action="https://www.paypal.com/de/cgi-bin/webscr"
method="post">
<input type="hidden" name="cmd" value="_xclick">
<input type="hidden" name="business" value="me@mybusiness.com">
<input type="hidden" name="currency_code" value="EUR">
<input type="hidden" name="item_name" value="Teddybär">
<input type="hidden" name="amount" value="12.99">
<input type="image" src="http://www.paypal.com/de_DE/i/btn/x-click-but01.gif" border="0"
name="submit" alt="Zahlen Sie mit PayPal – schnell,
kostenlos und sicher!">
</form>';
        if (is_object($row)){
            $row->text = str_replace($search, $replace, $row->text);
        } else{
            $row = str_replace($search, $replace, $row);
        }
        return true;
    }
}
```

Fangen wir mit dem Testfall an, bei dem der Text @paypalpaypal@ in der Variablen \$row, und somit im Text des anzuzeigenden Beitrags, genau einmal vorkommt. Bei einem fehlerfreien Programmablauf sollte folgendes passieren: Die Variable \$row enthält nach Durchlaufen der Methode onContentPrepare() anstelle des Textes

@paypalpaypal@

den Text

```
<form name="_xclick" action="https://www.paypal.com/de/cgi-bin/webscr" method="post">
<input type="hidden" name="cmd" value="_xclick"><input type="hidden" name="business"
value="me@mybusiness.com">
<input type="hidden" name="currency_code" value="EUR">
<input type="hidden" name="item_name" value="Teddybär">
<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" alt="Zahlen Sie mit PayPal – schnell, kostenlos und sicher!">
</form>
```

Ausschließlich diesen Punkt müssen wir testen, wenn wir die korrekte Bearbeitung bei einmaligem Vorkommen des Textes @paypalpaypal@ sicherstellen möchten. Ob andere Dinge korrekt gehandhabt werden, wird in den dafür zuständigen Klassen geprüft. Im Moment kümmern wir uns also noch nicht darum, ob zum Beispiel die Anzeige der Schaltfläche im Beitrag auf der Website korrekt ist. Wir testen hier nur diese Einheit, also die Methode `onContentPrepare()` unabhängig vom ganzen Programm!

Beim testgetriebenen Erstellen von Unittests haben sich in der Praxis die folgenden Regeln bewährt, die ich Ihnen hiermit ans Herz lege:

1. Prüfen Sie vor dem Hinzufügen einer neuen Funktion immer erst, ob alle schon vorhandenen Tests fehlerfrei durchlaufen werden und korrigieren Sie etwaige Fehler.
2. Implementieren Sie Tests, die die kleinste mögliche Einheit im Programmcode testen.
3. Implementieren Sie die Tests unabhängig voneinander.
4. Geben Sie den Tests sinnvolle Namen, damit diese gut wartbar und aussagekräftig sind und bleiben.

Die Implementierung des ersten Tests

Die Methoden in der generierten Testdatei enthalten noch keinen Programmcode. Dieses ändern wir nun! Sie finden die Testdatei `agpaypalTest.php`, also die Datei mit den leeren Methoden, im Verzeichnis `/joomla/tests/unit/suites/plugins/content/agpaypal/`. Zumindest dann, wenn Sie diese vorher während der Installation von Codeception im Kapitel *Codeception – ein Überblick | Codeception | Installation* mit mir erstellt haben. Öffnen Sie diese Datei bitte nun in Ihrer Entwicklungsumgebung zur Bearbeitung.

Für den ersten Test erstellen wir die Methode

testOnContentPrepareIfSearchstringIsInContentOneTime(). Jetzt wird es endlich konkret! Sehen Sie sich zunächst einmal die fertige Methode im nachfolgenden Programmcodebeispiel an. Ab hier habe ich den Text für den Paypal Jetzt-kaufen-Button der Übersicht halber um vier Zeilen gekürzt.

```
<?php
namespace suites\plugins\content\agpaypal;
class agpaypalTest extends \Codeception\Test\Unit {
    protected $tester;
    protected function _before() { }
    protected function _after() { }
    public function testOnContentPrepareIfSearchstringIsInContentOneTime()
    {
        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 = '<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>';
        $class->onContentPrepare("", $contenttextbefore, $params);
        $this->assertEquals($contenttextafter, $contenttextbefore);
    }
}
```



```
| }  
| }
```

Sehen wir uns den Programmcode der Testmethode genauer an: Ich habe für die Bedingung, dass der Text `@paypalpaypal@` zur Anzeige im Frontend in den Text, der einen PayPal Jetzt-kaufen-Button in einem HTML-Dokument anzeigt, umgewandelt wird, eine Behauptung aufgestellt. Diese Behauptung habe ich mithilfe der `assertEquals()` Methode formuliert. In dieser Methode musste ich die Parameter so setzen, dass meine Behauptungen erfüllt sind.

Die Variable `$contenttextafter` enthält den Text, den ich in meiner Behauptung erwarte und `$contenttextbefore` enthält den Eingabetext – also den Text, der zu Beginn die Passage `@paypalpaypal@` enthält. Nach dem Durchlaufen der Methode `onContentPrepare()` soll `$contenttextbefore` aber gleich `$contenttextafter` sein. Ich behaupte also `assertEquals($contenttextafter, $contenttextbefore)`. Den Rest übernimmt das PHPUnit-Framework.

Der Satz „Hier musste ich nur die Parameter so setzen, dass die Bedingungen erfüllt sind.“ hört sich einfach an. Ganz so einfach ist dies aber nicht:

- Als Erstes habe ich ein Objekt des Typs `PlgContentAgpaypal` erzeugt und in der Variablen `$class` gespeichert. Dazu musste ich die Objekte `$subject` und `$config` erzeugen. Diese Objekte benötigen die Klasse `PlgContentAgpaypal` bei der Instanziierung.
- Die Objekte `$params` und `$contenttextbefore` habe ich danach erstellt, weil diese als Parameter in der Methode `onContentPrepare()` benötigt werden. `$contenttextafter` enthält den Text, in den `$contenttextbefore` umgewandelt werden soll.
- Nun habe ich die zu testende Methode aufgerufen: `$class->onContentPrepare("$contenttextbefore, $params);`
- Zum Schluss habe ich dann die im vorhergehenden Kapitel *Was wollen wir testen?* beschriebenen Bedingungen als Parameter in die Methode `assertEquals()` eingefügt. Diese Methode verifiziert die Gleichheit zweier Objekte.

Nachdem nun klar ist, was der Testprogrammcode genau macht, führen wir diesen mithilfe des Befehls `codecept run unit` aus.

RANDBEMERKUNG:

Da wir im Moment nur einen Test in unserer Suite haben, können wir problemlos die ganze Suite auf einen Schlag ausführen. Mit dem Befehl `codecept run unit tests/unit/suites/plugins/content/agpaypal/agpaypalTest` könnten Sie den Test auf die Testdatei `agpaypalTest.php` beschränken.

```
/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 if searchstring is in content one time (0.01s)
-----
Time: 158 ms, Memory: 10.00MB
OK (1 test, 1 assertion)
```

Der Test war erfolgreich. Wurde er bei Ihnen auch als erfolgreich markiert? Dann probieren Sie doch einfach einmal was passiert, wenn Sie einen Buchstabendreher in die Variable `$contenttextbefore` oder die Variable `$contenttextafter` einbauen. Dann müsste Ihnen nämlich ein Fehler angezeigt werden! Wenn der Test bei Ihnen nicht erfolgreich war, haben Sie den Buchstabendreher schon irgendwo im Programmcode und sollten diesen nun suchen und korrigieren.

Standardmäßig sagt Ihnen PHPUnit genau, was fehlgeschlagen ist. Oft ist aber nicht direkt klar, warum dieser Fehler genau aufgetreten ist. Insbesondere dann, wenn Sie sehr viele aktive Tests haben, dieser eine, fehlgeschlagene Test ganz zu Beginn erstellt wurde und Sie sich nicht mehr exakt an diesen Test erinnern können.

Sie können sich das Leben leichter machen, wenn Sie Erläuterungen in den Testanweisungen als Parameter mitgeben. Zum Beispiel könnten Sie folgende Anweisung verwenden:

```
$this->assertEquals($contenttextafter, $contenttextbefore, "@paypalpaypal@ wurde nicht richtig umgewandelt als er genau einmal im Text vorkam.");
```

Der Text `"@paypalpaypal@ wurde nicht richtig umgewandelt als er genau einmal im Text vorkam."` wird so bei einem fehlgeschlagenen Test neben der Nachricht von PHPUnit mit ausgegeben. Dieser kleine Mehraufwand kann Ihnen auf lange Sicht

bei der Testanalyse viel Zeit sparen.

In diesem Buch hier nutze ich diesen Parameter nicht. Die Codebeispiele sind ohne diesen Parameter kürzer und kompakter.

Testgetrieben entwickeln

Wie sieht die testgetriebene Entwicklung nun in der Praxis aus? Ich möchte Sie dazu einladen mir einmal über die Schulter zu schauen. So können Sie einen ersten Eindruck und somit ein Gefühl für diese Methode bekommen. In der testgetriebenen Entwicklung schreiben Sie den Test, bevor Sie den eigentlichen Programmcode schreiben. Der Test ist also schon da, wenn Sie den Programmcode, der diesen Test erfüllt, erstellen. Sie starten mit einem ganz einfachen Design. Dieses Design verbessern Sie dann fortwährend. So werden komplizierte Designelemente, die nicht unbedingt notwendig sind, vermieden. Sie wählen immer den einfachsten Weg!

Ein testgetriebenes Programmierbeispiel

Kommen wir auf unser Beispiel zurück. Bisher wird der PayPal Jetzt-kaufen-Button mit fixen Werten eingesetzt. Ihre nächste Aufgabe ist es nun, eine Möglichkeit zu schaffen, dieser Schaltfläche unterschiedliche Preise zuzuordnen. Okay, wie machen Sie das? Ich schlage vor, Sie geben dem auszutauschenden Text einen Parameter. Warum machen Sie nicht gleich alle Optionen des Buttons flexibel? Vermutlich wird diese Aufgabe später auf Sie zukommen? Weil wir in kleinen Schritten vorgehen. Im Moment geht es nur darum, diese eine Option flexibel zu machen. Mehr ist momentan noch nicht gefordert. Beginnen wir mit der Implementierung des Tests.

Erst der Test ...

Indem wir zuerst den Test für eine Funktion erstellen, können wir nach jeder Änderung am Code sicherstellen, dass alle geforderten Funktionen erfüllt sind. Was ist nun genau der nächste Testfall? Wir haben uns schon dazu entschieden dem Suchtest einen Parameter mitzugeben. Eine mögliche Umsetzung wäre, den Betrag in die Mitte des Suchtextes einzufügen. Zum Beispiel so: `@paypal amount=15.00 paypal@`. Im nachfolgenden Codebeispiel finden Sie die dazugehörige Testmethode. Die Variable `$contenttextbefore` enthält den Parameter im Suchtext und in der Variable `$contenttextafter` wurde der Betrag ebenfalls angepasst.

```
<?php
namespace suites\plugins\content\agpaypal;
class agpaypalTest extends \Codeception\Test\Unit {
```

```

protected $tester;
protected function _before() {}
protected function _after() {}
public function testOnContentPrepareIfSearchstringIsInContentOneTimeAndContainsNoValue()
{...}
public function
testOnContentPrepareIfSearchstringIsInContentOneTimeAndContainsValueForAmount() {
    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 = '<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>';
    $class->onContentPrepare("", $contenttextbefore, $params);
    $this->assertEquals($contenttextafter, $contenttextbefore);
}
}

```

Dieser Test ist im Code der Produktivversion noch nicht umgesetzt und wird deshalb fehlschlagen. Überzeugen Sie sich selbst. Ich habe die Ausgabe der Kommandozeile im nächsten Kasten abgedruckt.

```

/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 (2) -----
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains no value
(0.00s)
✗ agpaypalTest: On content prepare if searchstring is in content one time and contains value for
amount (0.00s)
-----
Time: 144 ms, Memory: 10.00MB
There was 1 failure:
...
FAILURES!
Tests: 2, Assertions: 2, Failures: 1.

```

... dann die Funktion

Als Nächstes geht es nun darum, den Test zu erfüllen – und zwar auf möglichst einfache Art und Weise. Sicherlich denken Sie schon weiter. Sie vermuten wahrscheinlich, dass alle Werte die mit dem PayPal Jetzt-kaufen-Button mitgegeben werden, änderbar sein sollen. Wahrscheinlich haben Sie auch hierfür schon eine Lösung im Kopf. Komplizierte Designelemente, die nicht unbedingt notwendig sind, sollten Sie aber aufschieben. Sie programmieren nur das, was Sie tatsächlich jetzt gerade benötigen. Nicht das, was vielleicht später notwendig ist.

Erfüllen wir also nun den Test auf möglichst einfache Art und Weise: Dazu suchen wir die Texte @paypal und paypal@ mithilfe der Methode [preg_match_all\(\)](#). `preg_match_all()` speichert alle Funde in der Variablen `$matches[0][0]` und der Suchtext `(.*)` extrahiert den Text zwischen @paypal und paypal@ in der Variablen `$matches[1][0]`. Nun prüfen wir, ob `$matches[1][0]` den Text `amount=` enthält. Falls ja, belegen wir den auszutauschenden Suchtext mit `$matches[0][0]` und passen den Wert für den Preis an. Falls nicht, lassen wir alles beim Alten. Dies hat zudem keine negativen Auswirkungen auf den schon bestehenden Testfall. Dieser bleibt weiterhin erfüllt.

```

<?php
defined('_JEXEC') or die;
class PlgContentAgpaypal extends JPlugin {
    public function onContentPrepare($context, &$row, $params, $page = 0) {

```

```

        if (is_object($row)) {
            preg_match_all('/@paypal(.*)paypal@/i', $row->text, $matches,
PREG_PATTERN_ORDER);
        }
        else {
            preg_match_all('/@paypal(.*)paypal@/i', $row, $matches,
PREG_PATTERN_ORDER);
        }
        if (count($matches[0] > 0) && (strpos($matches[1][0], 'amount=') !== false) > 0) {
            $search = $matches[0][0];
            $amount = trim(str_replace('amount=', '', $matches[1][0]));
        }
        else {
            $search = "@paypalpaypal@";
            $amount = '12.99';
        }
        $replace = '<form name="_xclick" action="https://www.paypal.com/de/cgi-bin/webscr"
method="post">'
            . '<input type="hidden" name="amount" value="' . $amount . '">'
            . '<input type="image" src="http://www.paypal.com/de_DE/i/btn/x-click-but01.gif"
border="0" name="submit">'
            . '</form>';
        if (is_object($row)) {
            $row->text = str_replace($search, $replace, $row->text);
        }
        else {
            $row = str_replace($search, $replace, $row);
        }
        return true;
    }
}

```

Probieren Sie es aus! Was passiert, wenn Sie nun den Befehl `vendor/bin/codecept run unit` in die Kommandozeile eingeben? Die beiden Testfälle sind erfüllt.

```

/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 (2) -----

✓ agpaypalTest: On content prepare if searchstring is in content one time and contains no value (0.01s)

✓ agpaypalTest: On content prepare if searchstring is in content one time and contains value for amount (0.00s)

Time: 164 ms, Memory: 10.00MB

OK (2 tests, 2 assertions)

Das ist Ihnen zu einfach? Dann treten Sie den Beweis an. Schreiben Sie einen weiteren Test, der dies belegt. Zum Beispiel einen Test zum Suchtext `@paypal aomunt=15.00 paypal@`. Wie wollen Sie damit umgehen, wenn derjenige, der den PayPal Jetzt-kaufen-Button einfügen will, sich vertippt und anstelle von `amount` das Wort `aomunt` eingibt? Eine Möglichkeit sehen wir uns im nächsten Kapitel an.

Und weiter – erst testen, dann programmieren

Im Zusammenspiel von Testen und Implementieren wird das Programm immer weiter entwickelt. Die Software wird somit in ganz kleinen Schritten geschrieben. Der vorhergehende Testfall wurde erfüllt. Nun gibt es eine neue Aufgabe.

Die Aufgabe ergibt sich meist aus der Spezifikation, die noch nicht ganz erfüllt ist – das Programm also noch nicht ganz fertig ist. Es kann aber auch sein, dass Sie mit dem bisherigen Stand noch nicht zufrieden sind – Ihnen die bisherige Lösung nicht gut genug ist. In unserm Beispiel werden zwar die Testfälle erfüllt. Es gibt aber weitere Testfälle, die ein unvorhergesehenes Programmverhalten zeigen. Ein Beispiel ist die Eingabe eines Suchtextes, bei dem ein Tippfehler vorliegt. In diesem Falle würde Ihr Plugin nicht greifen. Das Problem ist, dass der auszutauschende Text nur mit einem Parameter `amount` richtig gesetzt wird. Falls `amount` nicht vorkommt, würde nur ein Text in der Form `@paypalpaypal@` ersetzt. Ich beweise Ihnen dieses mit dem folgenden Test.

```
...
public function
testOnContentPrepareIfSearchstringIsInContentOneTimeAndContainsIncorrectValue() {
    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 = '<p>Texte vor der Schaltfläche.</p>'
        . '<p>@paypal aomunt=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>';
    $class->onContentPrepare("", $contenttextbefore, $params);
    $this->assertEquals($contenttextafter, $contenttextbefore);
}
...

```

Wenn zwischen @paypal und paypal@ Text steht, das Wort amount= aber nicht in diesem Text enthalten ist, wird kein PayPal Jetzt-kaufen-Button eingefügt. Der Test schlägt fehl:

```

/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 (3) -----
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains no value
(0.01s)
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains value for
amount (0.00s)
✗ agpaypalTest: On content prepare if searchstring is in content one time and contains incorrect value
(0.00s)

```

Time: 168 ms, Memory: 10.00MB

There was 1 failure:

....

FAILURES!

Tests: 3, Assertions: 3, Failures: 1.

Eine kleine Änderung bringt hier die Lösung. Falls Text zwischen @papyal und paypal@ enthalten ist, setzen wir den auszutauschenden Text in jedem Fall gleich \$matches[0][0]. Den Preis belassen wir auf dem Standardwert. Es gibt keine Regel, aus der wir herleiten können, welche Tippfehler genau den Parameter amount meinen. Idealerweise erweitern wir das Plugin später so, dass gar keine Tippfehler mehr vorkommen können

```
<?php
defined('_JEXEC') or die;
class PlgContentAgpaypal extends JPlugin {
    public function onContentPrepare($context, &$row, $params, $page = 0) {
        if (is_object($row)) {
            preg_match_all('/@paypal(?:)paypal@/i', $row->text, $matches,
PREG_PATTERN_ORDER);
        }
        else {
            preg_match_all('/@paypal(?:)paypal@/i', $row, $matches,
PREG_PATTERN_ORDER);
        }
        if (count($matches[0] > 0)) {
            $search = $matches[0][0];
            if ((strpos($matches[1][0], 'amount=') !== false) > 0) {
                $amount = trim(str_replace('amount=', '', $matches[1][0]));
            }
            else {
                $amount = '12.99';
            }
        }
        else {
            $search = "@paypalpaypal@";
```

```

        $amount ='12.99';
    }

    $replace = '<form name="_xclick" action="https://www.paypal.com/de/cgi-
bin/webscr" method="post">'
        . '<input type="hidden" name="amount" value="' . $amount . '">'
        . '<input type="image" src="http://www.paypal.com/de_DE/i/btn/x-click-
but01.gif" border="0" name="submit">'
        . '</form>';

    if (is_object($row)) {
        $row->text = str_replace($search, $replace, $row->text);
    }
    else {
        $row = str_replace($search, $replace, $row);
    }
    return true;
}
}

```

Ein erneuter Testdurchlauf zeigt, dass nun alle drei bisherigen Tests erfolgreich sind.

```

/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 (3) -----
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains no value
(0.00s)
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains value for
amount (0.00s)
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains incorrect value
(0.00s)
-----
Time: 116 ms, Memory: 10.00MB
OK (3 tests, 3 assertions)

```

Schön. Das beruhigt. Auch wenn es zugegebenermaßen noch viele Baustellen in dieser Joomla! Erweiterung gibt.

Weiter geht es Schritt für Schritt

Machen wir mit der nächsten Baustelle weiter. Laut Spezifikation kann es vorkommen, dass in einem Text mehrere PayPal Jetzt-kaufen-Button vorkommen. Was sagt ein Test dazu?

```
...
public function testOnContentPrepareIfSearchstringIsInContentManyTimes() {
    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 = '<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>';
        $class->onContentPrepare(" ", $contenttextbefore, $params);
        $this->assertEquals($contenttextafter, $contenttextbefore);
    }

```

Mehrere PayPal Jetzt-kaufen-Button unterstützt das Plugin in der aktuellen Version leider noch nicht. Im nächsten Kasten sehen Sie, dass der Test fehlschlägt!

```

/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 (4) -----
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains no value
(0.00s)
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains value for
amount (0.00s)
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains incorrect value
(0.00s)
✗ agpaypalTest: On content prepare if searchstring is in content many times (0.00s)
...
FAILURES!
Tests: 4, Assertions: 4, Failures: 1.

```

Was müssen wir tun, um diese neue Forderung zu erfüllen? Die Funktion `preg_match_all()` liefert uns alle Textstellen, die mit `@paypal` beginnen und mit `paypal@` enden. Diese können wir in einer Schleife durchlaufen. Dabei können wir alles, was wir bisher an Programmcode geschrieben haben, weiterhin verwenden.

```

<?php
defined('_JEXEC') or die;
class PlgContentAgpaypal extends JPlugin {
    public function onContentPrepare($context, &$row, $params, $page = 0) {
        if (is_object($row)) {
            preg_match_all('/@paypal(?:.*)paypal@/i', $row->text, $matches,
            PREG_PATTERN_ORDER);
        }
    }
}

```

```

else {
    preg_match_all('/@paypal(?:.*)paypal@/i', $row, $matches,
PREG_PATTERN_ORDER);
}
if (count($matches[0] > 0)) {
    for ($i = 0; $i < count($matches [0]); $i++) {
        $search = $matches[0][$i];
        if ((strpos($matches[1][$i], 'amount=') !== false) > 0) {
            $amount = trim(str_replace('amount=', '', $matches[1][$i]));
        }
        else {
            $amount = '12.99';
        }
        $replace = '<form name="_xclick" action="https://www.paypal.com/de/cgi-
bin/webscr" method="post">'
            . '<input type="hidden" name="amount" value="' . $amount . '">'
            . '<input type="image" src="http://www.paypal.com/de_DE/i/btn/x-
click-but01.gif" border="0" name="submit">'
            . '</form>';
        if (is_object($row)) {
            $row->text = str_replace($search, $replace, $row->text);
        }
        else {
            $row = str_replace($search, $replace, $row);
        }
    }
}
return true;
}
}

```

Und das war es auch schon. Alle Tests sind nun erfolgreich.

```
/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 (4) -----
```

```
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains no value (0.00s)
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains value for amount (0.00s)
✓ agpaypalTest: On content prepare if searchstring is in content one time and contains incorrect value (0.00s)
✓ agpaypalTest: On content prepare if searchstring is in content many times (0.00s)
-----
Time: 152 ms, Memory: 10.00MB
OK (4 tests, 4 assertions)
```

Geht das noch einfacher? Gibt es Redundanzen?

Wir haben bei allen bisherigen Schritten versucht die einfachste Lösung umzusetzen. Manchmal ist der erste Versuch aber nicht der optimale. Wir wollen aber eine tragfähige Lösung umsetzen. Das bedeutet, dass wir zusätzlich fortlaufend auch immer mal wieder einen Blick auf das Design werfen sollten. Auch dieses sollten wir beständig fortentwickeln und verbessern. Der Code sollte in jeder Phase so einfach und verständlich wie möglich bleiben. Außerdem sollten Redundanzen im Keim erstickt werden. Prüfen Sie Ihren Programmcode nach jedem Schritt und bringen ihn, wenn nötig, in eine einfachere Form. Kommt Ihnen dies zu umständlich vor? Vertrauen Sie darauf, dass gut strukturierter Programmcode während der ganzen Entwicklungsphase auch gut erweiterbar ist.

Sehen Sie sich im nachfolgenden Beispiel an, was ich genau umgebaut habe.

```
<?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;
    }
}
```

```

    }
    private function startCreateButtons(&$text) {
        preg_match_all('/@paypal(?:.*)paypal@/i', $text, $matches, PREG_PATTERN_ORDER);
        if (count($matches[0] > 0)) {
            $this->createButtons($matches, $text);
        }
        return true;
    }
    private function createButtons($matches, &$text) {
        for ($i = 0; $i < count($matches[0]); $i++) {
            $search = $matches[0][$i];
            if ((strpos($matches[1][$i], 'amount=') !== false) > 0) {
                $amount = trim(str_replace('amount=', '', $matches[1][$i]));
            }
            else {
                $amount = '12.99';
            }
            $replace = '<form name="_xclick" action="https://www.paypal.com/de/cgi-
bin/webscr" method="post">
                .<input type="hidden" name="amount" value="' . $amount . "'>
                .<input type="image" src="http://www.paypal.com/de_DE/i/btn/x-click-
but01.gif" border="0" name="submit">
                .</form>';
            $text = str_replace($search, $replace, $text);
        }
        return true;
    }
}

```

Fällt Ihnen etwas auf? Wir haben eine neue Methode erstellt. Diese Änderung hat auch Auswirkungen auf unsere Tests. Wir möchten ja immer die kleinste Einheit testen. Unser aktueller Testaufbau testet aber nun das Zusammenspiel mehrerer Methoden der Klasse `AgpaypalTest`. Wir sollten den Test so abändern, dass jede Methode für sich getestet wird. Fangen wir mit der Methode `testStartCreateButtonsIfSearchstringIsInContentOneTimeAndContainsNoValue()` an. Die Methode `contentPrepare()` heben wir uns für später auf. An ihr sehen wir uns im Kapitel *Testduplikate* Testduplikate genauer an.

```

<?php
namespace suites\plugins\content\agpaypal;
class agpaypalTest extends \Codeception\Test\Unit {
    protected $tester;
    protected function _before() { }
    protected function _after() { }
    public function testStartCreateButtonsIfSearchstringIsInContentOneTimeAndContainsNoValue() {
        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 = '<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>';
        $class->startCreateButtons($contenttextbefore);
        $this->assertEquals($contenttextafter, $contenttextbefore);
    }
}
....

```

Nach dem Umstellen sollten die Tests weiterhin alle erfüllt sein. Im nächsten Kasten sehen Sie, dass dies auch so ist.


```
/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 (5) -----
- agpaypalTest: Start create buttons if searchstring is in content one time and contains no value (0.00s) ✓
- agpaypalTest: Start create buttons if searchstring is in content one time and contains value for amount (0.00s) ✓
- 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 Prüfmethode oder [Assert-Methoden](#). Unsere Klasse erbt diese Prüfmethode.

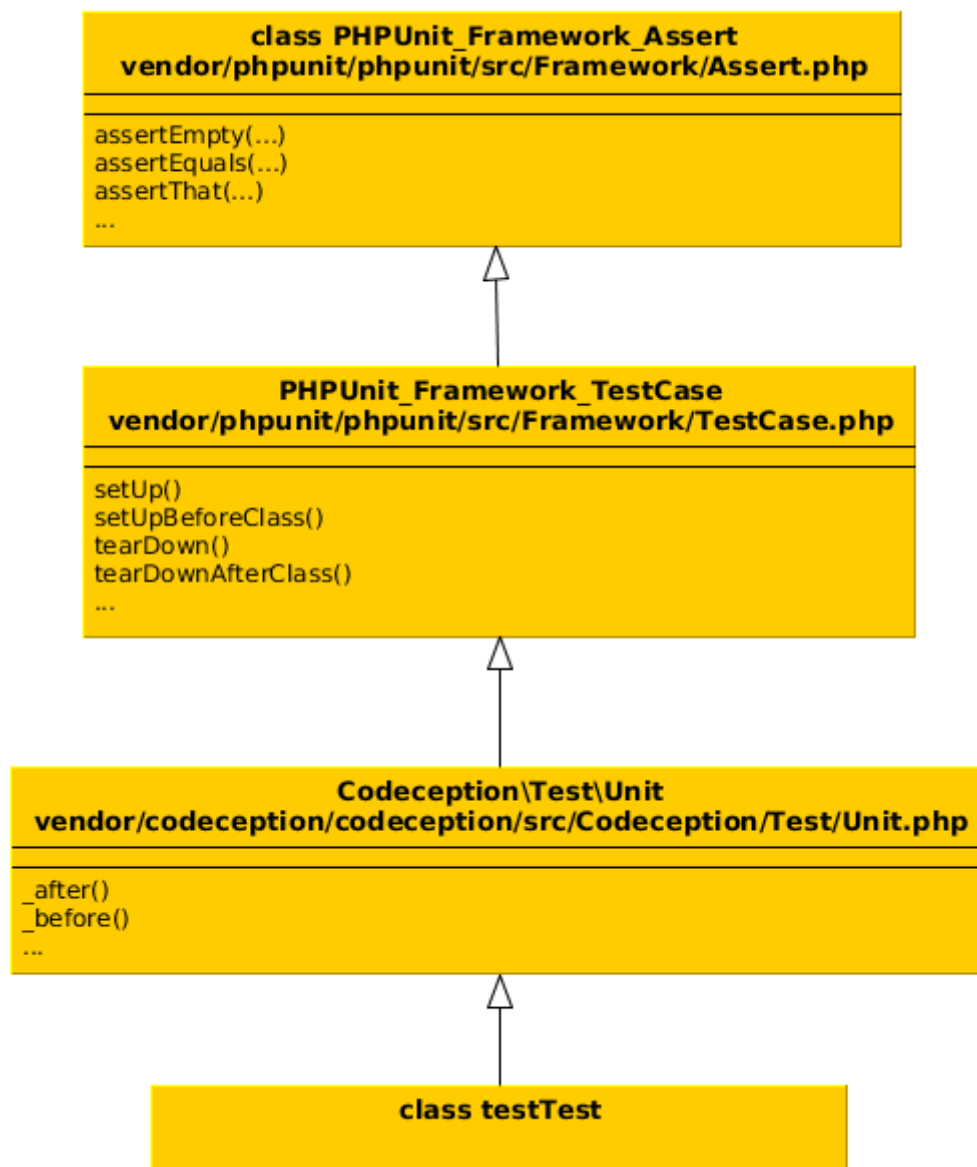


Abbildung 33: Unsere Test-Klasse erbt die Prüfmethode (Assert-Methoden) der Klasse `/joomla/vendor/phpunit/phpunit/src/Framework/Assert.php`. 959a.png

Die Testmethoden der Klasse PHPUnit_Framework_Assert

PHPUnit bietet eine Vielzahl von Prüfmethode(n) (Assert-Methoden), die Sie in Ihren Testfällen verwenden können.

Eine Liste aller Assert-Methoden 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.

RANDBEMERKUNG:

Besonders interessant ist die Funktion [assertThat\(\)](#). Mit ihr können Sie komplexe Assert-Methoden 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 dem Zeichen `/**` beginnen und mit dem 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 Sie im nächsten Kapitel ausprobieren. Hier werden wir die Annotation `@dataProvider` verwenden.

Das erste Testbeispiel verbessern

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

RANDBEMERKUNG:

Bei Problemen sind [Zusatzinformationen](#), die Sie über die Funktion `codecept_debug()` selbst bestimmen können, hilfreich. Diese Informationen können Sie sich in der Konsole anzeigen 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 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 für einen Testfall sichergestellt, dass die Umwandlung des Textmusters `@paypalpaypal@` in einen

PayPal Jetzt-kaufen-Button innerhalb eines Beitrags richtig erfolgt. Gilt dies auch für weitere Testfälle – zum Beispiel für den Fall, dass zwei PayPal Jetzt-kaufen-Buttons in einem Beitrag angezeigt werden sollen?

Erinnern Sie sich? Dass das Testen aller möglichen Eingabeparameter in der Realität unmöglich ist und ein systematisches stichprobenartiges Testen die einzig praktikable Lösung darstellt, 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 Testmethode 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 Test-Methode nacheinander verarbeitet werden.

- Der Lieferant, also der Data Provider, wird als eine Methode in die Testklasse integriert. Diese 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 – beziehungsweise die Methode `provider_PatternToPaypalbutton()`. Dieser Data Provider gibt ein mehrdimensionales Array zurück. Jeder Datensatz des Arrays wird als eigener Testfall von der Methode `testStartCreateButtons()` ausgeführt.

HALTEN WIR 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 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="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 haben wir den Data Provider, beziehungsweise die Methode `provider_PatternToPaypalbutton()`, eingefügt. Diese Methode gibt ein mehrdimensionales Array zurück.
- Als Nächstes haben wir die Annotation `@dataProvider provider_PatternToPaypalbutton` in den Kommentar über unserer Testmethode `testOnContentPrepare()` eingesetzt und dieser Test-Methode 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 wird die Methode sechsmal ausgeführt. Wenn wir den Test erneut starten, 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, eine Testmethode mehrmals mit unterschiedlichen Daten aufzurufen. Der Programmcode einer Methode wird mehrmals wiederverwendet. Dies ist eine große Vereinfachung.

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, da diese den Namen der Test-Methode beinhaltet. Hier können wir 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 in 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. Auch im Programm selbst gibt es Daten und Objekte. Diese sollten während eines Testlaufs einen bekannten Zustand haben. Zum Beispiel Konfigurationsparameter oder Datenbankinhalte. Im nächsten Kapitel geht es um Testduplikate und wie wir mit deren Hilfe eine vordefinierte bekannte Testumgebung aufbauen können.

Die Klasse Fixtures in Codeception

Testduplikate oder künstliche Objekte setzen wir ein, wenn wir in einer definierten bekannten Umgebung einen Testlauf starten 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.

Die Klasse `Fixtures` unterstützen Sie dabei, vor einem Testlauf definierte 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()` starten können, ohne sich darauf verlassen zu müssen, dass andere Werte vorher durch andere Klassen während der Programmausführung richtig gesetzt wurden. Beispielsweise die Variablen `$config`. Das Setzen dieses Wertes ist nicht Bestandteil dieses Tests. Wir testen hier ausschließlich die Methode `onContentPrepare()` der Klasse `PlgContentAgpaypal!`

Was tut die Klasse `Fixtures` genau? Die Klasse `Fixtures` bietet eine statische Variable, die mit einer bestimmten Belegung gespeichert wird. Mit dieser Belegung kann sie mehrmals im Testablauf abgerufen werden. In der Regel wird diese Variable in einer Methode mit Werten belegt, die vor jedem Test in einer Testdatei automatisch abläuft. Zu den automatisch ablaufenden Methoden können Sie im nächsten Abschnitt mehr lesen. Sie können die statische Variable in der Klasse `Fixtures` auch in einer der Konfigurationsdateien mit Werten belegen. In Codeception gibt es mehrere Konfigurationsdateien, die `_bootstrap.php` heißen. Welche Sie hierfür wählen sollten, können Sie im Kapitel *Codeception – ein Überblick | Codeception – ein erster Rundgang | Codeception unter die Lupe genommen | Konfiguration* nachlesen.

Als Nächstes ändern wir unser Beispiel so ab, dass wir für die Variable `$config` in unserer Testdatei die Klasse `Fixtures` 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` über die statische Methode `Fixtures::add(...)` in der Klasse `Fixtures` gespeichert und später über die Methode `Fixtures::get(...)` abgerufen.

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 die Werte in der Klasse `Fixtures` überschreiben, indem sie dieses erneut speichern. Geben Sie beispielsweise die Anweisung

```
Fixtures::add('benutzer', ['name' => 'Peter']);
```

in einer Testdatei ein und in der nächsten Testdatei führen Sie

```
Fixtures::add('benutzer', ['name' => 'Paul']);
```

aus, dann ist ab nun der Wert `Paul` in der Variablen gespeichert und der Befehl

```
Fixtures::get('benutzer');
```

gibt den Benutzer **Paul** zurück.

Sie sehen schon. Die Verwendung der Klasse `Fixtures` sollte geplant werden. Schon allein deshalb ist es sinnvoll, die statischen Methoden der Klasse `Fixtures` nur an bestimmten Stellen zu verwenden. 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.

RANDBEMERKUNG:

Allgemein gilt, dass alle Testfälle einer Testklasse die Klasse `Fixtures` gemeinsam nutzen sollten. Hat eine Testmethode für Daten, die in der Klasse `Fixtures` gespeichert sind keine 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 produktiv Klasse mehrere korrespondierende Testklassen existieren. Jede von diesen besitzt ihre individuellen Daten in der Klasse `Fixtures`.

Vor dem Test den Testkontext herstellen und hinterher aufräumen

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 das Aufräumen hinterher 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 bestimmte Methoden automatisch in einer festgelegten Reihenfolge ausgeführt.

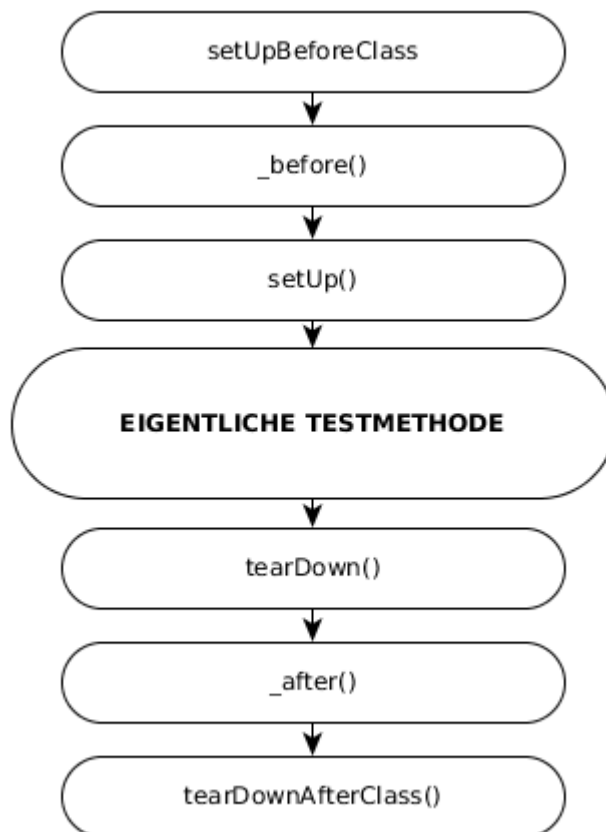


Abbildung 34: Vor und nach jedem Test werden die abgebildeten Methoden automatisch in der angezeigten Reihenfolge ausgeführt. 993.png

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 effektiv einzusetzen, müssen Sie wissen, wie Sie effektive Testfälle schreiben. Unter anderem sollten Sie die Werte in der Klasse Fixtures und die Testduplikate nicht im Konstruktor einer Testklasse initialisieren. Wenn Sie objektorientiert programmieren, ist dies vielleicht Ihr erster Gedanke. Sinnvoller ist es aber, die in PHPUnit und Codeception 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()` der Klasse `Codeception\Test\Unit` erweitert Codeception das PHPUnit Framework. Alle anderen oben aufgeführten Methoden sind Standard PHPUnit Methoden. Auf diese Weise können Sie über die Klasse `Codeception\Test\Unit` alle Funktionen der verschiedenen Codeception Module und Hilfsklassen, zusätzlich zu allen PHPUnit Funktionen verwenden!

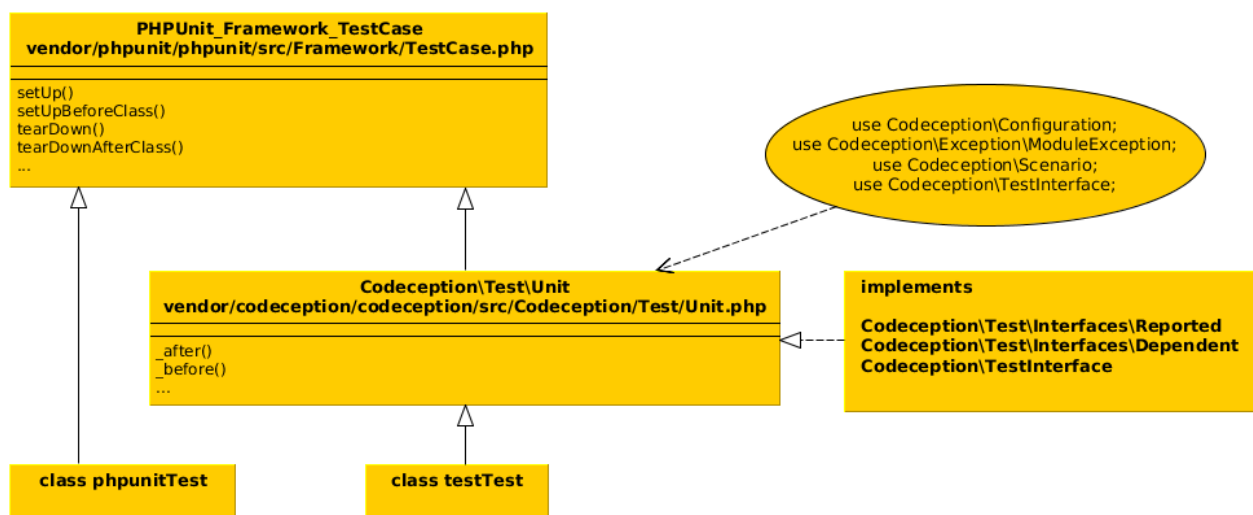


Abbildung 35: Über die Klasse `Codeception\Test\Unit` können alle Funktionen der verschiedenen Codeception Module und Hilfsklassen, zusätzlich zu allen PHPUnit Funktionen, verwendet werden. 959c.png

Im Falle von PHPUnit Tests werden Sie sicherlich am häufigsten die Methoden `setUp()` und `tearDown()` verwenden. Möchten Sie die Hilfsklassen oder Module von Codeception nutzen, sind die Methoden `_setUp()` und `_after()` die richtigen für Sie. Später in diesem

Kapitel werden Sie die Klasse Fixtures über die Methode `_setup()` in Ihren Tests mit Werten belegen. An dieser Stelle 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 zugreifen, 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 ansehen die vor und nach jedem Test ablaufen – darauf vor, mit deren Hilfe unsere 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 Objekt `$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 Daten in der Klasse `Fixtures` in einem wohldefinierten bekannten Zustand verwenden können.

Kurzgefasst

Hauptthema dieses Kapitels war die testgetriebene Softwareentwicklung und die Erstellung von Unittests. Außerdem wissen Sie nun, wie Codeception PHPUnit integriert und erweitert. Ein erstes Beispiel für ein Testduplikat haben Sie mit der Klasse `Fixtures` auch schon kennen gelernt. Dieses Thema vertiefen wir im nächsten Kapitel.

Testduplikate

Der Test ist der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen.

[[Ernst Denert](#)]

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 herausgearbeitet. 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 Stub-Objekten (Stubs) und Mock-Objekten (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. Daten, die in der Klasse Fixtures gespeichert sind, 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 Fixtures realisiert haben, ist ein optionales Array, in dem Konfigurationseinstellungen gespeichert sind. Daten, die in der Klasse Fixtures gespeichert sind, sollen komplexe Abläufe oder Berechnungen in der Anwendung ersetzen, indem Sie das Ergebnisobjekt künstlich zur Verfügung stellen. Das heißt, wir belegen die Werte im Array einfach mit den Konfigurationseinstellungen, die wir testen möchten!

RANDBEMERKUNG:

Für Testduplikate gibt es unterschiedliche Bezeichnungen. [Martin Fowler](#) unterscheidet Dummy-Objekte, Fake-Objekte, Stub-Objekte 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 in Codeception, im Gegensatz zu den Daten der Klasse Fixtures, reale Objekte ersetzen.

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 oder Mock-Objekten ersetzt 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 Objekten, die es auch in der Applikation Joomla! gibt. Ich meine die Objekte \$row und \$params. \$row instanziiert die PHP eigene Klasse stdClass und \$params instanziiert die Klasse \JRegistry.

RANDBEMERKUNG:

JRegistry finden Sie in Joomla! in der Datei [joomla/libraries/vendor/joomla/registry/src/Registry.php](https://github.com/joomla/joomla-cms/blob/master/libraries/vendor/joomla/registry/src/Registry.php). Diese wird über die Datei [/joomla/libraries/classmap.php](https://github.com/joomla/joomla-cms/blob/master/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. Lassen Sie sich vom Namen der Methode nicht verwirren. Momentan verwenden wir diese – mit der Methode `getMockBuilder()` erstellten – Objekte ausschließlich als Stubs!

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`. Die Methode [getMockBuilder\(\)](#) 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 befindet sich 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 Programmcode 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 mit einem Wert belegen. Ich gebe zu, in unserem Falle ist dies ein sehr 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. Hier im Buch soll dies als Einführung genügen. Wenn Sie praktisch mit Testduplikaten arbeiten, werden Sie sicherlich weitere Funktionen benötigen. In englischer Sprache werden alle möglichen Funktionen in der [Dokumentation von PHPUnit](#) erläutert.

RANDBEMERKUNG:

Wie könnte es anders sein? Codeception bietet Ihnen eine Klasse, die das Erstellen von Stub-Objekten vereinfacht, nämlich die Klasse `\Codeception\Util\Stub`. Wenn Sie lieber diese Klasse verwenden möchten, können Sie obiges Beispiel wie folgt abändern. Das Endergebnis bleibt aber das Gleiche und ich habe diese Klasse deshalb bewusst in eine Randbemerkung ausgelagert.

```
<?php
namespace suites\plugins\content\agpaypal;
use \Codeception\Util\Fixtures;
use \Codeception\Util\Stub;
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 = Stub::make('stdClass');
        $this->params = Stub::make('JRegistry');
    }
    ....

```

Ein Mock-Objekt

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 – unabhängig davon, ob die Variabel `$row` ein Objekt ist oder nicht.

```

...
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 – das heißt, fügen Sie die Zeichen `//` an den Beginn der Zeile `$this->startCreateButtons($row);`.

```

<?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 Einfügen der Zeichen // an den Beginn 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 dem Objekt übergeben. Bei einem erneuten 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:testOnContentPrepareMethodRunsOneTime
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 Mocks von Stubs

Nun haben Sie unterschiedliche Testduplikate praktisch kennengelernt. Das der Begriff Fixtures – schwammig definiert – keine realen Arbeitsobjekte beschreibt und Daten

der Klasse Fixtures sich so von Mocks und Stubs unterscheiden, habe ich schon abgegrenzt. Den Unterschied zwischen Mock-Objekten und Stub-Objekten haben wir aber noch nicht geklärt.

In einem Satz kann ich es so beschreiben: Stub-Objekte prüfen den **Status** einer Anwendung und Mock-Objekte 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 Tests 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 Status nach dem Ausführen der Funktion.
4. Stellen Sie den Ausgangszustand der Anwendung wieder her. Dies erfolgt in der Regel in der 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 Status nach dem Ausführen der Funktion.
6. Stellen Sie den Ausgangszustand der Anwendung wieder her. Dies erfolgt in der Regel in der Methode `_after()`.

Sowohl Mocks als auch Stubs geben eine Antwort auf die Frage: Was ist das Ergebnis.
--

Im Falle von Mocks spielt zusätzlich eine Rolle, **wie** das Ergebnis erreicht wurde.

BDD Spezifikationen

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

Codeception unterstützt Sie mit den Erweiterungspaketen [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. Dieses Merkmal der verhaltensgetriebenen Softwareentwicklung hatte ich schon im Kapitel *Softwaretests – eine Einstellungssache? | Behaviour-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 nicht optimal ist, sind diese sehr zahlreich (und leider noch überwiegend in einer veralteten PHPUnit Version geschrieben).

Was meinen Sie was passiert, wenn beim Integrieren einer neuen Funktion die notwendige Programmcodeänderung Fehler bei der Ausführung eines Tests 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 den Programmcode des Tests, der nun fehlschlägt. Wahrscheinlich wird der fehlerhafte Test 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 Teammitglied 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 dieses nicht verstehen, wird der gute Code am Ende „Wegwerfcode“ sein und überdies wird es im Team viel Frust geben.

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

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

Codeception Werkzeuge Specify und Verify

Die 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 einer besser lesbaren Form geschrieben und erfüllen so die Merkmale des Behaviour Driven Development.

Achten Sie darauf, dass Sie das Paket Codeception\Specify *innerhalb* der Klassendefinition mit dem use-Operator inkludieren, 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 einen Data Provider mit Specify nutzen möchten, ist es wichtig zu wissen, dass Sie diesen nicht beliebig benennen 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? Noch 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 mithilfe des Einfügens der Zeichen `//` auskommentiert und an deren Stelle die Methode `verify(...)` eingesetzt. So können Sie die unterschiedlichen Versionen vergleichen:

```
<?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 gegebenenfalls die ausführliche [Dokumentation auf Github](#), wenn Ihnen `verify()` gefällt und Sie das Paket einsetzen möchten.

Kurzgefasst

In diesem Kapitel ging es in erster Linie darum, wie Objekte, die in einer Abhängigkeitsbeziehung zum Testobjekt stehen, simuliert werden können. Bei Unittests geht es darum, kleine Programmeinheiten unabhängig voneinander zu

testen. Deshalb sind Testduplikate beim Erstellen von Unittests unentbehrlich. Im nächsten Abschnitt werden wir uns das Zusammenspiel der einzelnen Units genauer ansehen.

Funktionstest

Thema dieses Kapitels sind Funktionstests. Im den vorhergehenden Kapiteln haben wir eine selbst erstellte Joomla! Erweiterung als einzelne Einheit mithilfe von Unittests getestet. Dass unser Test-Plugin korrekt arbeitet, konnten wir mit Unittests belegen. Die Zusammenarbeit mit anderen Joomla! Klassen haben wir aber noch nicht getestet. Wird unserem Plugin der Beitragstext in der Variablen `$row` überhaupt richtig übergeben? Diese Variable haben wir bisher in unserm Testcode selbst erzeugt und mit einem Wert belegt! Oder: Verarbeiten alle anderen Joomla! Klassen die Daten, die unsere Erweiterung umwandelt, richtig weiter? Die bisherigen Tests waren unabhängig von anderen Programmteilen. In diesem Kapitel werden wir das Zusammenspiel von mehreren Einheiten genauer unter die Lupe nehmen.

Tauchen Sie mit mir in diesem Kapitel 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. Auf diesem Beispiel bauen wir in diesem Kapitel auf. Haben Sie beim Erstellen der Beispieltests Änderungen an diesem Aufbau vorgenommen? Dann stellen Sie am besten den Zustand, den wir im Kapitel *Praxisteil: Die Testumgebung einrichten | Joomla! mit einem eigenen Plugin erweitern* aufgebaut haben, wieder her.

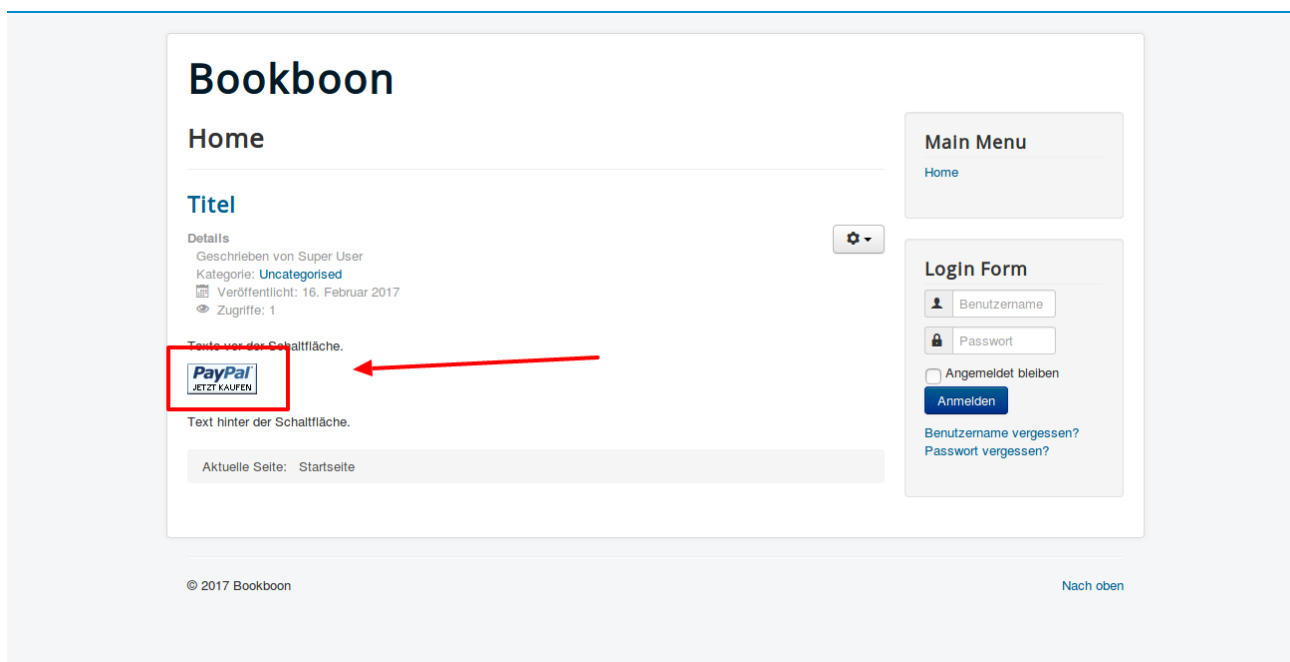


Abbildung 36: Unsere Website enthält bisher nur einen einzigen Beitrag und der PayPal Jetzt-kaufen-Button wird sofort auf der Startseite angezeigt. 980a.png

Ein erstes Beispiel

Den ersten Funktionstest generieren

Im Kapitel *Codeception – ein Überblick* hatte ich die wichtigsten codecept Befehle erklärt. Eigentlich ist der Befehl `vendor/bin/codecept generate:cept functional`

`/suites/plugins/content/agpaypal/agpaypal`, mit dem Sie ein Codefragment ([Boilerplate](#)) für einen Funktionstest erstellen können, aber selbsterklärend. Mit diesem Befehl wird die Datei `agpaypalCept.php` im Verzeichnis

`/var/www/html/joomla/tests/functional//suites/plugins/content/agpaypal/` angelegt.

RANDBEMERKUNG:

Ich habe hier der Einfachheit halber eine CEPT-Datei erstellt. Wenn Sie lieber mit CEST-Dateien arbeiten möchten, können Sie dies gerne tun. Sehen Sie sich dann das Kapitel *Codeception – ein Überblick | Codeception – ein erster Rundgang | Mit Codeception Tests organisieren und erweitern* noch einmal an.

```
/var/www/html/joomla$ vendor/bin/codecept generate:cept functional
/suites/plugins/content/agpaypal/agpaypal
Test was created in
/var/www/html/joomla/tests/functional//suites/plugins/content/agpaypal/agpaypalCept.php
```

Die Datei enthält in dieser automatisch generierten Version zwei Zeilen.

```
<?php
$I = new FunctionalTester($scenario);
$I->wantTo('perform actions and see result');
```

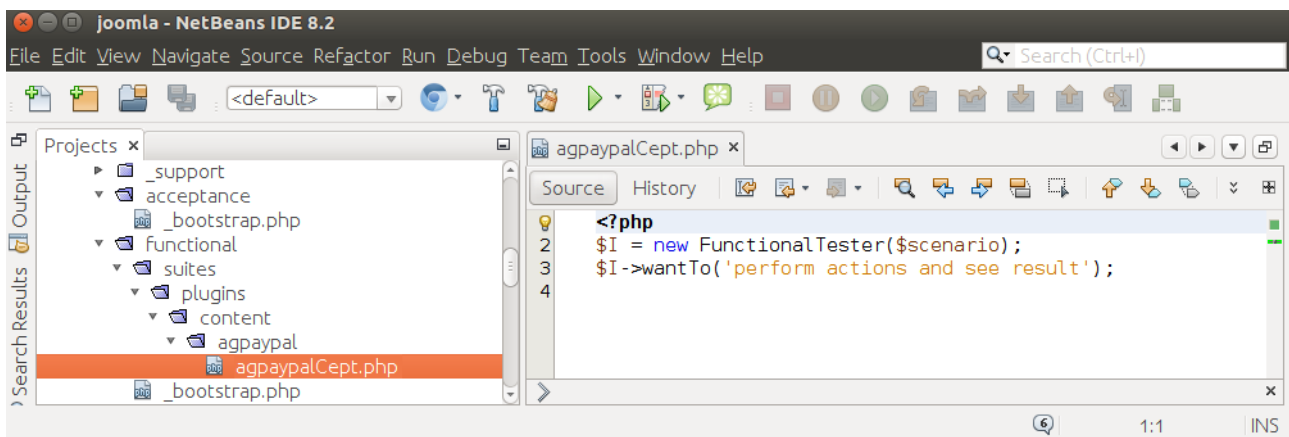


Abbildung 37: Die mit dem Befehl vendor/bin/codecept generate:cept functional /suites/plugins/content/agpaypal/agpaypal angelegte Datei enthält zwei Zeilen. Sie sehen diese Datei hier geöffnet in der Entwicklungsumgebung Netbeans. 973.png

Ich habe den Text „perform actions and see result“ in der automatisch erstellen Datei in „Ich will sicherstellen, dass eine PayPal Schaltfläche angezeigt wird.“ geändert und den Test ausgeführt.

```
:/var/www/html/joomla$ vendor/bin/codecept run functional
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.15 by Sebastian Bergmann and contributors.
Functional Tests (1) -----
✓ agpaypalCept: Ich will sicherstellen, dass eine paypal schaltfläche angezeigt wird. (0.00s)
-----
Time: 156 ms, Memory: 8.00MB
OK (1 test, 0 assertions)
```

Alles verlief erfolgreich. Bisher passiert auch noch nicht viel. Sehen wir uns die beiden Codezeilen der automatisch generierten Datei trotzdem einmal genauer an.

Der Text `$I = new FunctionalTester($scenario);` in der ersten Zeile instanziiert den Akteur – also den fiktiven Tester. Dieser heißt in meiner Standardkonfiguration `FunctionalTester`

– das bedeutet die Klasse heißt `FunctionalTester`. Sie können dem Tester auch einen anderen Namen geben. Den Namen des Testers legen Sie in der Konfigurationsdatei `/var/www/html/joomla/tests/functional.suite.yml` fest. Wissen Sie noch wo Sie die Konfigurationsdatei finden? Falls nicht, lesen dies Sie im Kapitel *Codeception – ein Überblick | Codeception – ein erster Rundgang | Codeception unter die Lupe genommen | Konfiguration* nach.

Der Text `$I->wantTo('Ich will sicherstellen, dass eine PayPal Schaltfläche angezeigt wird.');` in der zweiten Zeile ist optional. Hier wird der Testfall mit einem Satz mithilfe der Methode `wantTo()` beschrieben. Dieser Methodenaufruf hilft Projektmitarbeitern den Testfall leichter und schneller zu verstehen.

Die Methode `wantto()` sollte nur einmal pro Testfall aufgerufen werden. Ein weiterer Aufruf würde den vorherigen Aufruf überschreiben. Bei einem Test der `wantto('Ich will A')` und `wantto('Ich will B')` enthält würde nur 'Ich will B' ausgegeben werden. Probieren Sie es selbst aus:

```
<?php
$I = new FunctionalTester($scenario);
$I->wantTo('Ich will A');
$I->wantTo('Ich will B');
```

```
/var/www/html/joomla$ vendor/bin/codecept run functional
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.17 by Sebastian Bergmann and contributors.
Functional Tests (1) -----
✓ agpaypalCept: Ich will b (0.00s)
-----
Time: 182 ms, Memory: 10.00MB
OK (1 test, 0 assertions)
```

Konfiguration

Wie geht es nun weiter? Wie gehen Sie am besten vor, um den automatisch generierten Test so zu erweitern, dass er die korrekte Darstellung des PayPal Jetzt-kaufen-Buttons im Zusammenspiel mit den anderen Komponenten des Content Management Systems Joomla! sicherstellt? Am besten sehen wir uns dazu als Erstes die Module an, mit denen Codeception das Erstellen von Funktionstests unterstützt.

Die Methode `wantto()` konnten wir nutzen, weil Codeception diese über die Klasse `FunctionalTester` in der Klasse `Actor` zur Verfügung stellt. Vielleicht gibt es ja noch mehr Verwendbares in dieser Klasse? Sie finden die Klasse `FunctionalTester` im Verzeichnis `/var/www/html/joomla/tests/_support/` in der Datei `FunctionalTester.php`.

Wie Sie im nachfolgenden Programmcodeausschnitt sehen verwendet die Klasse `FunctionalTester` zusätzlich den Trait `FunctionalTesterActions`.

```
<?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($description)
 * @method \Codeception\Lib\Friend haveFriend($name, $actorClass = NULL)
 *
 * @SuppressWarnings(PHPMD)
 */

class FunctionalTester extends \Codeception\Actor {
    use _generated\FunctionalTesterActions;
    /**
     * Define custom actions here
     */
}
```

Den Programmcode der Klasse `FunctionalTesterActions` können Sie sich im Verzeichnis `/var/www/html/joomla/tests/_support/_generated/` in der Datei `FunctionalTesterActions.php` ansehen.

```
<?php
namespace _generated;

// This class was automatically generated by build task
```

```
// You should not change it manually as it will be overwritten on next build
// @codingStandardsIgnoreFile
use Helper\Functional;
trait FunctionalTesterActions{
    /**
     * @return \Codeception\Scenario
     */
    abstract protected function getScenario();
}
```

Die Auswahl ist noch nicht groß. Sie wird aber größer, wenn Sie in der Konfiguration weitere Module hinzufügen. Fügen Sie nun bitte das Modul PhpBrowser in der Konfigurationsdatei functional.suite.yml hinzu. Sie finden diese Datei im Verzeichnis `/var/www/html/joomla/tests/`

```
class_name: FunctionalTester
modules:
    enabled:
        - \Helper\Functional
        - PhpBrowser:
            url: 'http://localhost/joomla'
```

Damit alle Aktionen, die Codeception mit diesem Modul bietet, in den Trait FunctionalTesterActions integriert werden, müssen Sie den Befehl `vendor/bin/codecept build` ausführen.

```
/var/www/html/joomla$ vendor/bin/codecept build
Building Actor classes for suites: unit, acceptance, functional
-> UnitTesterActions.php generated successfully. methods added
\UnitTester includes modules: Asserts, \Helper\Unit
-> AcceptanceTesterActions.php generated successfully. methods added
\AcceptanceTester includes modules: PhpBrowser, \Helper\Acceptance
-> FunctionalTesterActions.php generated successfully. methods added
\FunctionalTester includes modules: PhpBrowser, \Helper\Functional
```


RANDBEMERKUNG:

Werden in Ihrer Installation keine weiteren Aktionen in den Trait `FunctionalTesterActions` eingefügt? Höchstwahrscheinlich enthält eine Ihrer Konfigurationsdateien oder eine der Helper-Dateien einen Syntaxfehler. In diesem Fall bricht der Build-Prozess, ohne einen Fehler zu melden, einfach ab.

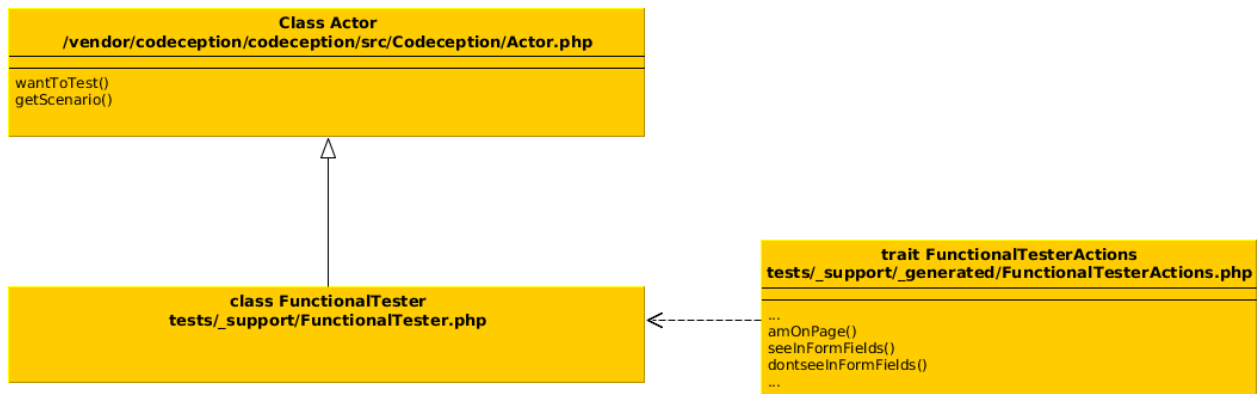


Abbildung 38: Die Methode `wantto()` konnten wir nutzen, weil Codeception diese über die Klasse `FunctionalTester` in der Klasse `Actor` zur Verfügung stellt. 954.png

Sehen Sie sich nun die automatisch generierte Methoden im Trait `tests/_support/_generated/FunctionalTesterActions.php` noch einmal an. Der Trait enthält aufgrund des Hinzufügens des Moduls `Codeception\Module\PhpBrowser` sehr viele neue Methoden, die Sie in Ihren Tests nutzen können. Praktisch, oder?

Den Test ausbauen

Die im Praxisteil erstellte Schaltfläche manuell testen

So nun kann es losgehen. Wir wollen sicherstellen, dass der PayPal Jetzt-kaufen-Button mit dem richtigen Betrag angezeigt wird. In unserer aktuellen Installation startet Joomla! das Frontend so, dass der Beitrag mit dem PayPal Jetzt-kaufen-Button sofort auf der Startseite erscheint. Wir hatten ihn ja als Haupteintrag markiert.

Sehen Sie sich den Inhalt dieses Beitrages zur Sicherheit noch einmal an. Öffnen Sie dazu im Administrationsbereich das Menü `Inhalt | Beiträge`. Wählen Sie dann den im Praxisteil *Praxisteil: Die Testumgebung einrichten | Joomla! mit einem eigenen Plugin erweitern* erstellten Beitrag. Wenn Sie meinem Beispiel gefolgt sind, sehen Sie nur diesen einen Beitrag mit dem einfallsreichen Titel „Titel“.

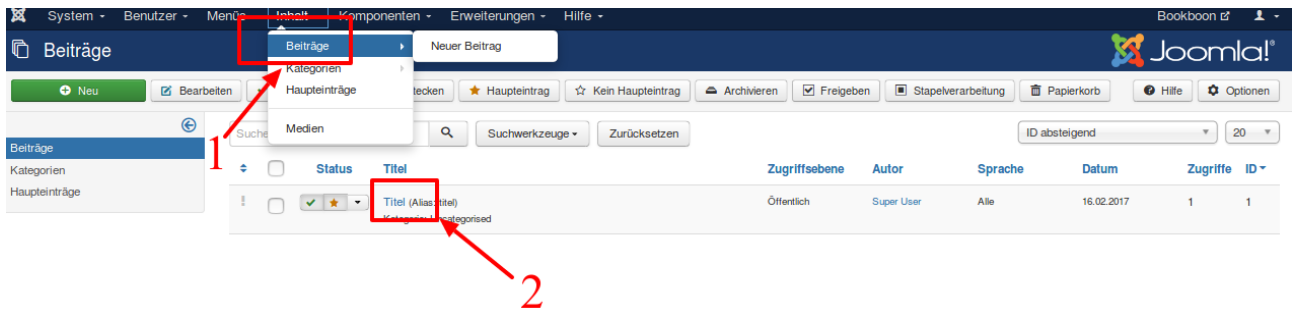


Abbildung 39: Das Menü Inhalt | Beiträge im Administrationsbereich von Joomla!. 972.png

Den Inhalt des Beitrags können Sie sich ansehen, indem Sie im Menü Inhalt | Beiträge den Titel anklicken. Der Inhalt des Beitrages sollte das Muster @paypalpaypal@ enthalten.

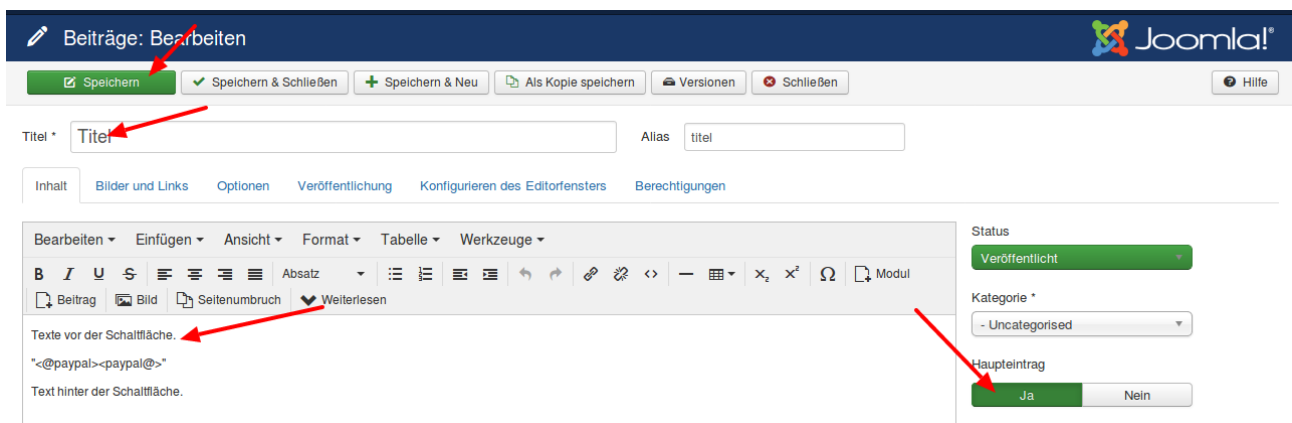


Abbildung 40: Der Inhalt des Beitrages sollte das Muster @paypalpaypal@ enthalten. 971.png

Über die URL <http://localhost/joomla/> sollte der PayPal Jetzt-kaufen-Button zu sehen sein. Da kein Betrag im Muster angegeben wurde, sollte der PayPal Jetzt-kaufen-Button mit dem Wert 12,99 Euro belegt sein. Dies hatten wir im Plugin als Standard für den Fall, dass kein spezieller Betrag gewünscht ist, so programmiert.

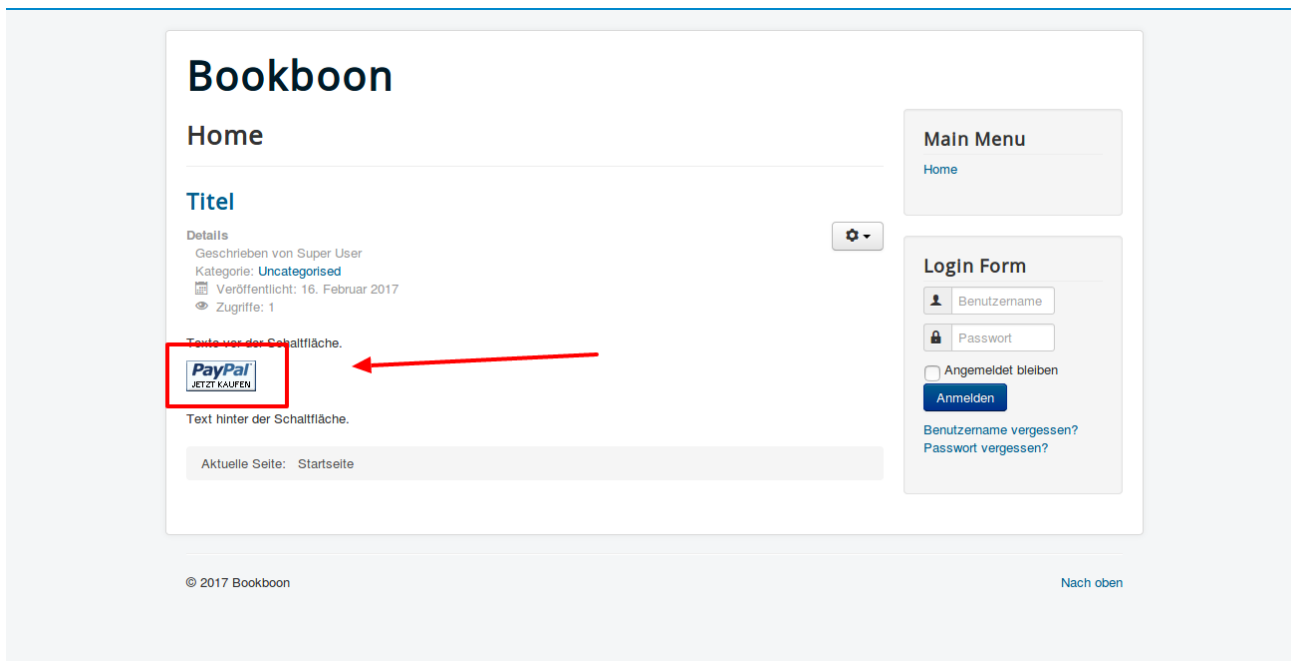


Abbildung 41: Der PayPal Jetzt-kaufen-Button wird angezeigt und ist mit dem Wert 12,99 Euro belegt. 970.png

Ein Blick in den Quellcode zeigt, dass alles richtig ist.

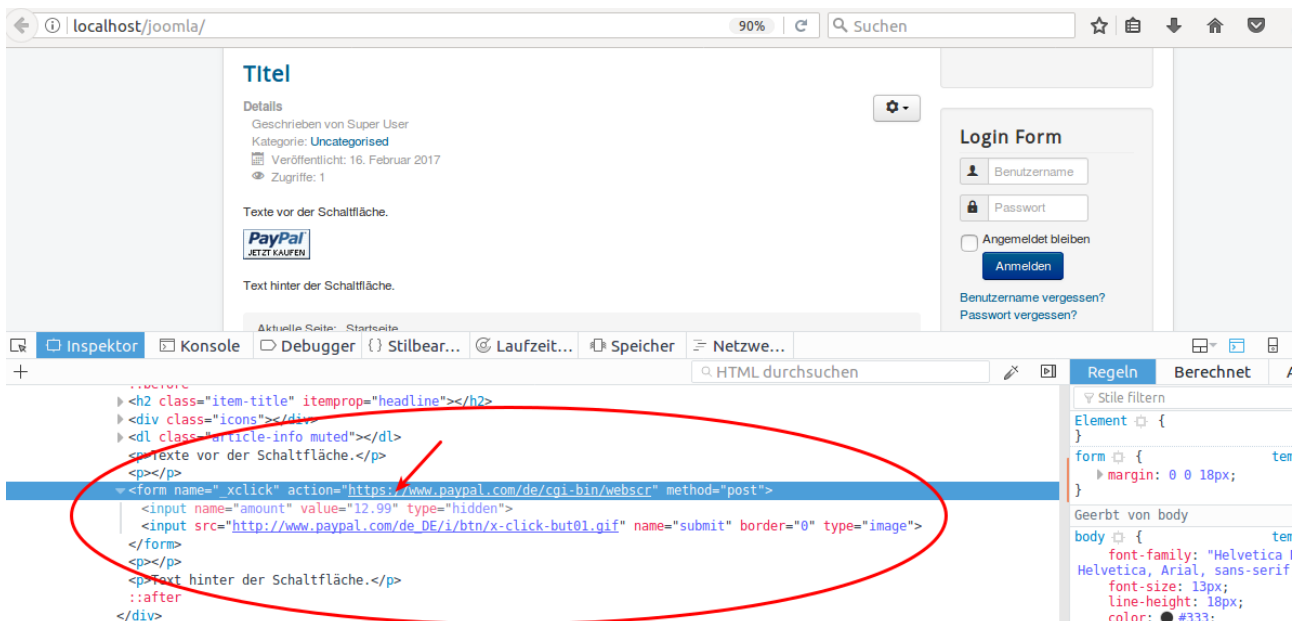


Abbildung 42: Ein Blick in den Quellcode zeigt, dass der PayPal Jetzt-kaufen-Button tatsächlich mit dem Wert 12,99 Euro belegt ist. 953.png

RANDBEMERKUNG:

Den Quelltext können Sie sich beispielsweise im Internetbrowser Mozilla Firefox über den [Inspektor](#) ansehen. Falls Sie lieber mit dem Internetbrowser Google

Chrome arbeiten, können sie sich den Quelltext mithilfe der [Entwicklertools](#) ansehen.

So, manuell haben wir den Test nun durchgeführt. Das möchten wir aber nicht nach jeder Änderung selbst von Hand erledigen. Dieser Test soll automatisch ohne menschliches Zutun ablaufen!

Den Testablauf automatisieren

Das Automatisieren des eben manuell durchgeführten Testablaufs ist nach dem Hinzufügen des Moduls PhpBrowser relativ einfach. PhpBrowser bietet unter anderem die Methoden `amOnPage()` und `seeInFormFields()`. Diese Methoden können Sie über den Trait `FunctionalTesterActions` verwenden. Diesen Trait finden Sie in der Datei `tests/_support/_generated/FunctionalTesterActions.php`.

Ich habe die beiden Methoden `amOnPage()` und `seeInFormFields()` in unsere Testdatei eingefügt. Im nachfolgenden Programmcodebeispiel können Sie diese sehen.

```
<?php
$I = new FunctionalTester($scenario);
$I->wantTo('Ich will sicherstellen, dass eine PayPal Schaltfläche angezeigt wird.');
```

\$I->amOnPage('http://localhost/joomla');

\$I->seeInFormFields('form[name=_xclick]', [

'amount' => '12.99',

]);

Die Zeile `$I->amOnPage('http://localhost/joomla');` ist selbsterklärend. Mit dem Aufruf `$I->amOnPage('http://localhost/joomla');` öffnet sich eine Verbindung zur URL `http://localhost/joomla`, also zu derjenigen URL die als Parameter in der Methode mitgegeben wird.

Die Methode [seeInFormFields\(\)](#) ist etwas komplizierter. Diese Methode erwartet zwei Parameter. Zum einen den Namen des Formulars in der Syntax `form[name=_xclick]` und zum anderen einen Array mit den zu testenden Feldnamen in Kombination mit den zu erwarteten Belegungen dieser Felder – `['amount' => '12.99']`. Das Formular, mit dem der PayPal Jetzt-kaufen-Button angezeigt wird, hat das Attribut `name="_xclick"` und sollte ein Feld mit dem Namen `amount` und dem Wert `12.99` enthalten:

```

...
<form name="_xclick" action="https://www.paypal.com/de/cgi-
bin/webscr" method="post">
<input type="hidden" name="amount" value="12.99">
...
</form>;
...

```

Sehen wir uns nun an, ob der Funktionstest erfolgreich verläuft. Analog zu dem Befehl `vendor/bin/codecept run unit` bei den PHPUnittests, starten Sie alle Funktionstests mit dem Befehl `vendor/bin/codecept run functional`.

```

/var/www/html/joomla$ vendor/bin/codecept run functional
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.15 by Sebastian Bergmann and contributors.
Functional Tests (1) -----
✓ agpaypalCept: Ich will sicherstellen, dass eine paypal schaltfläche angezeigt wird. (0.07s)
-----
Time: 232 ms, Memory: 10.00MB
OK (1 test, 1 assertion)

```

Der Test ist erfolgreich. Was passiert, wenn Sie den Testprogrammcode auf die Prüfung eines anderen Betrag abändern? Probieren Sie es aus, ändern Sie den eben erstellten Test und suchen Sie nach einer Schaltfläche, die als Geldbetrag 16.00 enthält.

```

$I->seeInFormFields('form[name=_xclick]', [
    'amount' => '16.00',
]);

```

Eine solche Schaltfläche gibt es auf der Website nicht. Ihnen wird ein Fehler gemeldet. Wohingegen die Prüfung, ob genau diese Schaltfläche nicht vorkommt, wieder bestanden wird:

```
$I->DontSeeInFormFields('form[name=_xclick]', [
    'amount' => '16.00',
]);
```

Genau diese negative Prüfung könnte auch irgendwann einmal wichtig für das Verhalten Ihrer Webanwendung sein und eines Tests bedürfen.

Wenn Sie eine Webanwendung testen, ist die Angabe eines Selektors in vielen Methoden Pflicht. Codeception greift Ihnen bei der Suche nach dem Selektor so gut wie möglich unter die Arme. In vielen Methoden, zum Beispiel in der Methode [seeInField\(\)](#) können Sie den Selektor auf unterschiedliche Art angeben. Möchten Sie in diesem Kapitel schon Methoden ausprobieren, die Selektoren als Eingabeparameter voraussetzen? Dann blättern Sie bitte zum nächsten Kapitel vor und lesen, wie Sie in Codeception am besten Selektoren nutzen. Sie finden den relevanten Teil im Kapitel *Akzeptanztests | Implementatierung der Testmethoden | Automatisch generierte Methoden nutzen*. Wir machen hier nun aber erst mit unserem Beispiel weiter.

Was tun, wenn etwas schief läuft?

Schlägt einer Ihrer Tests fehl und Sie haben keine Erklärung dafür? Sehen Sie sich dann einmal den Inhalt im Verzeichnis `/var/www/html/joomla/tests/_output` an. Schlägt ein Test fehl, wird hier nämlich das HTML-Dokument, das Grundlage des Tests war, abgelegt. So können Sie in Ruhe die Fehlerursache suchen.

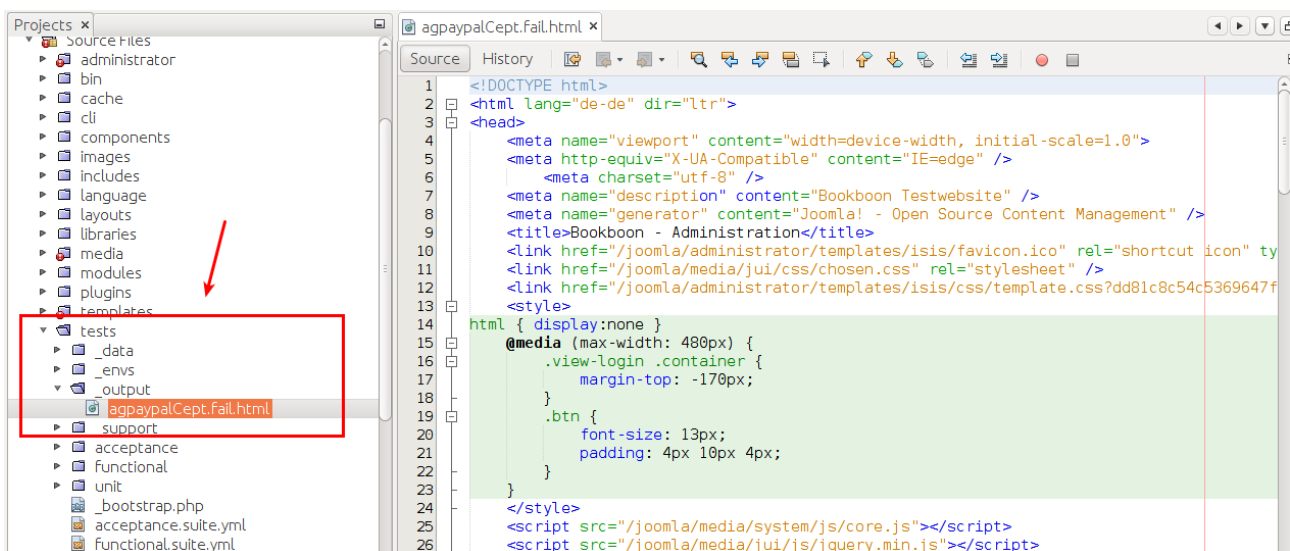


Illustration 43: Ein Test schlägt fehl? Im Verzeichnis `_output` finden Sie Informationen zur Fehlerursache. 967.png

Wir haben etwas vergessen

Bei Erstellen des Tests im vorhergehenden Abschnitt haben Sie sich vielleicht gewundert. Ich hatte schon an mehreren Stellen geschrieben, dass wir unabhängige Tests schreiben sollten. Dies gilt natürlich auch für Funktionstests. Der soeben erstellte Test kann aber nur fehlerfrei durchlaufen, wenn ein bestimmter PayPal Jetzt-kaufen-Button vorher von Ihnen manuell erstellt wurde und in einem Haupteintrag auf der Startseite erscheint. Genau dieses müsste eigentlich vorher beim Aufbau des Testkontextes sichergestellt werden – es dürfte nicht im Test vorausgesetzt werden.

Eine Möglichkeit den passenden Kontext zu schaffen ist es, den Test so zu erweitern, dass während des Tests ein Joomla! Beitrag mit Schaltfläche angelegt wird und dieser dann im weiteren Verlauf getestet wird. Zum Anlegen eines Beitrags ist eine erfolgreiche Anmeldung im Backend die erste Voraussetzung. Das Testen des Anmeldeformulars ist ein gutes Beispiel für Formulartests. Beginnen wir also den Aufbau des unabhängigen Tests, indem wir zu Beginn des Tests das Anmeldeformular ausfüllen und absenden. Wir testen also zunächst das Anmeldeformular für den Administrationsbereich, das Sie über die URL <http://localhost/joomla/administrator> öffnen können.



Illustration 44: Anmeldeformular zum Joomla! Administrationsbereich 968.png

Sehen Sie sich das – auf die wesentlichen Bereiche reduzierte – HTML-Element zum Anmeldeformular kurz an, bevor sie den Test für dieses Formular angehen.

```
...
<form action="/joomla/administrator/index.php" method="post" id="form-login">
<input id="mod-login-username"/>
<input id="mod-login-password"/>
<button>Anmelden</button>
</form>
...
```

Die Methode `amOnPage()` haben Sie im vorhergehenden Beispiel schon verwendet. Im nächsten Teil werden Sie die Methoden `fillField()`, `click()` und `see()` kennenlernen.

```
<?php
$I = new FunctionalTester($scenario);
$I->wantTo('Ich will sicherstellen, dass die Anmeldung im Administrationsbereich funktioniert.');
```

```
$I->amOnPage('http://localhost/joomla/administrator');
$I->fillField(['id' => 'mod-login-username'], 'admin');
$I->fillField(['id' => 'mod-login-password'], 'admin');
$I->click('Anmelden');
$I->see('Kontrollzentrum', ['class' => 'page-title']);
```

Was testen wir genau? Zunächst stellen wir eine Verbindung zur URL `http://localhost/joomla/administrator` her. Dann füllen wir die, für eine erfolgreiche Anmeldung, notwendigen Felder. Im Anschluss klicken wir auf die Schaltfläche, die mit dem Text *Anmelden* beschriftet ist. Um sicherzustellen, dass alles geklappt hat, also dass der Test erfolgreich war, stellen wir sicher, dass wir uns nun im Kontrollzentrum befinden. Wir sind sicher im Kontrollzentrum angekommen, wenn wir den Text *Kontrollzentrum* auf der Website sehen.

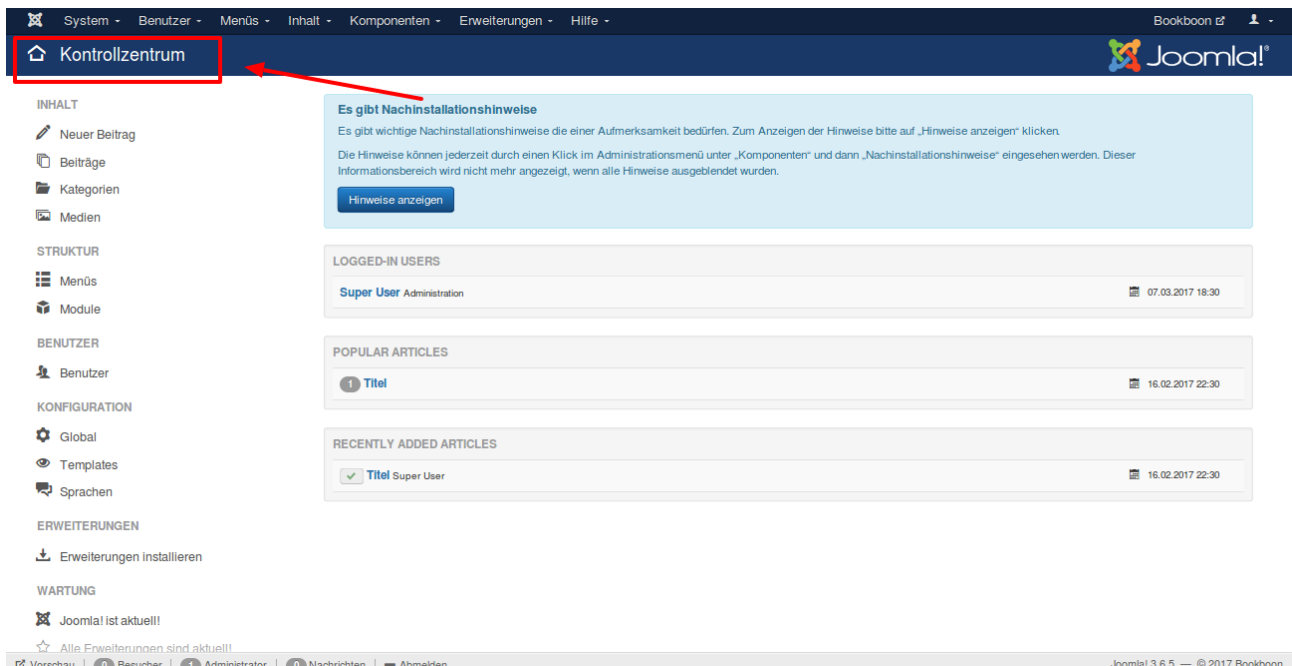


Abbildung 45: Die Ansicht des Kontrollzentrums im Joomla! Administrationsbereich.
969.png

Wenn Ihr Test erfolgreich ist, sehen Sie die gleiche Meldung wie ich – Sie können diese im nachfolgenden Kasten ablesen.

```
/var/www/html/joomla$ vendor/bin/codecept run functional
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.15 by Sebastian Bergmann and contributors.
Functional Tests (1) -----
✓ agpaypalCept: Ich will sicherstellen, dass die anmeldung im administrationsbereich funktioniert.
(0.34s)
-----
Time: 501 ms, Memory: 12.00MB
OK (1 test, 1 assertion)
```

So, die Anmeldung im Backend klappt und Sie wissen alles grundlegende, um mit der Erstellung des Beitrags weiterzumachen. Dies wäre ja nun der nächste Schritt zum Vervollständigen des hier begonnenen Tests. Danach müssten Sie nur noch den Menüpunkt anlegen. Zum Schluss können Sie dann den, ganz zu Beginn in diesem Kapitel erstellten Test, also die Anzeige im Frontend, durchlaufen lassen. Die notwendigen Voraussetzungen, also das Erstellen des Beitrags zur Anzeige des PayPal

Jetzt-kaufen-Buttons im Frontend, müssen Sie nun nicht mehr selbst erledigen. Dies übernimmt nun die Testroutine für Sie automatisch.

Die passenden Methoden finden Sie im Trait `FunctionalTesterActions`. Denken Sie auch daran, am Ende wieder den Ausgangszustand herzustellen – also den Beitrag mit dem PayPal Jetzt-kaufen-Button zu löschen. Wichtig ist dies insbesondere dann, wenn im Anschluss weitere Tests ausgeführt werden.

Hier im Buch geht es nun mit dem Optimieren der Tests weiter. Wie schreiben Sie Tests so, dass Sie so viel Testprogrammcode wie möglich wiederverwenden können?

Wiederverwendbare Tests

Wenn Sie das letzte Kapitel durchgearbeitet haben können Sie sich sicher vorstellen, dass das Erstellen von Tests teilweise eine monotone Tätigkeit sein kann.

Insbesondere dann, wenn es um Formulare geht, die viele Felder enthalten und sich vielleicht sogar über mehrere Unterseiten erstrecken. Demzufolge sollten wir Testcode so schreiben, dass wir diesen so oft wie möglich an anderen Stellen wieder verwenden können. Wie bewerkstelligen wir das am besten?

Das Anmeldeformular für den Administrationsbereich ist ein gutes Beispiel. Höchstwahrscheinlich werden Sie dieses vor fast jedem Test ausfüllen müssen. Das schreit geradezu nach Wiederverwendbarkeit. In Codeception sind Akteure, Step-Objekte und Page-Objekte für die Wiederverwendbarkeit von Programmcode zuständig.

RANDBEMERKUNG:

Akteure, Step-Objekte und Page-Objekte werden auch in Akzeptanztest eingesetzt. Um Akzeptanztests geht es im nächsten Kapitel.

Akteure

Akteure bieten eine Möglichkeit Testcode wiederzuverwenden. Den Akteur `FunctionalTester` hatten Sie schon kennengelernt. Sie finden die Klasse `FunctionalTester` im Verzeichnis `joomla/tests/_support`. Öffnen Sie hier die Datei `FunctionalTester.php` und fügen den im nachfolgenden Text fett markierten Teil ein.

```
<?php
class FunctionalTester extends \Codeception\Actor{
    use _generatedFunctionalTesterActions;
```

```

public function adminLogin(){
    $I = $this;
    $I->fillField(['id' => 'mod-login-username'], 'admin');
    $I->fillField(['id' => 'mod-login-password'], 'admin');
    $I->click('Anmelden');
}
}

```

Kommt Ihnen der eingefügte Text bekannt vor? Genau diesen hatten wir im vorhergehenden Kapitel im Programmcode der eigentlichen Testdatei verwendet. In der eigentlichen Testdatei reicht nun ein einfacher Methodenaufruf aus. Im nachfolgende Testcodeausschnitt sehen Sie die relevante Stelle fett hervorgehoben.

```

<?php
$I = new FunctionalTester($scenario);
$I->wantTo('Ich will sicherstellen, dass die Anmeldung im Administrationsbereich funktioniert.');
```

\$I->amOnPage('http://localhost/joomla/administrator');

\$I->adminLogin();

```

$I->see('Kontrollzentrum', ['class' => 'page-title']);

```

Warum haben wir diese Änderung gemacht? Bisher hat uns dies noch keinen Vorteil gebracht. Aber ab nun können Sie diese Methode in jedem Test einsetzen. Sicherlich werden Sie relativ oft einen Test mit einer Anmeldung im Administrationsbereich beginnen. Durch das Verschieben der notwendigen Schritte für die Anmeldung in die Methode `adminLogin()` der Klasse `FunctionalTester` müssen Sie in der Testkasse nun anstelle von vier Zeilen nur noch eine Zeile eingeben. Sie sparen drei Zeilen. Bei 10 Tests summiert sich dies auf 30 Zeilen. Dies bedeutet 30 Zeilen weniger, in denen Sie Tippfehler einbauen könnten. Außerdem ist Ihre Testklasse mit weniger Programmcode viel übersichtlicher!

Step-Objekte

Step-Objekte kommen ins Spiel, wenn Sie verschiedene inhaltlich zusammenhängende Bereiche testen. Zum Beispiel ist es sinnvoll, wiederverwendbare Testmethoden für das Frontend von den Methoden für das Backend zu separieren. Wenn Sie beispielsweise einen Onlineshop betreiben könnte es sinnvoll sein, Testmethoden, die die Bezahlung, die Produktdarstellung oder/oder die Suchfunktion

zum Gegenstand haben, separat abzulegen. So bleibt alles übersichtlicher und die passenden Methoden sind leicht auffindbar. Codeception bietet für diesen Zweck spezielle Klassen. Die Objekte dieser Klassen sind die Step-Objekte.

Das Codefragmente (Boilerplate) für ein Step-Objekt können Sie automatisch generieren. Die Testmethode für das Anmeldeformular zum Backend, dass wir eben beim Akteur implementiert hatten, würde besser in ein Step-Objekt passen, welches die Methoden für den Administrationsbereich gruppiert. Bauen wir also ein Step-Objekt und integrieren hier die Methode login(). Am einfachsten realisieren Sie dies über den Befehl `vendor/bin/codecept generate:stepobject functional Backend`.

```
/var/www/html/joomla$ vendor/bin/codecept generate:stepobject functional Backend
Add action to StepObject class (ENTER to exit): login
Add action to StepObject class (ENTER to exit):
StepObject was created in /var/www/html/joomla/tests/_support/Step/Functional/Backend.php
```

Wenn Sie möchten, können Sie natürlich auch Unterordner angeben: `vendor/bin/codecept generate:stepobject functional Unterordner/Backend`:

```
/var/www/html/joomla$ vendor/bin/codecept generate:stepobject functional Unterordner/Backend
Add action to StepObject class (ENTER to exit): login
Add action to StepObject class (ENTER to exit):
StepObject was created in
/var/www/html/joomla/tests/_support/Step/Functional/Unterordner/Backend.php
```

Zum automatisch erstellten Code des nun neu generierten Step-Objekts gibt es nicht viel zu sagen. Ich habe Ihnen dieses Objekt trotzdem der Vollständigkeit halber hier eingefügt.

```
<?php
namespace Step\Functional;
class Backend extends \FunctionalTester{
    public function login() {
        $I = $this;
```

```
| }  
| }
```

Vervollständigen Sie nun die Methode `login()` in der neuen Klasse `Backend`. Hierbei können Sie sich an der Methode `adminLogin()` in der Klasse `FunctionalTester` orientieren. Die Methode `adminLogin()` der Klasse `FunctionalTester` können Sie dann löschen. Diese brauchen wir nicht mehr. Die gleiche Funktionalität bietet ab jetzt die Methode `login()` der Klasse `Backend`.

```
<?php  
namespace Step\Functional;  
class Backend extends \FunctionalTester{  
    public function login(){  
        $I = $this;  
        $I->fillField(['id' => 'mod-login-username'], 'admin');  
        $I->fillField(['id' => 'mod-login-password'], 'admin');  
        $I->click('Anmelden');  
    }  
}
```

Wenn Sie das Step-Objekt im Testdurchlauf verwenden möchten, müssen Sie als Nächstes den Tester anpassen.

```
<?php  
use Step\Functional\Backend as AdminTester;  
$I = new AdminTester($scenario);  
$I->wantTo('Ich will sicherstellen, dass die Anmeldung im Administrationsbereich funktioniert.');
```

Page-Objekte

Es gibt neben Aktionen auch Elemente im HTML-Dokument, die mehrfach genutzt werden. Für diese Fälle sieht Codeception Page-Objekte vor.

Ein Page-Objekt repräsentiert

- eine Webseite als Klasse,
- die DOM-Elemente auf dieser Seite als Eigenschaften der Klasse und
- grundlegende Interaktionen als Methoden in der Klasse.

Page-Objekte sind sehr wichtig, wenn Sie Ihre Tests flexibel entwickeln möchten. Codieren Sie in diesem Fall keine komplexen [CSS](#)-Selektoren oder [XPath](#)-Ausdrücke in den Programmcode ihrer Testdateien. Verschieben Sie diese Ausdrücke in Page-Objekte. So müssen Sie bei eventuellen späteren Änderungen im Aufbau des HTML-Dokumentes nur an einer Stelle einen Eintrag anpassen.

Soviel zur Theorie. Wechseln wir nun zur Praxis. Der Befehl `vendor/bin/codecept generate:pageobject functional Backend` erstellt das Page-Objekt Backend in der Datei Backend.php im Verzeichnis `/var/www/html/joomla/tests/_support/Page/Functional/`.

```
/var/www/html/joomla$ vendor/bin/codecept generate:pageobject functional Backend
/var/www/html/joomla/tests/_support/Page/Functional/Backend.php
PageObject was created in /var/www/html/joomla/tests/_support/Page/Functional/Backend.php
```

Das automatisch generierte Page-Objekt enthält Variablen, die Sie füllen sollten. Eine dieser Variablen ist die URL. Sehen Sie sich das Codefragmente (Boilerplate) an, bevor Sie die Klasse an Ihre Tests anpassen:

```
<?php
namespace Page\Functional;
class Backend{
    // include url of current page
    public static $URL = "";
    /**
     * Declare UI map for this page here. CSS or XPath allowed.
     * public static $usernameField = '#username';
     * public static $formSubmitButton = "#mainForm input[type=submit]";
     */
    /**
     * Basic route example for your current URL
     * You can append any additional parameter to URL
     * and use it in tests like: Page\Edit::route('/123-post');
```

```

    */
    public static function route($param){
        return static::$URL.$param;
    }
    /**
     * @var \FunctionalTester;
     */
    protected $functionalTester;
    public function __construct(\FunctionalTester $I){
        $this->functionalTester = $I;
    }
}

```

Um das Page-Objekte zur Verwendung in unserer aktuellen Testdatei anzupassen, sollten wir mindestens die folgenden vier Zeilen einfügen. Diese Zeilen bewirken, dass die URL des Anmeldeformulars und die IDs jeweils in einer eigenen Variablen gespeichert werden.

```

<?php
namespace Page\Functional;
class Backend{
    public static $URL = 'http://localhost/joomla/administrator';
    public static $username = ['id' => 'mod-login-username'];
    public static $password = ['id' => 'mod-login-password'];
    public static $anmelden = 'Anmelden';
    ...
}

```

Anhand der fett abgedruckten Teile im nachfolgenden Programmcode sehen Sie, wie Sie das Page-Objekt im Step-Objekt verwenden können.

```

<?php
namespace Step\Functional;
use Page\Functional\Backend as BackendPage;
class Backend extends \FunctionalTester{
    public function login(){
        $I = $this;
    }
}

```

```

    $I->fillField(BackendPage::$Susername, 'admin');
    $I->fillField(BackendPage::$Spassword, 'admin');
    $I->click(BackendPage::$anmelden);
  }
}

```

Im Testcode selbst ändern Sie noch die als Text eingegebene URL. Verwenden Sie anstelle des Textes die im Page-Objekt erstellte Variable.

```

<?php
use Step\Functional\Backend as AdminTester;
use Page\Functional\Backend as BackendPage;
$I = new AdminTester($scenario);
$I->wantTo('Ich will sicherstellen, dass die Anmeldung im Administrationsbereich funktioniert.');
```

```

$I->amOnPage(BackendPage::$URL);
$I->login();
$I->see('Kontrollzentrum', ['class' => 'page-title']);

```

Nun müssen Sie keine Sorge mehr vor Änderungen auf der Produktivwebsite haben. Wenn Sie konsequent alle Elemente über Variablen in einem Page-Objekt ansprechen, reicht eine Korrektur im Page-Objekt aus, um Ihre Tests wieder an die Produktivwebsite anzupassen.

REST-Schnittstelle

REST ist eine einfache Möglichkeit, Interaktionen zwischen unabhängigen Systemen zu organisieren.

Konfiguration

Wenn Sie die REST-Schnittstelle in Funktionstests verwenden möchten, müssen Sie die Konfiguration anpassen. Fügen Sie das Modul REST in der Konfigurationsdatei `functional.suite.yml` hinzu. Die Datei `functional.suite.yml` finden Sie im Verzeichnis `/var/www/html/joomla/tests/` und was Sie genau in diese Datei einfügen müssen habe ich nachfolgend fett hervorgehoben.

```

class_name: FunctionalTester
modules:

```


enabled:

- PhpBrowser:
 - url: 'http://localhost/joomla'
- \Helper\Functional
- **REST:**
 - depends: PhpBrowser**
 - url: 'http://localhost/joomla/v1'**

Damit Aktionen des REST-Modules in den Trait `FunctionalTesterActions` integriert werden und von Ihrem Tester verwendet werden können, müssen Sie den Befehl `vendor/bin/codecept build` ausführen.

```
/var/www/html/joomla$ vendor/bin/codecept build
Building Actor classes for suites: unit, acceptance, functional
-> UnitTesterActions.php generated successfully. 0 methods added
\UnitTester includes modules: Asserts, \Helper\Unit
-> AcceptanceTesterActions.php generated successfully. 0 methods added
\AcceptanceTester includes modules: PhpBrowser, \Helper\Acceptance
-> FunctionalTesterActions.php generated successfully. 0 methods added
\FunctionalTester includes modules: PhpBrowser, \Helper\Functional, REST
```

In der Ausgabe der Konsole können Sie sich davon vergewissern, dass das REST-Modul richtig geladen ist. Wenn Sie dann zusätzlich noch einen Blick in die Datei `/joomla/tests/_support/_generated/FunctionalTesterActions.php` werfen, werden Sie hier einige neue Funktion finden. Zum Beispiel die Methode `sentGET()`, die wir im nächsten Beispiel verwenden.

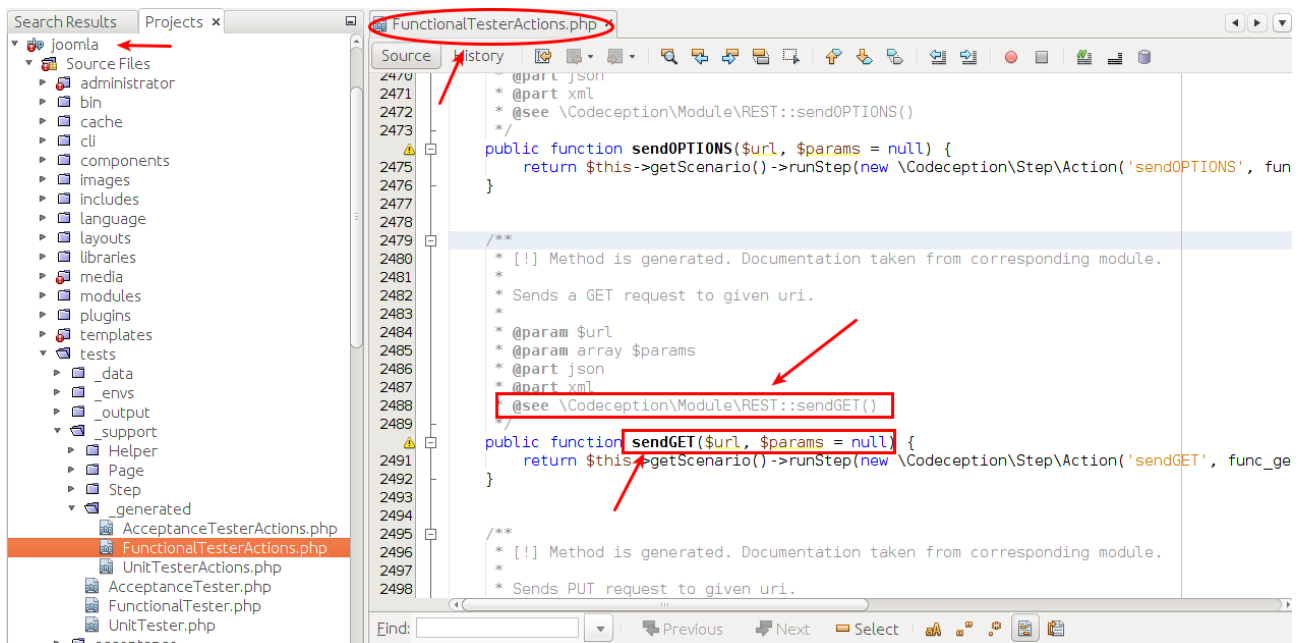


Abbildung 46: Die Methode `sendGET()` in der Datei `/joomla/tests/_support/_generated/FunctionalTesterActions.php` gehört zum REST-Module. 952.png

Ein Beispiel für GET

Ich hatte schon im Kapitel *Codeception – ein Überblick | Codeception – ein erster Rundgang | Testtypen in Codeception* angesprochen, das Joomla! das REST-Paradigma nicht vollständig erfüllt. Ich möchte Ihnen hier aber trotzdem kurz an einem Beispiel zeigen, wie Sie die wichtigsten HTTP-Methoden in einem Funktionstest nutzen können. Dazu erstellen wir als Erstes die Testdatei. Wie üblich lassen wir uns diese von Codeception mithilfe des Befehls `vendor/bin/codecept generate:cept functional /suites/plugins/content/agpaypal/rest` generieren.

```
/var/www/html/joomla$ vendor/bin/codecept generate:cept functional
/suites/plugins/content/agpaypal/rest
Test was created in
/var/www/html/joomla/tests/functional//suites/plugins/content/agpaypal/restCept.php
```

Zu Beginn dieses Kapitels haben wir das Formular für die Anmeldung im Administrationsbereich getestet. Zur Abwechslung testen wir nun das Anmeldeformular im Frontend. Hier nutzen wir nicht das Login-Formular des Moduls, das bei einer Standardinstallation in der rechten Seitenleiste integriert ist. Dieses Modul kann auch deaktiviert werden. Über die Adresse `http://localhost/joomla/index.php?`

option=com_users wird Ihnen aber in Joomla! immer das Anmeldeformular der Komponenten com_users angezeigt.

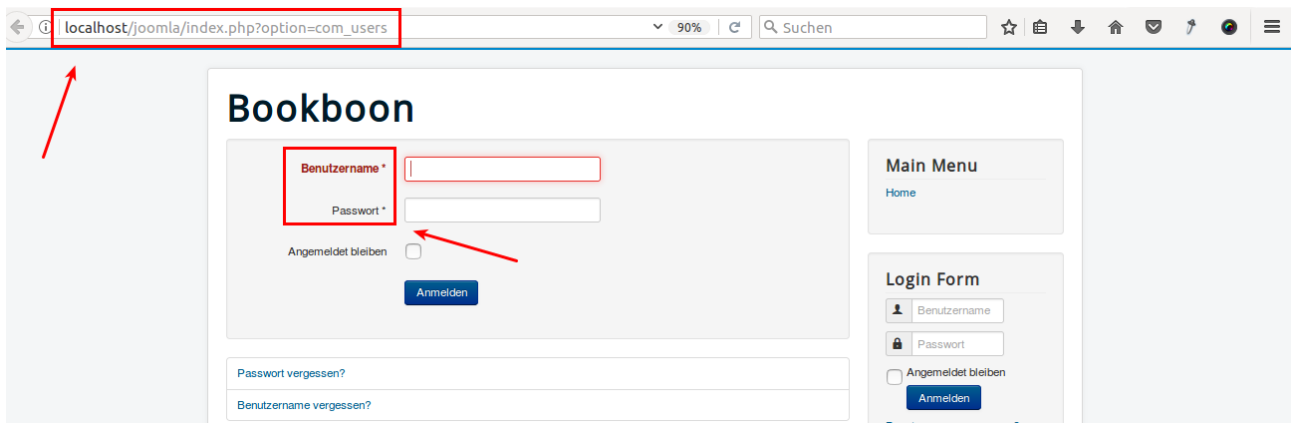


Abbildung 47: Das Anmeldeformular im Joomla! Frontend. 966.png

Beginnen wir unseren Test, indem wir mit `$I->sendGET('http://localhost/joomla/index.php?option=com_users');` die Verbindung zum Anmeldeformular herstellen und dann überprüfen, ob die Felder des Formulars auch angezeigt werden.

```
<?php
$I = new FunctionalTester($scenario);
$I->wantTo('perform actions and see result');
$I->sendGET('http://localhost/joomla/index.php?option=com_users');
$I->see('Benutzername');
$I->see('Passwort');
```

`$I->see('Benutzername');` und `$I->see('Passwort');` sind nur erfolgreich, wenn wir den Text Benutzername und Passwort auch auf der Website sehen. Probieren wir dies aus:

```
//var/www/html/joomla$ vendor/bin/codecept run functional
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.15 by Sebastian Bergmann and contributors.
Functional Tests (2) -----
✓ agpaypalCept: Ich will sicherstellen, dass die anmeldung im administrationsbereich funktioniert.
(0.32s)
✓ restCept: Perform actions and see result (0.04s)
-----
```

Time: 519 ms, Memory: 12.00MB

OK (2 tests, 3 assertions)

Ja, der Test ist erfolgreich. Das Formular wird richtig erkannt. Nun müssen wir dieses mit korrekten Anmeldedaten füllen und absenden um sicherzustellen, dass auch die Verarbeitung innerhalb des Formulars korrekt funktioniert. Dies würden wir mithilfe der POST-Methode tun. Aber: Das Anmeldeformular in Joomla! kann zur Zeit noch nicht ohne weiteres mit einem Post-Request gefüllt werden. Die Eingaben im Anmeldeformular werden nicht weitergeleitet, weil ein Joomla! internes Sicherheitstoken von außen nicht korrekt gefüllt werden kann. Joomla! arbeitet an einer REST-Schnittstelle. Im Joomla! Google Summer of Code (GSoC) 2017 wird sich ein [Projekt mit der REST-Schnittstelle](#) befassen. Der Vollständigkeit halber zeige ich Ihnen im nächsten Kapitel das Absetzen eines Post-Requests an einem ganz einfachen konstruierten Beispiel.

Ein Beispiel für POST

Zunächst erstellen wir ein einfaches Formular in einem HTML-Dokument. Erstellen Sie das Unterverzeichnis `Beispiel` im Stammverzeichnis Ihres Webserver. Legen Sie dann die Datei `index.html` mit dem folgenden Inhalt in diesem Verzeichnis an.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Titel</title>
  </head>
  <body>
    <form action="action.php" method="post">
      <p>Name: <input type="text" name="name" /></p>
      <p><input type="submit" /></p>
    </form>
  </body>
</html>
```

Sie verfügen nun über eine Datei mit dem Namen `index.html` im Verzeichnis `/var/www/html/beispiel`.

Erstellen Sie im selben Verzeichnis die PHP Datei `action.php`. Diese Datei, beziehungsweise dieses Skript, soll ausgeführt werden, wenn das Formular abgesandt wird. Schreiben Sie folgende Zeile in die Datei `action.php`.

```
| Hello <?php echo htmlspecialchars($_POST['name']); ?>.
```

Wir haben nun ein simples Formular erstellt. Wenn Sie die Adresse `http://localhost/beispiel` in Ihren Internetbrowser eingeben, können Sie dieses Formular sehen. Testen Sie nun, ob das Formular auch funktioniert. Dazu geben Sie als Nächstes einen Namen im Feld *Name* ein und klicken auf die Schaltfläche die mit *Daten absenden* beschriftet ist.

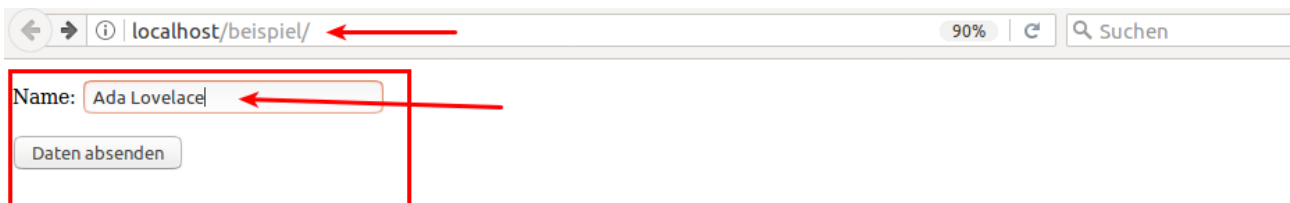


Abbildung 48: Ein simples Formular zum Test eines POST-Requests. 951a.png

Wenn alles richtig läuft, werden Sie zur URL `http://localhost/beispiel/action.php` weitergeleitet. Hier sehen Sie nun den Begrüßungstext im Inhaltsbereich. Sie werden mit dem Namen begrüßt, denn Sie vorher im Formular eingegeben haben.

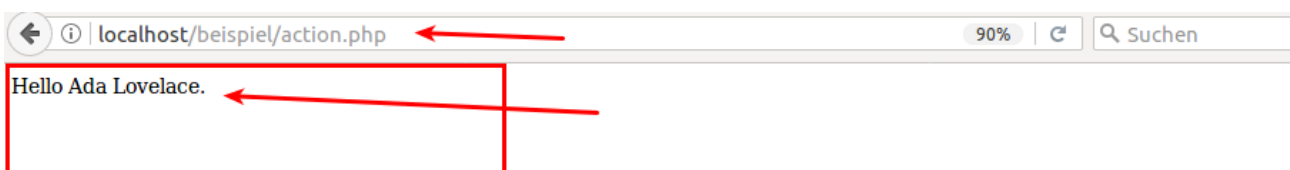


Abbildung 49: Die Ausgabe des Testformulars. 951b.png

Dieser manuelle Test des schlichten Formulars hat bei Ihnen sicherlich auch funktioniert. Dann können Sie dieses Formular nun als Grundlage für das Erstellen von Funktionstests mithilfe der `sendPost()`-Methode verwenden. Mit dem Aufruf der Methode `sendPOST()` wird, wie der Name schon sagt, ein Post-Request abgesetzt.

Erstellen wir nun die Testdatei. Der Befehl `vendor/bin/codecept generate:cept functional /suites/plugins/content/agpaypal/rest2` übernimmt dies für Sie.

```
/var/www/html/joomla$ vendor/bin/codecept generate:cept functional
/suites/plugins/content/agpaypal/rest2
Test was created in
/var/www/html/joomla/tests/functional//suites/plugins/content/agpaypal/rest2Cept.php
```

Füllen Sie die Datei `/joomla/tests/functional//suites/plugins/content/agpaypal/rest2Cept.php` nun mit nachfolgendem Testcode. Die Zeile `$I->amOnPage('http://localhost/php/index.php');` stellt eine Verbindung zum Formular her. Danach wird der Name Peter per POST-Methode an die URL `http://localhost/beispiel/action.php` gesandt. Nun müsste der Text „Hello Peter“ zu sehen sein! Dies prüft die Methode `$I->see('Hello Peter');`.

```
$I->amOnPage('http://localhost/php/index.php');
<?php
$I = new FunctionalTester($scenario);
$I->wantTo('Ich will sicherstellen, dass das Formular funktioniert.');
```

```
$I->amOnPage('http://localhost/beispiel/index.html');
$I->sendPOST(
    'http://localhost/beispiel/action.php',
    [ 'name' => 'Peter', ]);
$I->see('Hello Peter');
```

Starten Sie den Test dieser Methode mittels `vendor/bin/codecept run functional`
`./tests/functional/suites/plugins/content/agpaypal/rest2Cept.php`. Sie werden sehen, er wird erfolgreich sein.

```
/var/www/html/joomla$ vendor/bin/codecept run functional
./tests/functional/suites/plugins/content/agpaypal/rest2Cept.php
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.17 by Sebastian Bergmann and contributors.
Functional Tests (1) -----
✓ rest2Cept: Ich will sicherstellen, dass das formular funktioniert. (0.02s)
-----
Time: 193 ms, Memory: 10.00MB
```

Das Prinzip von Funktionstests, die eine REST-Schnittstelle verwenden, habe ich Ihnen nun an zwei Beispielen gezeigt. Wenn Sie das REST-Modul verwenden möchten, finden Sie detailliertere Informationen in der Codeception Dokumentation zum [REST-Module](#) und bei den [Anleitungen](#).

Was Sie bei Funktionstests beachten sollten

Funktionstests sind schnell aber dafür auch weniger stabil

Ein Vorteil von Funktionstests ist, dass diese relativ schnell ablaufen – zumindest schneller als Akzeptanztests. Warum Akzeptanztests langsamer sind, erkläre ich Ihnen im nächsten Kapitel. Leider laufen Funktionstests aber auch weniger stabil. Denn, die zu testende Anwendung wird nicht auf einem separaten Webserver, sondern mit dem Modul PHPBrowser im selben Speicherbereich wie die Tests, ausgeführt. Somit terminiert auch das Testskript, wenn in der zu testenden Anwendung die Methode [exit\(\)](#) oder die Methode [die\(\)](#) aufgerufen wird.

Auch die Auswertung von globalen Variablen ist problematisch. Eine globale Variable kann von mehreren Funktionen gleichzeitig verwendet werden. Manche Funktionen ändern die Variable andere fragen lediglich ihren Wert ab. So bewirken globale Variablen eine starke Kopplung der betroffenen Funktionen. Wenn eine globale Variable in einer Funktion geändert wird, hat dies Auswirkungen auf andere Funktionen. Dies erschwert auch das Testen und das gilt ganz besonders für Funktionstests.

Header

Eines der häufigsten Probleme im Zusammenhang mit Funktionstests tritt bei der Verwendung der Methode [header\(\)](#) auf. Mithilfe dieser Funktion können Sie [HTTP-Header-Felder](#) im Rohformat senden. Dies muss allerdings geschehen, bevor auch nur ein Zeichen gesendet wurde. Ist bereits ein Zeichen gesendet, ist es nicht mehr möglich den Header zu beeinflussen – denn dieser ist ja bereits abgesandt worden. Da Funktionstests die Anwendung oft mehrmals hintereinander ausführen, kann es zu Überschneidungen kommen. Eventuell werden dann Fehler gemeldet, die eigentlich gar keine Fehler sind.

Speichermanagement

Ich hatte es eben schon geschrieben: Alle Funktionstests laufen im gleichen Speicherbereich wie die zu testende Anwendung. Und, anders als bei der Ausführung der eigentlichen Anwendung auf einem Webserver, wird dieser Speicherbereich nach dem Ende eines Tests nicht automatisch wieder freigegeben. Während jedes Tests

sollte die Methode `_after()` dafür sorgen, dass alle nicht mehr benötigten Speicherbereiche wieder freigegeben werden. Falls hier etwas nicht korrekt läuft oder vergessen wurde, kann dies zu Problemen führen. Zeigen Ihre Tests Fehler an und Sie sind sich gleichzeitig sicher, dass die Testbedingung erfüllt ist, könnten Speicherprobleme die Ursache sein. Separieren Sie dann einen Test und führen ihn einmal getrennt von anderen Tests aus. Läuft der Test nun durch, wissen Sie, dass nicht der Test das Problem ist. Die Fehlerursache sollten Sie im Speichermanagement suchen.

Kurzgefasst

Funktionstests waren das Hauptthema dieses Kapitels. Beim Erstellen dieser Tests haben Sie gesehen, dass Codeception Sie dabei unterstützt, wiederverwendbaren und gut lesbaren Testcode zu schreiben. Dieses Wissen können Sie auch bei den Akzeptanztests, die Thema des nächsten Kapitels sind, anwenden. Außerdem wissen Sie nun auch, was Sie beim Erstellen von Funktionstests beachten müssen und wie Sie eine REST-Schnittstelle nutzen können.

Akzeptanztests

Mit den Akzeptanztests widme ich mich dem letzten Testtyp. Machen wir uns noch einmal klar, warum wir welchen Test erstellen. Mit Unittests wollen wir sicherstellen, dass jede kleine Einheit korrekt arbeitet. Funktionstests sollen sicherstellen, dass diese kleinsten Einheiten richtig zusammen arbeiten.

Mit Akzeptanztests überprüfen wir, ob die Spezifikationen, die ganz zu Beginn des Projektes aufgestellt wurden, erfüllt sind: Kann man mithilfe der Software tatsächlich das, was man erreichen will? Vielleicht kommt es Ihnen abwegig vor. Aber es könnte sein, dass unser PayPal Jetzt-kaufen-Button auf Websites angeboten wird, deren Besucher von einem Intranet aus zugreifen. Wenn die PayPal Website in diesem Intranet geblockt ist, kann der PayPal Jetzt-kaufen-Button nicht verwendet werden. Dies ist ganz klar ein Mangel. Alleine mit Unittests und Funktionstests würde dieser Mangel aber nicht erkannt. Erst ein Akzeptanztest, der so realitätsnah wie möglich ausgeführt werden soll, würde auf diesen Missstand aufmerksam machen.

Selenium WebDriver – eine Einführung

Um zu überprüfen, ob Ihre Software tatsächlich das tut, was sie tun soll, müssen Akzeptanztests so realitätsnah wie möglich durchgeführt werden. Anders als die Funktionstests im vorausgehenden Kapitel führen wir Akzeptanztests deshalb auf einem echten Browser aus. Und, da wir diese Tests nicht manuell, sondern

automatisch durchführen möchten, nutzen wir das Framework [Selenium](#). Wir befinden uns da in guter Gesellschaft – Selenium ist eines der meistgenutzten Testwerkzeuge in der Webentwicklung. Sehen wir uns als Erstes den Teil von Selenium, den wir für unsere Tests benötigen, genauer an. Im nächsten Kapitel installieren wir zunächst [Selenium Standalone Server](#) und [ChromeDriver](#) auf unserem Rechner und starten dann den Selenium Standalone Server.

RANDBEMERKUNG:

Ein WebDriver bietet eine Schnittstelle zu einem Browser. Über diese Schnittstelle kann der Browser automatisch gesteuert und kontrolliert werden. Interessant ist in dem Zusammenhang, dass es einen Entwurf zu einer [W3C-Spezifikation WebDriver](#) gibt. Grundlage für diese Spezifikation ist [Selenium WebDriver](#). Codeception nutzt eine in PHP programmierte [Selenium WebDriver Implementierung](#), die von Facebook entwickelt wurde.

Warum habe ich [Google Chrome](#) als Beispiel gewählt? Dafür gibt es keinen Grund. Ziel ist es ohnehin die Anwendung auf allen möglichen Browsern zu testen. Wenn Ihre Akzeptanztests mit Google Chrome laufen, können Sie weitere Browser hinzufügen. In der [Codeception Dokumentation](#) finden Sie Informationen dazu, wie Sie Ihre Tests so erweitern können, dass diese auf unterschiedliche Browser gleichzeitig durchgeführt werden. Beginnen wir aber hier nun mit Google Chrome. Dabei gehe ich davon aus, dass Sie den Internetbrowser [Google Chrome](#) auf Ihrem Rechner installiert haben. Falls nicht, müssten Sie dies nun nachholen. Die aktuelle Version von Google Chrome finden Sie unter der Adresse <https://www.google.de/chrome/>.

Selenium und ChromeDriver installieren

Sind Sie gespannt darauf, wie die Tests im wirklichen Browser automatisch ablaufen? Dann besorgen Sie sich die neueste Version des WebDrivers für den Browser Chrome. Diese finden Sie unter der Adresse

<https://sites.google.com/a/chromium.org/chromedriver/downloads>. Ich habe die

Version 2.28 mithilfe des Befehls `wget -N`

`http://chromedriver.storage.googleapis.com/2.28/chromedriver_linux64.zip -P ~/` auf meinen Ubuntu Rechner kopiert.

```
$ wget -N http://chromedriver.storage.googleapis.com/2.28/chromedriver_linux64.zip -P ~/
--2017-03-15 19:25:11-- http://chromedriver.storage.googleapis.com/2.28/chromedriver_linux64.zip
```

```
Auflösen des Hostnamen »chromedriver.storage.googleapis.com
(chromedriver.storage.googleapis.com)«... 172.217.22.112, 2a00:1450:4001:81d::2010
Verbindungsaufbau zu chromedriver.storage.googleapis.com (chromedriver.storage.googleapis.com)
172.217.22.112|:80... verbunden.
HTTP-Anforderung gesendet, warte auf Antwort... 200 OK
Länge: 3458530 (3,3M) [application/zip]
In »»/home/meinName/chromedriver_linux64.zip«« speichern.
chromedriver_linux64 100%[=====>] 3,30M 3,75MB/s in 0,9s
2017-03-15 19:25:12 (3,75 MB/s) - »/home/meinName/chromedriver_linux64.zip« gespeichert
[3458530/3458530]
```

Entpacken Sie die gerade heruntergeladene Datei zum Beispiel mit dem Befehl `unzip ~/chromedriver_linux64.zip -d ~/` und löschen Sie dann der Ordnung halber das Archiv mit dem Befehl `rm ~/chromedriver_linux64.zip`.

```
$ unzip ~/chromedriver_linux64.zip -d ~/
Archive: /home/meinName/chromedriver_linux64.zip
  inflating: /home/meinName/chromedriver
meinName@acer:~$ rm ~/chromedriver_linux64.zip
```

Ubuntu erwartete den ChromeDriver im Verzeichnis `/usr/local/share/`. Deshalb verschieben wir diesen mit dem Befehl `sudo mv -f ~/chromedriver /usr/local/share/` nun dorthin und machen ihn mit dem Befehl `sudo chmod +x /usr/local/share/chromedriver` ausführbar. Zuletzt setzen wir noch einen symbolischen Link zum Verzeichnis `/usr/local/bin/chromedriver` – dies erwartet Ubuntu ebenfalls.

```
meinName@acer:~$ sudo mv -f ~/chromedriver /usr/local/share/
meinName@acer:~$ sudo chmod +x /usr/local/share/chromedriver
meinName@acer:~$ sudo ln -s /usr/local/share/chromedriver /usr/local/bin/chromedriver
```

Nachdem der WebDriver für Chrome nun fertig installiert ist, nehmen wir die Installation von Selenium Standalone Server in Angriff. Auch hier müssen wir als Erstes die Installationsdatei auf unseren Rechner kopieren. Die neueste Version des Selenium Standalone Servers finden Sie unter der Internetadresse <http://www.seleniumhq.org/download/>. Ich habe die gepackte Version 3.0.1 des

Servers mit dem Befehl `wget -N http://selenium-release.storage.googleapis.com/3.0/selenium-server-standalone-3.0.1.jar -P ~/` auf meinen Rechner kopiert.

```
meinName@acer:~$ wget -N http://selenium-release.storage.googleapis.com/3.0/selenium-server-standalone-3.0.1.jar -P ~/
--2017-03-15 19:26:23-- http://selenium-release.storage.googleapis.com/3.0/selenium-server-standalone-3.0.1.jar
Auflösen des Hostnamen »selenium-release.storage.googleapis.com (selenium-release.storage.googleapis.com)«... 172.217.22.112, 2a00:1450:4001:81d::2010
Verbindungsaufbau zu selenium-release.storage.googleapis.com (selenium-release.storage.googleapis.com)|172.217.22.112|:80... verbunden.
HTTP-Anforderung gesendet, warte auf Antwort... 200 OK
Länge: 22139212 (21M) [application/java-archive]
In »»/home/meinName/selenium-server-standalone-3.0.1.jar«« speichern.
selenium-server-standalone-3.0.1.jar 100%[=====>] 21,11M 4,85MB/s in 4,6s
2017-03-15 19:26:28 (4,62 MB/s) - »/home/meinName/selenium-server-standalone-3.0.1.jar« gespeichert [22139212/22139212]
```

Wenn Sie sich die gepackte Version des Servers ebenfalls kopiert haben, ist auf Ihrem Computer nun ein ausführbares [Java Archiv](#) mit dem Namen `selenium-server-standalone-x.x.x.jar`.

RANDBEMERKUNG:

Voraussetzung für das Ausführen von Dateien mit der Endung JAR, beziehungsweise Java-Programmen, ist immer eine installierte [Java-Laufzeitumgebung](#) – also eine JRE.

Verschieben Sie dieses Java Archiv analog zur Vorgehensweise beim WebDriver für den Browser Chrome und erstellen anschließend einen symbolischen Link. Die Befehle dazu finden Sie im nachfolgenden Kasten. Das war es auch schon.

```
meinName@acer:~$ sudo mv -f ~/selenium-server-standalone-3.0.1.jar /usr/local/share/
meinName@acer:~$ sudo chmod +x /usr/local/share/selenium-server-standalone-3.0.1.jar
meinName@acer:~$ sudo ln -s /usr/local/share/selenium-server-standalone-3.0.1.jar /usr/local/bin/selenium-server-standalone-3.0.1.jar
```

Sie können den Selenium Standalone Server ab jetzt mit dem Befehl `java -Dwebdriver.chrome.driver=/usr/local/bin/chromedriver -jar /usr/local/bin/selenium-server-standalone-3.0.1.jar -debug` starten. Probieren Sie es aus!

```
java -Dwebdriver.chrome.driver=/usr/local/bin/chromedriver -jar /usr/local/bin/selenium-server-standalone-3.0.1.jar -debug
...
21:27:25.825 INFO - Selenium Server is up and running
```

Wenn der Selenium Standalone Server Prozess läuft und Codeception so konfiguriert ist, dass es WebDriver verwendet, wird sich später beim Start eines Tests Ihr Browser automatisch öffnen und die zu testenden Schritte wie von Geisterhand ausführen. Das ist spannend, oder? Machen wir also schnell mit der Konfiguration von Codeception weiter.

RANDBEMERKUNG:

Wenn Sie die Pakete `openjdk-8-jre-headless`, `xvfb`, `libxi6` und `libgconf-2-4` auf Ihrem Ubuntu Rechner installiert haben, können Sie die Tests auch schneller kopflos oder headless – also ohne grafische Anzeige – ausführen. Die Aufrufvariante `xvfb-run java -Dwebdriver.chrome.driver=/usr/local/bin/chromedriver -jar /usr/local/bin/selenium-server-standalone-3.0.1.jar -debug` führt die Tests mit dem Browser Google Chrome durch, Sie sehen hierbei allerdings keine Benutzeroberfläche.

```
$ xvfb-run java -Dwebdriver.chrome.driver=/usr/local/bin/chromedriver -jar /usr/local/bin/selenium-server-standalone-3.0.1.jar -debug
...
19:27:12.089 INFO - Selenium Server is up and running
```

Codeception konfigurieren

Damit Ihre Akzeptanztests mithilfe des WebDriver im Browser Chrome ausgeführt werden, müssen Sie die Konfigurationsdatei `/var/www/html/joomla/tests/acceptance.suite.yml` anpassen. Als Erstes müssen Sie das Modul WebDriver aktivieren und die passenden Parameter dazu angeben. Im nachfolgenden Codebeispiel finden Sie die Angaben, die Sie für unser Beispiel einfügen sollten. Tun Sie dies bitte nun. Möchten Sie sich selbst einen Überblick über alle möglichen Einstellungen verschaffen? Weitere Informationen finden Sie in der [Dokumentation von Codeception](#).

```
class_name: AcceptanceTester
modules:
  enabled:
    - WebDriver
    - \Helper\Acceptance
  config:
    WebDriver:
      url: 'http://localhost/joomla'
      browser: 'chrome'
      window_size: 1024x768
      restart: true
```

Für unser Beispiel müssen Sie für den WebDriver die Konfigurationsparameter `url` und `browser` setzen. Alle anderen Parameter sind optional.

RANDBEMERKUNG:

Manchmal kommt es auf die kleinen Dinge an. Da ich über diesen Punkt gestolpert bin, erwähne ich ihn hier: Ganz wichtig ist, dass Sie **WebD**river mit einem großen D schreiben.

Sie wissen es schon, nach Änderungen in einer der Konfigurationsdateien müssen Sie den Befehl `vendor/bin/codecept build` ausführen, damit Codeception diese Änderung auch in allen anderen Unterstützungsdateien übernimmt.

```
/var/www/html/joomla$ vendor/bin/codecept build
Building Actor classes for suites: unit, acceptance, functional
-> UnitTesterActions.php generated successfully. 0 methods added
\UnitTester includes modules: Asserts, \Helper\Unit
-> AcceptanceTesterActions.php generated successfully. 0 methods added
\AcceptanceTester includes modules: WebDriver, \Helper\Acceptance
-> FunctionalTesterActions.php generated successfully. 0 methods added
\FunctionalTester includes modules: PhpBrowser, \Helper\Functional, REST
```

Möchten Sie sich einen Überblick über die nun verfügbaren Testmethoden verschaffen? Dann sehen Sie sich die Datei

`/var/www/html/joomla/tests/_support/_generated/AcceptanceTesterActions.php` an. Bevor Sie diese Testmethoden anwenden können, müssen Sie aber noch eine andere Sprache lernen, nämlich die Beschreibungssprache [Gherkin](#). Das hört sich komplizierter an, als es ist. Diese Sprache ist ganz einfach, Sie werden sehen.

Warum müssen Sie Gherkin lernen? Wie auch bei den meisten anderen Behaviour-Driven-Development-Tools werden in Codeception Szenarien in Akzeptanztests mittels Gherkin beschrieben. Gherkin verwendet die natürliche Schriftsprache als Grundlage. Nur bestimmte Schlüsselwörter werden besonders behandelt. Sehen wir uns das im nächsten Abschnitt, an einem in Gherkin geschriebenen Szenario, in der Praxis an.

Gherkin

Zunächst müssen Sie wissen, dass die in Gherkin beschriebenen Szenarien für Akzeptanztests in sogenannten Feature-Dateien gespeichert werden. Diese Feature-Dateien sollten Sie bei einer testgetriebenen Entwicklung vor der eigentlichen Implementierung des Programmcodes anlegen. Nur so würde das in der Einleitung angesprochene Problem mit dem Intranet, das keinen Zugriff auf die Website von PayPal hat, frühzeitig erkannt. Und dass Fehler, die in der Entwurfsphase eines Softwareprojektes entstehen und erst spät erkannt werden die teuersten Fehler sind, hatten wir bereits ganz zu Beginn im Kapitel *Softwaretests – eine Einstellungssache?* | *Softwaretests in den Arbeitsablauf integrieren* herausgearbeitet.

RANDBEMERKUNG:

Wenn Sie mit der integrierten Entwicklungsumgebung Netbeans arbeiten, sollten Sie sich das Plugin [Cucumber](#) ansehen. Diese Erweiterung wurde seit 2010 nicht mehr aktualisiert. Das hat mich zunächst skeptisch gemacht. Das Plugin arbeitet bei mir aber mit der Netbeans Version 8.2 fehlerfrei. Was tut das Plugin genau? Es hebt die Syntax von Gherkin hervor.

Alle Vorbereitungen sind nun erledigt. Nun können wir endlich den ersten Test praktisch angehen. Erstellen Sie im Verzeichnis `/var/www/html/joomla/tests/acceptance/` die Feature-Datei `paypal.feature` mit folgendem Inhalt:

```
Feature: paypal
  In order to manage content articles with paypal links
  I need to create a content article with a paypal link
```

Background:

When I login into Joomla administrator

And I see the administrator dashboard

Scenario: Create an Article

Given There is a add content link

When I create new content with field title as "Beitrag mit PayPal-Button" and content as a "Dies ist mein erster Beitrag."

And I save an article

Then I should see the article "Beitrag mit PayPal-Button" is created

Jede Feature-Datei sollte am Anfang eine kurze Beschreibung des Testgegenstandes haben.

Danach kann optional ein Background Block eingefügt werden. Steps in diesem Background Block werden vor jedem Szenario ausgeführt. Sinnvoll ist die Verwendung eines Background Blocks zum Beispiel, wenn Sie Tests im Administrationsbereich implementieren und jedes Szenario eine erfolgreiche Anmeldung im Administrationsbereich voraussetzt. Wenn Sie dieses Login in einem Background Block einfügen, können Sie sich diese Zeile in jedem einzelnen Szenario sparen.

Ein Szenario kann aus beliebig vielen verschiedenen Zeilen oder Steps bestehen. Ein Step kann anstelle von GIVEN, WHEN oder THEN auch mit AND beginnen. Wenn Sie beispielsweise mehrere Steps für ein WHEN in Gherkin beschreiben möchten werden Sie feststellen, dass Sie das Szenario leichter lesen können, wenn Sie diese Steps mit AND verbinden, anstelle von mehreren THEN hintereinander.

Beim Erstellen der Feature-Dateien sollten Sie die Intention der verschiedenen Steps beachten.

- **GIVEN:**

Die Anweisungen in den Steps die mit GIVEN beginnen, sollen die Voraussetzungen für diesen Testfall schaffen. Wenn es beispielsweise um die Erstellung von neuen Beiträgen geht, sollte im ersten Step Given There is a add content link der Browser so geöffnet werden, dass der Benutzer sich auf der Unterseite der Anwendung befindet, auf der sich die Schaltfläche zum Erstellen eines Beitrags befindet.

- **WHEN:**

Die Steps, die mit WHEN beginnen führen die eigentliche Testaktion aus. In unserem Beispiel legen wir im Step When I create new content with field title as "Beitrag mit PayPal-Button" and content as a "Dies ist mein erster Beitrag." den Beitrag an und speichern ihn danach im Step And I save an article.

- **THEN:**

Das, was im WHEN-Teil ausgeführt wurde, soll ein bestimmtes Ergebnis zur Folge haben. Steps, die mit THEN beginnen, also Then I should see the article "Beitrag mit PayPal-Button" is created, prüfen nun, ob dieses Ergebnis wirklich erreicht wurde.

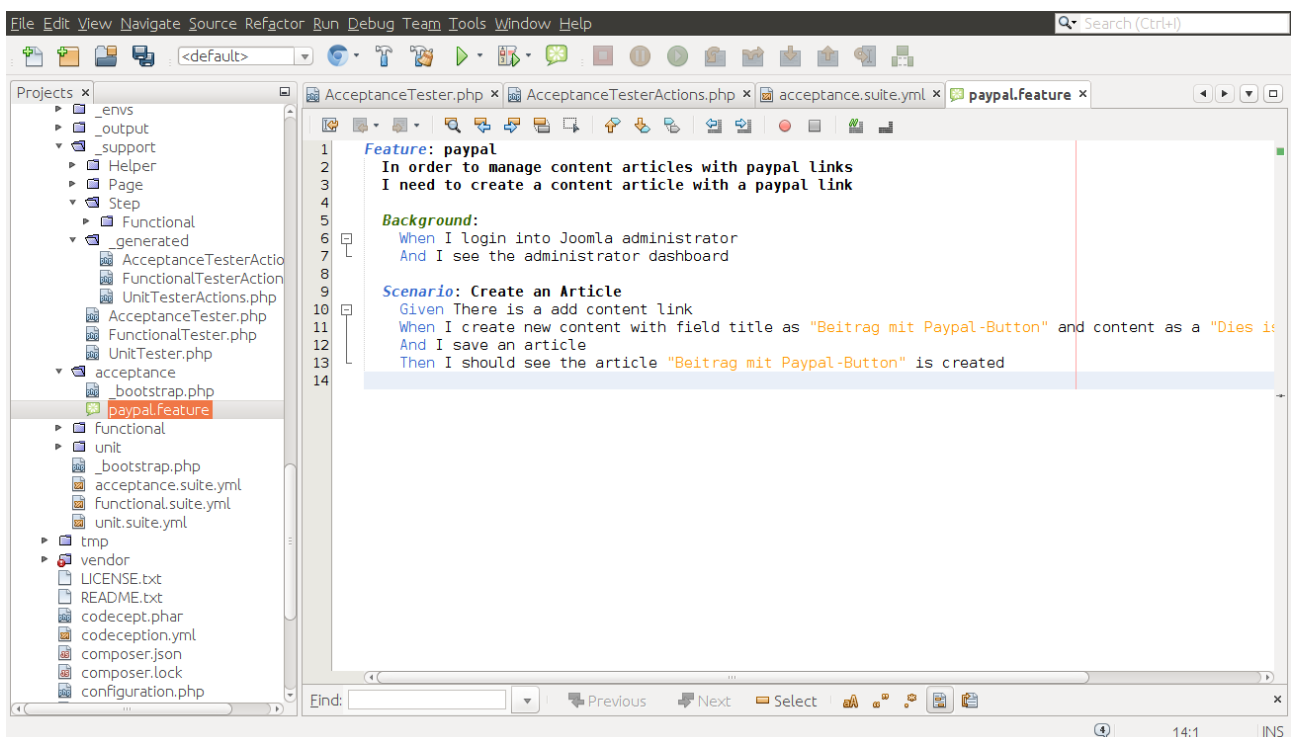


Abbildung 50: Die Ansicht einer Feature Datei in Netbeans mit installiertem Cucumber Plugin. 962.png

Starten Sie nun mit dem Befehl `vendor/bin/codecept run acceptance` einen Testlauf für dieses Szenario.

```
//var/www/html/joomla$ vendor/bin/codecept run acceptance
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.15 by Sebastian Bergmann and contributors.
Acceptance Tests (1) -----
I paypal: Create an Article
```



```
-----  
Time: 166 ms, Memory: 10.00MB
```

OK, but incomplete, skipped, or risky tests!

```
Tests: 1, Assertions: 0, Incomplete: 1.
```

```
run with '-v' to get more info about skipped or incomplete tests.
```

Das war zu erwarten. Der Test ist zwar nicht fehlgeschlagen – erfolgreich war er aber auch nicht. Die Testroutine hat gemerkt, dass noch kein wirklicher Testcode ausgeführt wird.

Damit echter Testcode ausgeführt werden kann, muss dieser nun erstellt werden. Konkret bedeutet dies, dass wir für jede Zeile im Szenario eine Testmethode erstellen müssen. Diese Testmethoden müssen eine bestimmte Syntax einhalten. Machen Sie es sich einfach. Nutzen Sie den Codeschnipsel Generator von Codeception. Dafür müssen Sie nur den Befehl `vendor/bin/codecept gherkin:snippets acceptance` eingeben. In der Ausgabe finden Sie nun alle Methodenrumpfe in der korrekten Syntax.

```
/var/www/html/joomla$ vendor/bin/codecept gherkin:snippets acceptance
```

```
Snippets found in:
```

```
- paypal.feature
```

```
Generated Snippets:
```

```
-----  
/**  * @Given There is a add content link  */  
public function thereIsAAddContentLink() {  
    throw new \Codeception\Exception\Incomplete("Step `There is a add content link` is not  
defined");  
}  
/**  * @When I create new content with field title as :arg1 and content as a :arg2  */  
public function iCreateNewContentWithFieldTitleAsAndContentAsA($arg1, $arg2) {  
    throw new \Codeception\Exception\Incomplete("Step `I create new content with field title as  
:arg1 and content as a :arg2` is not defined");  
}  
/**  * @When I save an article  */  
public function iSaveAnArticle() {  
    throw new \Codeception\Exception\Incomplete("Step `I save an article` is not defined");  
}  
/**  * @Then I should see the article :arg1 is created  */
```

```

public function iShouldSeeTheArticleIsCreated($arg1) {
    throw new \Codeception\Exception\Incomplete("Step `I should see the article :arg1 is created` is
not defined");
}

/**  * @When I login into Joomla administrator  */
public function iLoginIntoJoomlaAdministrator() {
    throw new \Codeception\Exception\Incomplete("Step `I login into Joomla administrator` is not
defined");
}

/**  * @When I see the administrator dashboard  */
public function iSeeTheAdministratorDashboard() {
    throw new \Codeception\Exception\Incomplete("Step `I see the administrator dashboard` is not
defined");
}
}
-----
6 snippets proposed
Copy generated snippets to AcceptanceTester or a specific Gherkin context

```

Codeception hat Ihnen sechs Methoden vorgeschlagen. Diese Codeschnipsel können Sie kopieren und in eine Step-Klasse einfügen. Dazu erzeugen wir zunächst die Step-Klasse, die die Methoden aufnehmen soll. Mit dem Befehl `vendor/bin/codecept generate:step acceptance Paypal` erstellen Sie eine leere Step-Klasse im Verzeichnis `/joomla/tests/_support/Step/Acceptance` in der Datei `Paypal.php`.

```

/var/www/html/joomla$ vendor/bin/codecept generate:step acceptance Paypal
Add action to StepObject class (ENTER to exit):
StepObject was created in /var/www/html/joomla/tests/_support/Step/Acceptance/Paypal.php

```

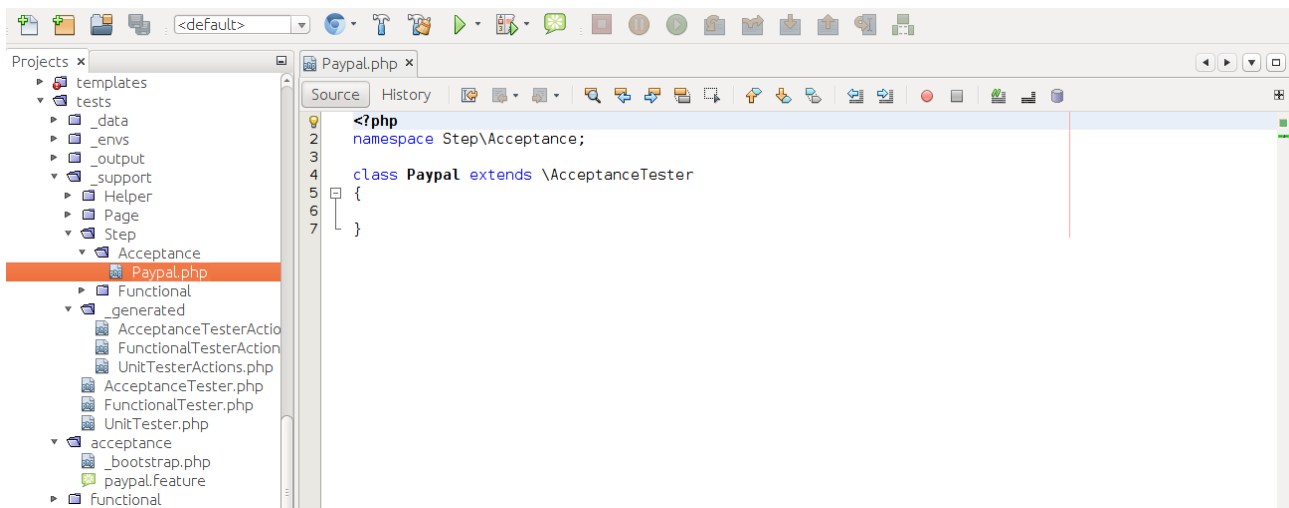


Abbildung 51: Hier sehen Sie die mithilfe des Generators erzeugte Step-Klasse unmittelbar nach der Erstellung. 964.png

Fügen Sie nun die vorher mit dem Befehl `vendor/bin/codecept gherkin:snippets acceptance` erstellten Codeschnipsel in die gerade erzeugte Step-Klasse ein. Falls Sie diese nicht mehr in der Kommandozeile sehen, können Sie die Schnipsel neu erstellen. Geben Sie dazu den Befehl `vendor/bin/codecept gherkin:snippets acceptance` einfach noch einmal ein.

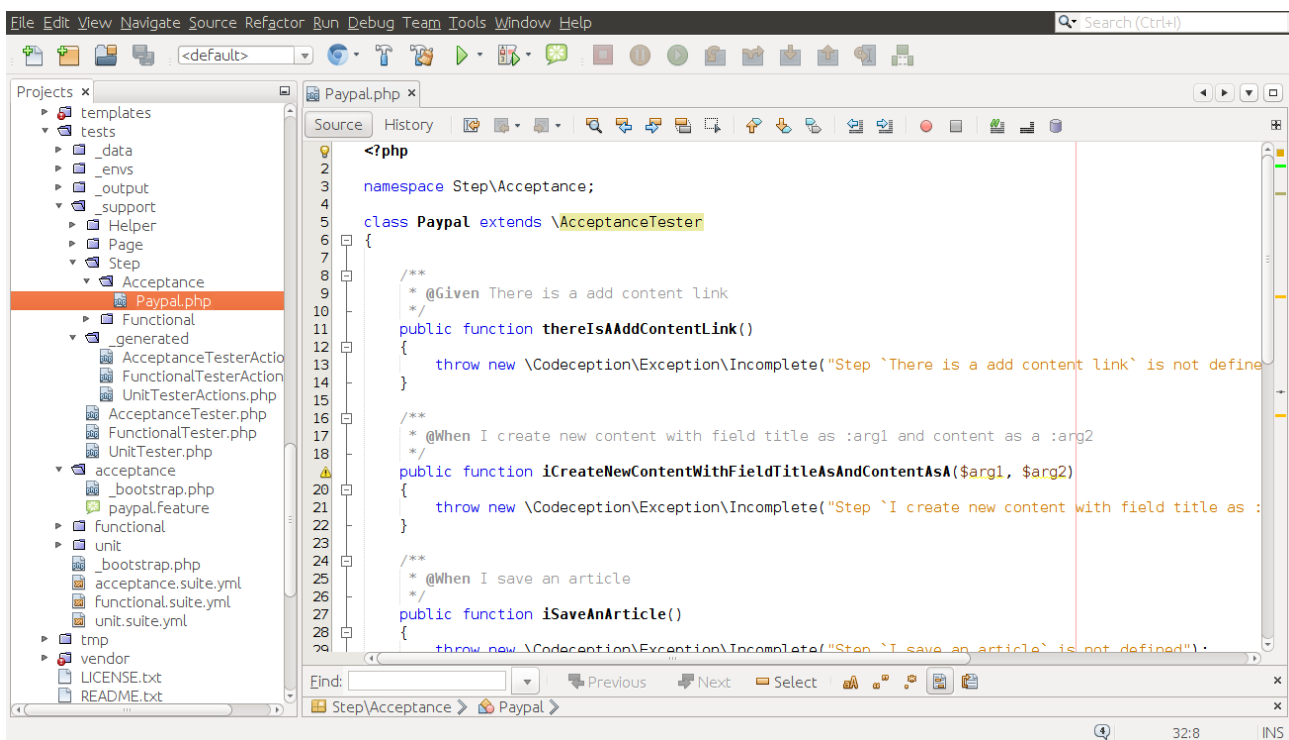


Abbildung 52: Hier sehen Sie die mithilfe des Generators erzeugte Step-Klasse nach dem Einfügen der Codeschnipsel. 961.png

Wenn Sie nun mit dem Befehl `vendor/bin/codecept run acceptance` erneut einen Testlauf für dieses Szenario starten, hat dies noch keine Auswirkungen auf das Testergebnis.

```
//var/www/html/joomla$ vendor/bin/codecept run acceptance
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.15 by Sebastian Bergmann and contributors.
Acceptance Tests (1) -----
I paypal: Create an Article
-----
Time: 166 ms, Memory: 10.00MB
OK, but incomplete, skipped, or risky tests!
Tests: 1, Assertions: 0, Incomplete: 1.
run with `-v` to get more info about skipped or incomplete tests.
```

Genau wie eben schlagen die Tests nicht fehl – sie sind aber auch nicht erfolgreich. Das Einfügen der Schnipsel bringt Ihnen zunächst nur folgenden Vorteil: Sie können über einen Befehl abfragen, ob alle notwendigen Methoden nun in Ihren Tests vorhanden sind. Hierfür müssen Sie allerdings noch einen Eintrag in der Konfiguration vornehmen. Fügen Sie in die Datei `/joomla/tests/acceptance.suite.yml` den, im folgenden Codebeispiel fett hervorgehobenen, Gherkin-Block ein. Den build-Befehl müssen Sie nicht ausführen, da Sie kein weiteres Modul laden.

```
class_name: AcceptanceTester
modules:
  enabled:
    - WebDriver
    - \Helper\Acceptance
  config:
    WebDriver:
      url: 'http://localhost/joomla'
      browser: 'chrome'
      window_size: 1024x768
      restart: true
gherkin:
  contexts:
    default:
      - Step\Acceptance\Paypal
```

Alle Feature-Dateien, die im Gherkin Block der Konfigurationsdatei `/joomla/tests/acceptance.suite.yml` eingetragen sind, werden beim Aufruf des Befehls `vendor/bin/codecept gherkin:snippets acceptance` durchsucht. Wenn Sie nun noch einmal diesen Befehl eingeben, wird Codeception Ihnen mitteilen, dass alle Gherkin-Steps mit einer Testmethode verbunden sind.

```
/var/www/html/joomla$ vendor/bin/codecept gherkin:snippets acceptance
All Gherkin steps are defined. Exiting...
```

So, nun wird es spannend. Als Nächstes füllen wir die als Codeschnipsel eingefügten Methoden mit echtem Testcode. Vorher sehen wir uns noch kurz zwei häufige Fehlermeldungen bei der Verwendung von [Selenium Standalone Server](#) und [ChromeDriver](#) an.

Mögliche Fehlermeldungen

Wird Ihnen eine Fehlermeldung angezeigt, wenn Sie den Befehl `vendor/bin/codecept run acceptance` eingeben? Diesen Fehler müssen Sie erst klären, bevor Sie in diesem Kapitel weiter arbeiten können. Falls Selenium Standalone Server bei Ihnen fehlerfrei läuft, können Sie im nächsten Kapitel mit der Implementierung der Testmethoden weiter machen.

Der WebDriver wird nicht gefunden

Wenn Ihnen eine Fehlermeldung angezeigt wird, die sinngemäß aussagt, dass der WebDriver nicht gefunden werden kann, sollten sie überprüfen, ob der Treiber und der symbolische Link richtig gesetzt sind. Erklärungen hierzu finden Sie zu Beginn dieses Kapitels im Abschnitt *Selenium und ChromeDriver installieren*.

```
/var/www/html/joomla$ vendor/bin/codecept run acceptance
Codeception PHP Testing Framework v2.2.9
...
The path to the driver executable must be set by the webdriver.chrome.driver system property;
for more information, see https://github.com/SeleniumHQ/selenium/wiki/ChromeDriver. The
latest version can be downloaded from http://chromedriver.storage.googleapis.com/index.html
```

Die Verbindung zum WebDriver kann nicht hergestellt werden

Können Sie die Verbindung zum WebDriver nicht herstellen?

```
/var/www/html/joomla$ vendor/bin/codecept run acceptance
Codeception PHP Testing Framework v2.2.9
...
Can't connect to Webdriver at http://127.0.0.1:4444/wd/hub. Please make sure that Selenium
Server or PhantomJS is running.
```

Überprüfen Sie in diesem Falle, ob der Selenium Standalone Server Prozess tatsächlich aktiv ist. Wie machen Sie dies am besten? Zum einen können Sie die Ausgabe in der Kommandozeile beim Absetzen des Startbefehls überprüfen.

Eine andere Möglichkeit ist im Browser die Adresse `http://localhost:4444/` aufzurufen. Falls Sie den Selenium Standalone Server so wie ich es im Kapitel *Selenium und ChromeDriver installieren* beschrieben habe installiert haben, müssten Sie eine Informationsseite dieses Servers angezeigt bekommen. Wenn diese angezeigt wird bedeutet dies, dass der Prozess richtig läuft.

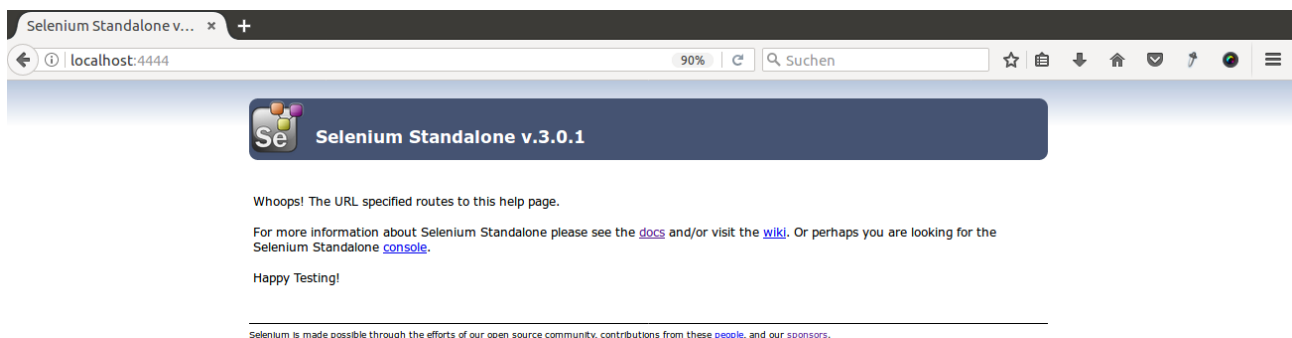


Abbildung 53: Die Startseite des Selenium Standalone Servers. 958a.png

Versuche Sie in diesem Fall per Klick auf den Link `console` die Selenium Konsole zu öffnen. Vielleicht finden Sie hier einen Hinweis auf eine mögliche Fehlerursache.

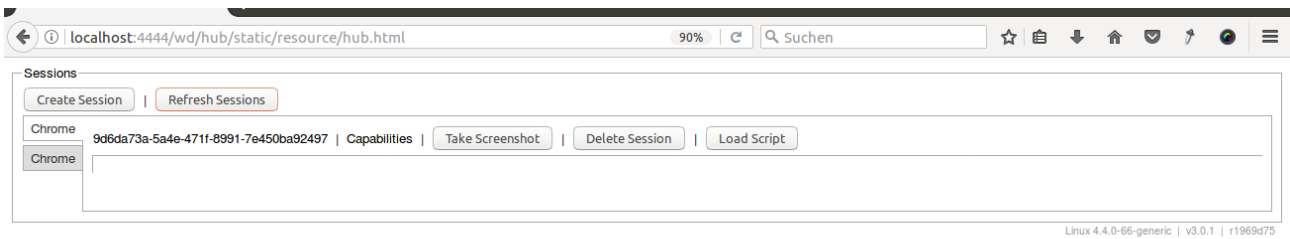


Abbildung 54: Die Konsole des Selenium Standalone Servers. 958b.png

Wenn der Selenium Prozess nicht läuft, sollten Sie zunächst die Installation des Selenium Standalone Servers überprüfen und diesen dann erneut zu starten. Informationen hierzu finden Sie im Kapitel *Selenium und ChromeDriver installieren*.

Implementatierung der Testmethoden

Nun geht es darum, die leeren Methoden – also die Codeschnipsel – mit sinnvollem Programmcode zu füllen. In der Datei

joomla/tests/_support/_generated/AcceptanceTesterActions.php finden wir eine ganze Menge Methoden, die wir über die Klasse AcceptanceTester nutzen können.

Automatisch generierte Methoden nutzen

Die Methoden in der Datei joomla/tests/_support/_generated/AcceptanceTesterActions.php benötigen, analog zur Datei joomla/tests/_support/_generated/FunctionalTesterActions.php die Sie im letzten Kapitel kennengelernt haben, als Eingabeparameter teilweise lediglich einen Text. Zum Beispiel kann mit der Methode `see(„Willkommen“)` geprüft werden, ob auf einer Website an einer beliebigen Stelle der Text „Willkommen“ angezeigt wird. Wenn Sie sicherstellen wollen, dass ein Text sich in einem bestimmten [Selektor](#) befindet, kommen Sie um die Angabe eines Selektors aber nicht herum.

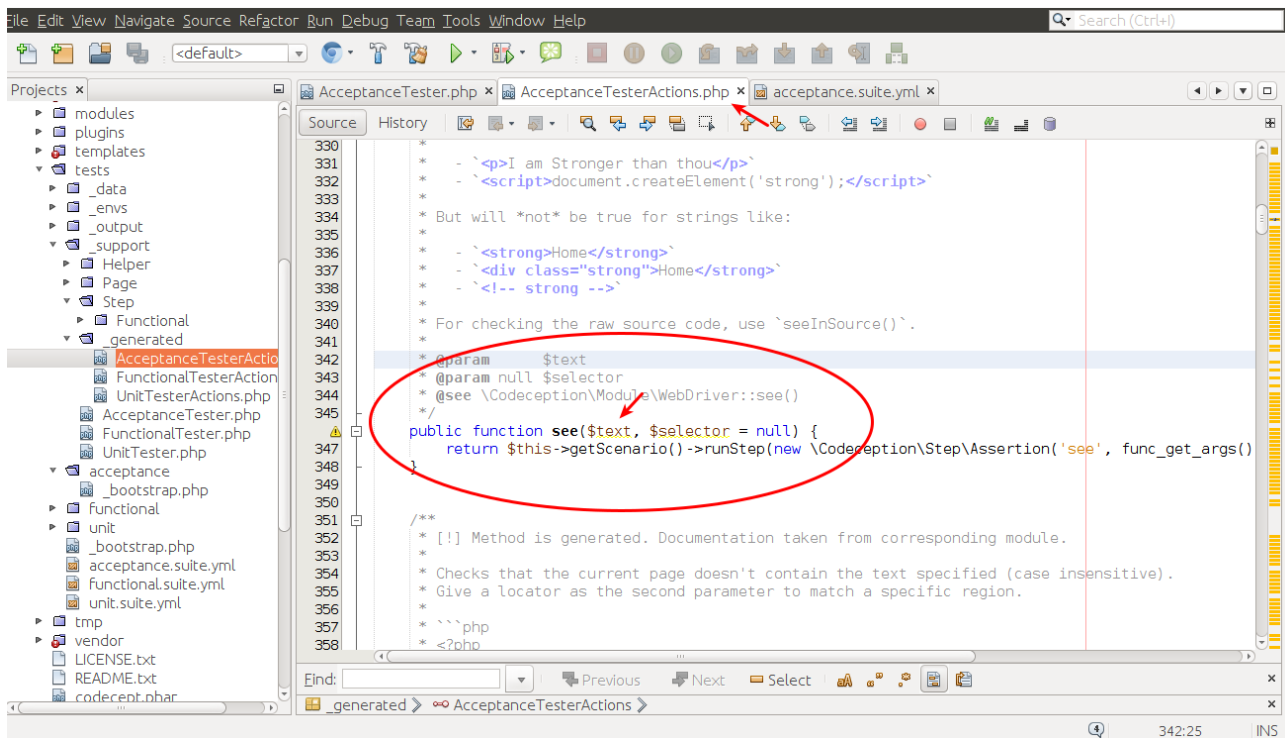


Abbildung 55: Die Methode `seeElement()` in der Datei `AcceptanceTesterActions.php` benötigt als Parameter einen Text - genauer kann allerdings mit der zusätzlichen Eingabe eines Selektors getestet werden. 963.png

Bei vielen Methoden ist die Angabe des Selektors sogar Pflicht. Codeception greift Ihnen bei der Suche nach dem Selektor allerdings so gut wie möglich unter die Arme. Sie können Selektoren ungenau beschreiben. Wobei aber auch hier gilt: Je genauer Sie den Selektor beschreiben, desto schneller wird der Test abgearbeitet und desto korrekter ist das Testergebnis. Mit den nachfolgenden Erläuterungen will ich Sie nun nicht zum ungenauen Arbeiten ermutigen, sondern Ihnen nur die Arbeitsweise von Codeception erklären.

Was passiert also, wenn Sie eine Methode mit einem ungenauen Suchwort für den Selektor aufrufen – wenn Sie also beispielsweise Ihren *AcceptanceTester* mit `see(„Willkommen“, „title“)`; ausführen? Wie in der vorhergehenden Abbildung erkennbar, können Sie der Methode `see()` als zweiten Eingabeparameter optional einen Selektor mitgeben. Wie gesagt, Codeception unterstützt Sie so gut es geht und versucht herauszufinden, welchen Selektor Sie mit „title“ meinten:

1. Codeception wird als Erstes nach einem Element mit der ID `#titel` suchen.
2. Wenn es kein Element mit der ID `#title` findet, setzt Codeception die Suche nach einem Element mit der Klasse `.title` fort.

3. Ist Codeception immer noch nicht fündig geworden, wird der Text „title“ als [XPath](#) interpretiert.
4. Wenn alle Suchversuche negativ ausfallen, wird Codeception Ihnen in einer Fehlermeldung mitteilen, dass es das Element nicht finden konnte und auch den Test als fehlgeschlagen kennzeichnen.

Sie möchten sicherlich den Selektor so genau wie möglich beschreiben, oder? Dies können Sie wie folgt:

- Die Beschreibung

['id' => 'title']

findet das Element mit der ID „title“ – zum Beispiel das Element

<div id="title">

- Die Beschreibung

['name' => 'title']

findet alle Elemente die das Attribut name="title" haben, also beispielsweise Element

<div name="title">

- Die Beschreibung

['css' => 'input[type=input][value=title]']

findet das Element

<input type="input" value="title">

- Die Beschreibung

['xpath' => '//input[@type='submit'][contains(@value, 'title')]']

findet das Element

<input type="submit" value="titlebar">

RANDBEMERKUNG:

Die [XML Path Language \(XPath\)](#) ist eine vom [W3-Konsortium entwickelte Abfragesprache](#), um Teile eines XML-Dokumentes zu adressieren und auszuwerten.

- Die Beschreibung

['link' => 'Click here']

findet das Element

```
<a href="codeception.com">Click here</a>
```

- Die Beschreibung

```
['class' => 'title']
```

findet ein Element der Klasse „title“, zum Beispiel das Element

```
<div class="title">
```

Weitere Informationen hierzu können Sie in der Dokumentation von Codeception [im Bereich WebDriver](#) nachlesen.

Ein konkretes Beispiel

Das Ganze sehen wir uns jetzt an einem konkreten Beispiel an. Wir füllen nun die im Kapitel *Gherkin* kopierten Codeschnipsel. Achten Sie dabei darauf, dass Sie die Kommentare der Methoden nicht bearbeiten.

```
/**  
 * @Given There is a add content link  
 */
```

Anhand der Annotation in diesem Kommentar verbindet Codeception die zum Kommentar gehörende Methode der Testdatei, in unserm Fall heißt die Methode `thereIsAAddContentLink()`, mit dem Step `Given There is a add content link` in der Feature-Datei. Den Namen der Methode und eventuelle Eingabeparameter dürfen Sie beliebig verändern. Wenn Ihnen `thereIsAAddContentLink()` nicht gefällt oder zu lang ist, können Sie diesen Namen modifizieren. Ich habe im nachfolgenden Programmcodebeispiel alle Methoden gefüllt. Meiner Meinung nach sind die Methoden selbsterklärend. Kopieren Sie meinen Testcode und probieren diesen in Ihrer Testumgebung aus.

```
<?php  
namespace Step\Acceptance;  
class Paypal extends \AcceptanceTester{  
    /**  
     * @Given There is a add content link  
     */  
    public function thereIsAAddContentLink(){  
        $I = $this;  
        $I->amOnPage("/administrator/index.php?option=com_content&view=articles");
```

```

        $I->click(['xpath' => "//div[@id='toolbar-new']/button"]);
    }
/**
 * @When I create new content with field title as :arg1 and content as a :arg2
 */
public function iCreateNewContentWithFieldTitleAsAndContentAsA($title, $content){
    $I = $this;
    $I->fillField(['id' => 'jform_title'], $title);
    $I->scrollTo(['css' => 'div.toggle-editor']);
    $I->click("Editor an/aus");
    $I->fillField(['id' => 'jform_articletext'], $content);
}
/**
 * @When I save an article
 */
public function iSaveAnArticle()
{
    $I = $this;
    $I->click(['xpath' => "//div[@id='toolbar-apply']/button"]);
}
/**
 * @Then I should see the article :arg1 is created
 */
public function iShouldSeeTheArticleIsCreated($item)
{
    $I = $this;
    $I->see('Der Beitrag wurde gespeichert!', ['id' => 'system-message-container']);
}
/**
 * @When I login into Joomla administrator
 */
public function iLoginIntoJoomlaAdministrator()
{
    $I = $this;
    $I->amOnPage("/administrator/index.php");
    $I->fillField(['id' => 'mod-login-username'], 'admin');
    $I->fillField(['id' => 'mod-login-password'], 'admin');
    $I->click(['xpath' => "//button[contains(normalize-space(), 'Anmelden')]"]);
}

```

```

}
/**
 * @When I see the administrator dashboard
 */
public function iSeeTheAdministratorDashboard(){
    $I = $this;
    $I->waitForText('Kontrollzentrum');
    $I->see('Kontrollzentrum', ['class' => 'page-title']);
}

```

Wenn alles richtig läuft, dann öffnet sich mit der Eingabe des Befehls `vendor/bin/codecept run acceptance` Ihr Browser und die Schritte, die wir in der Feature-Datei benannt und in den Testmethoden codiert haben, werden automatisch ausgeführt. Sie können die einzelnen Schritte im Browser mitverfolgen. Das Ergebnis sollte folgende Ausgabe sein:

```

/var/www/html/joomla$ vendor/bin/codecept run acceptance
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.17 by Sebastian Bergmann and contributors.
Acceptance Tests (2) -----
✓ paypal: Create an Article (6.87s)
-----
Time: 13.1 seconds, Memory: 10.00MB

```

RANDBEMERKUNG:

Ich möchte hier in diesem Kapitel den Programmcode so kurz wie möglich halten und den Fokus auf die Besonderheiten der Akzeptanztests legen. Deshalb habe ich auf das Erstellen von Page-Objekten und Step-Objekten verzichtet. Wie und warum Sie diese Objekte erstellen hatte ich im Kapitel *Funktionstest* | *Wiederverwendbare Tests* beschrieben.

Die Grenzen von Selenium

Akzeptanztests werden realitätsnah durchgeführt. Der Ablauf ist genauso, wie ihn ein Mensch tätigen würde. Sie sparen eine Menge Zeit, die Sie selbst mit Tests erbringen würden. Kein Mensch muss wiederholt ein und dasselbe Formular ausfüllen. Selenium

WebDriver macht dies anstandslos. Leider gibt es aber ein paar Dinge, die Selenium WebDriver nicht sicher stellen kann. Zum Beispiel fast alles, was mit Design zu tun hat. Ist die Farbzusammenstellung harmonisch? Ist das Layout ausgewogen und entspricht einem natürlichen Gleichgewicht oder passen die Proportionen der einzelnen Elemente nicht zusammen? Kriterien in diesem Bereich kann man einer Maschine nicht mitgeben. Und ob eine Website auf allen Geräten gut lesbar und übersichtlich ist, also ob sie responsive ist, lässt sich auch mit einem WebDriver nicht sicher testen.

RANDBEMERKUNG:

Welche Qualitätsmerkmale sind testbar? Die **Funktionalität** und die **Zuverlässigkeit** können Sie systematisch und objektiv testen. Die **Benutzbarkeit** und die **Effizienz** sind hingegen nur teilweise testbar – sofern passende Maße vorliegen. Bei der **Veränderbarkeit, der Ästhetik** und der **Übertragbarkeit** stoßen Tests allerdings an ihre Grenzen. Diese Merkmale sind nicht testbar.

Kurzgefasst

In diesem Kapitel haben Sie gelernt, wie Sie Akzeptanztests erstellen und mithilfe des Frameworks Selenium automatisch in einem Browser ablaufen lassen können. Sie haben nun alle Testtypen kennen gelernt. Im nächsten Kapitel geht es nun darum, wie Sie diese Tests messen und verbessern können.

Analyse

Unit Testing gives you the what.

Test-Driven-Development gives you the when.

Behaviour Driven-Development gives you the how.

[Jani Hartikainen] Quelle: <https://entwickler.de>

In den vorhergehenden Kapiteln haben wir uns die verschiedenen Testtypen angesehen. Sie wissen nun was ein Unittest, was ein Funktionstest und was ein Akzeptanztest ist. Außerdem haben Sie das notwendige Handwerkszeug, um jede dieser Testarten zu erstellen. Es wäre aber nicht sinnvoll, einfach so ins Blaue hinein

Testdateien zu erzeugen. Außerdem möchten Sie sicherlich auch gerne wissen, ob Sie den zu Beginn im Kapitel *Softwaretests – eine Einstellungssache?* | *Tests planen* aufgestellten Testplan erfüllen. Wie können Sie sicher sein, dass Sie genügend relevante Tests geschrieben haben?

Es gibt Kennzahlen, die Sie zur Beurteilung Ihrer Software heranziehen können und Codeception integriert Werkzeuge, die Ihre Software anhand dieser Kennzahlen analysieren. Außerdem integriert Codeception Werkzeuge, die Sie bei der Erstellung von Reports unterstützen. Beides zusammen trägt maßgeblich dazu bei, die Qualität Ihrer Tests und die Qualität der zu testenden Software zu verbessern.

Sie sollten allerdings immer im Hinterkopf behalten, dass Sie auch mit Tests, die nichts Wichtiges überprüfen, eine hohe Codeabdeckung erreichen können. Deshalb hat die Codeabdeckung für die Qualität der Tests nur eine eingeschränkte Aussagekraft.

Die Codequalität verbessern – welche Kriterien gibt es?

„Nichts auf der Welt ist so gerecht verteilt wie der Verstand, denn jedermann ist überzeugt, dass er genug davon habe“. Diesen Satz hat der französische Mathematiker und Philosoph Rene Descartes schon vor ungefähr 400 Jahren geprägt. Und tatsächlich gibt es eine ganze Menge Menschen, die wissen, wie man Programme besser machen kann und Kriterien hierfür aufstellen. Ich habe mir diejenigen [Kriterien](#), die auf der Website von PHPUnit für das Testen von Software beschrieben sind, genauer angesehen.

1. Linienabdeckung oder Zeilenüberdeckung:

Der Wert zur Linienabdeckung gibt an, inwieweit ein Test jede ausführbare Zeile der zu testenden Software ausgeführt hat. Beispiel: Eine Software beinhaltet Tests, die bei einem Durchlauf 500 Zeilen der Software ausführen. Wenn die Software insgesamt aus 1000 Zeilen besteht, ist die Linienabdeckung zu 50 % erreicht.

Bei der Zeilenüberdeckung werden nicht die Anweisungen betrachtet, sondern nur die ausführbaren Quellcodezeilen. Beliebige Testfälle für

```
if(false){print "abgedeckt?";}
```

würden zu einer Zeilenüberdeckung von 100 % führen, während

```
if(false){  
    print "abgedeckt?";  
}
```

}

nur zu einer Zeilenüberdeckung von 50 % führt.

2. Funktions- oder Methodenabdeckung:

Die Funktions- oder Methodenabdeckung misst, wie viele Funktionen oder Methoden vom Test aufgerufen wurde. Eine Funktion oder eine Methode wird nur dann berücksichtigt, wenn die Linienabdeckung 100 % beträgt.

3. Klassen und Trait Abdeckung:

Die Klassen und Trait Abdeckung misst, wie viele Methoden einer Klasse oder eines Traits aufgerufen wurde. Eine Klasse oder ein Trait wird nur dann berücksichtigt, wenn die Linienabdeckung 100 % beträgt.

4. Anweisungsabdeckung:

Die Anweisungsabdeckung misst, ob jede Anweisung einer Funktion oder einer Methode während des Testdurchlaufs ausgeführt wurde.

Zum Beispiel könnte der nachfolgende Programmcode mit nur einem Test eine Anweisungsüberdeckung von 100 % erreichen – nämlich immer dann, wenn z größer als x ist.

```
int z = 5;  
    if (z > x)  
        z = x;  
x *= 2;
```

5. Zweigabdeckung:

Die Zweigabdeckung geht einen Schritt weiter als die Anweisungsüberdeckung. Mit Hilfe der Zweigabdeckung lassen sich nicht ausführbare Programmzweige finden. Anhand dessen können Sie Softwareteile, die oft durchlaufen werden, gezielt optimieren. Sehen wir uns das Beispiel zur Anweisungsabdeckung noch einmal an:

```
int z = 5;  
    if (z > x)  
        z = x;  
x *= 2;
```

Im Gegensatz zum Anweisungsüberdeckungstest ist nun mehr als ein Testfall notwendig um 100 % zu erreichen. Beim Messen der Zweigabdeckung müssen Sie das Programm sowohl mit der Variablenbelegung $z > x$, als auch mit der Variablenbelegung $z < x$ prüfen, um eine hundertprozentige Abdeckung zu erreichen.

6. Pfadabdeckung:

Beim Pfadabdeckungstest werden die möglichen Pfade vom Startknoten bis zum Endknoten betrachtet.

7. Änderungsrisiko (Change Risk Anti-Patterns – CRAP):

Der CRAP-Index ist ein Maß für die Beurteilung der Wartbarkeit von Software. Das Änderungsrisiko wird auf der Grundlage der [McCabe-Metrik](#), also der Komplexität, und der [Testabdeckung](#) einer Software berechnet. Programmcode, der nicht sehr komplex ist und eine hohe Testabdeckung hat, hat einen niedrigen CRAP-Index. Wohingegen der CRAP-Index eines komplexen Programms mit niedriger Testabdeckung hoch ist. Sie können also durch das Hinzufügen von Tests und/oder durch eine Verbesserung der Struktur des Programmcodes den CRAP-Index verbessern.

Wann ist die Struktur eines Programmcode komplex? Vereinfacht ausgedrückt ist eine Software komplex, wenn es viele Verzweigungen innerhalb des Programmcodes gibt. Genauer können Sie dies aber auf der Website verifysoft.com nachlesen.

Codeception nutzt zur Erstellung von Reports zur Testabdeckung – genau wie PHPUnit – das Paket [PHP_Codecoverage](#). Die Kennzahlen Anweisungsabdeckung, Zweigabdeckung und Pfadabdeckung werden von PHP_Codecoverage momentan noch nicht unterstützt.

Unterstützt werden aber die Kennzahlen Linienabdeckung oder Zeilenüberdeckung, Funktions- oder Methodenabdeckung, Klassen oder Trait Abdeckung und das Änderungsrisiko (CRAP). Die Werte hierzu finden Sie später auf der Übersichtseite des Reports zur Testabdeckung.

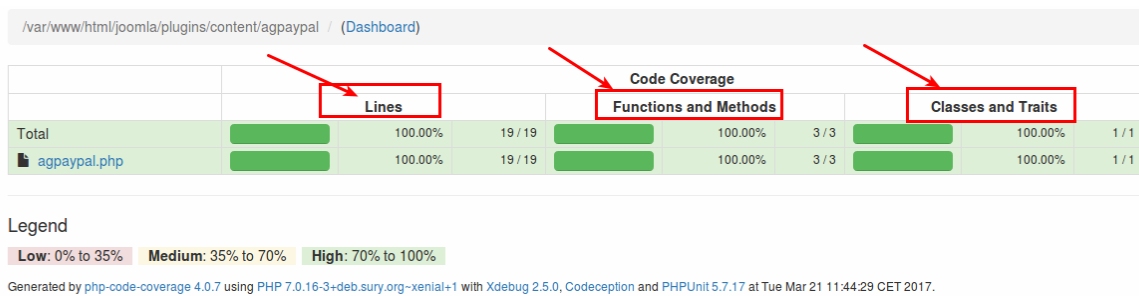


Abbildung 56: Codeception nutzt zur Erstellung von Reports zur Testabdeckung das Paket PHP_Codecoverage. 957d.png

Wenn Sie auf der Übersichtseite eine spezielle Datei auswählen, finden Sie in der Detailansicht die Kennzahl CRAP.

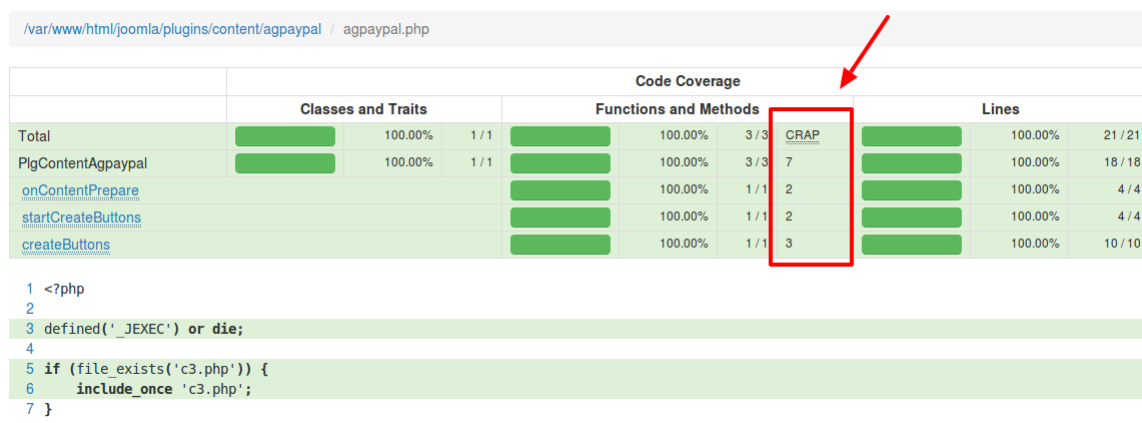


Abbildung 57: Der CRAP-Index ist ein Maß für die Beurteilung der Wartbarkeit von Software. 956.png

Codeception Konfiguration – wie aktivieren Sie die Messung der Testabdeckung

Irgendwo müssen wir beginnen. Deshalb aktivieren wir als Erstes die Messung der Testabdeckung für Unittests. Dies tun wir in der Datei `/var/www/html/joomla/codeception.yml`. Fügen Sie bitte den im nachfolgenden Beispiel fett hervorgehobenen Text in die Konfigurationsdatei `/var/www/html/joomla/codeception.yml` ein. Die fett hervorgehobenen Zeilen bewirken, dass bei zukünftigen Testdurchläufen die Messung der Testabdeckung für alle Dateien im Verzeichnis `plugins/content/agpaypal` aktiviert ist.

```

actor: Tester
paths:
  tests: tests

```

```
log: tests/_output
data: tests/_data
support: tests/_support
envs: tests/_envs
settings:
  bootstrap: _bootstrap.php
  colors: true
  memory_limit: 1024M
extensions:
  enabled:
    - Codeception\Extension\RunFailed
modules:
  config:
    Db:
      dsn: "
      user: "
      password: "
      dump: tests/_data/dump.sql
coverage:
  enabled: true
  include:
    - plugins/content/agpaypal/*
```

Sie können in der Konfiguration noch weitere Einstellungen vornehmen. Interessant sind sicherlich noch die Parameter `exclude`, `low_limit` und `high_limit`. In der Codeception Dokumentation finden Sie im Bereich [Codecoverage](#) weitere Informationen hierzu.

Die Testabdeckung messen

Ich experimentiere gerne und habe als Erstes den vorhandenen Unittest gelöscht. Genauer gesagt habe ich ihn in ein anderes Verzeichnis verschoben, weil ich ihn später wieder nutzen will. Danach habe ich den Befehl `vendor/bin/codecept run unit --coverage` ausgeführt.

RANDBEMERKUNG:

Achten Sie darauf, dass Sie den Parameter `coverage` mit zwei vorangestellten Minuszeichen an den Befehl zum Start der Tests anhängen.

```

/var/www/html/joomla$ vendor/bin/codecept run unit --coverage
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.15 by Sebastian Bergmann and contributors.
Unit Tests (0) -----
Time: 142 ms, Memory: 8.00MB
No tests executed!
Code Coverage Report:
2017-03-21 10:05:28
Summary:
Classes: 0.00% (0/1)
Methods: 0.00% (0/3)
Lines: 0.00% (0/35)
Remote CodeCoverage reports are not printed to console

```

Die Testabdeckung liegt bei 0 %. Das passt. Momentan gibt es ja auch keinen Test. Stimmen aber auch die anderen Werte Classes: 0.00% (0/1), Methods: 0.00% (0/3), Lines: 0.00% (0/35)? In dem Verzeichnis, in dem wir die Testabdeckung messen möchten ist

- eine Klassen (Classes: 0.00% (0/1)), nämlich die Klasse PlgContentAgpaypal,
- mit drei Methoden (Methods: 0.00% (0/3)), nämlich den Methoden onContentPrepare(), startCreateButtons() und createButtons().
- Die Anzahl der Zeilen (Lines: 0.00% (0/35)) in der Datei passt nicht ganz. Das sehen wir uns aber später genauer an.

Ich habe Ihnen das Plugin nachfolgend noch einmal abgedruckt, damit Sie leichter vergleichen können:

```

<?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){
    preg_match_all('/@paypal(?:.*)paypal@/i', $text, $matches, PREG_PATTERN_ORDER);
    if (count($matches[0] > 0)){
        $this->createButtons($matches, $text);
    }
    return true;
}

private function createButtons($matches, &$text){
    for ($i = 0; $i < count($matches[0]); $i++){
        $search = $matches[0][$i];
        if ((strpos($matches[1][$i], 'amount=') !== false) > 0){
            $amount = trim(str_replace('amount=', '', $matches[1][$i]));
        }else{
            $amount = '12.99';
        }
        $replace = '<form name="_xclick" action="https://www.paypal.com/de/cgi-
bin/webscr" method="post">
        .<input type="hidden" name="amount" value="' . $amount . "'>
        .<input type="image" src="http://www.paypal.com/de_DE/i/btn/x-click-
but01.gif" border="0" name="submit">'
        .</form>';
        $text = str_replace($search, $replace, $text);
    }
    return true;
}
}

```

Testabdeckung genauer betrachtet

Und wie ist nun die Testabdeckung unseres Tests? Um dies zu überprüfen, kopiere ich den Unittest wieder zurück in die richtige Test-Suite und gebe den Befehl

vendor/bin/codecept run unit --coverage erneut ein. Nachfolgend sehen Sie, um

Missverständnisse zu vermeiden, noch einmal den gesamten Testcode der Datei

/tests/unit/suites/plugins/content/agpaypal/agpaypalTest.php.

```
<?php
namespace suites\plugins\content\agpaypal;
use \Codeception\Util\Fixtures;
use \Codeception\Verify;
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, $hint);
            verify($contenttextafter)->equals($contenttextbefore);
        }, ['examples' => $this->provider_PatternToPaypalbutton()]);
    }
    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>',
        'hint' => 'Zwei Paypalbuttons)],
        [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="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>',
        'hint' => 'Ein Paypalbutton mit Betrag)],
        [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>',
    'hint' => 'Zwei Paypalbuttons']],
    [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>',
    'hint' => 'Ein Paypalbutton ohne Betrag']],
    [array('contenttextbefore' => "jsdkl",
    'contenttextafter' => "jsdkl",
    'hint' => 'Kein Paypalbutton']],
    [array('contenttextbefore' => "",
    'contenttextafter' => "",
    'hint' => 'Beitrag enthält keinen Text')]
];
}
}

```

Ein erneuter Durchlauf, diesmal inklusive Test, bringt folgende Ausgabe zutage:

```

/var/www/html/joomla$ vendor/bin/codecept run unit --coverage
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.15 by Sebastian Bergmann and contributors.
Unit Tests (7) -----
✓ agpaypalTest: On content prepare method runs one time (0.10s)
✓ agpaypalTest: Start create buttons | #0 | #0 (0.21s)
✓ agpaypalTest: Start create buttons | #1 | #1 (0.20s)

```



```
✓ agpaypalTest: Start create buttons | #2 | #2 (0.21s)
✓ agpaypalTest: Start create buttons | #3 | #3 (0.20s)
✓ agpaypalTest: Start create buttons | #4 | #4 (0.20s)
✓ agpaypalTest: Start create buttons | #5 | #5 (0.20s)
```

```
-----
Time: 1.5 seconds, Memory: 12.00MB
```

```
OK (7 tests, 38 assertions)
```

```
Code Coverage Report:
```

```
2017-03-21 10:14:19
```

```
Summary:
```

```
Classes: 100.00% (1/1)
```

```
Methods: 100.00% (3/3)
```

```
Lines: 54.29% (19/35)
```

```
PlgContentAgpaypal
```

```
Methods: 100.00% ( 3/ 3) Lines: 100.00% ( 18/ 18)
```

```
Remote CodeCoverage reports are not printed to console
```

Klassen und Methoden sind zu 100 % abgedeckt. Die Zeilenabdeckung ist noch nicht optimal! Woran liegt das genau? Die Ausgabe der Konsole hilft bei der Klärung dieser Frage nicht weiter. Ein Report, der genau anzeigt, welche Zeilen abgedeckt sind und in welchen Bereichen noch Nachholbedarf besteht, ist vonnöten. Im nächsten Kapitel zeige ich Ihnen, wie Sie sich einen solchen Report ausgeben lassen können.

Ein schöner Report

Im letzten Kapitel haben wir gesehen, dass die Ausgabe der Konsole beim Absetzen des Befehls `vendor/bin/codecept run unit --coverage` für einen ersten Überblick gut ist. Wenn man detailliertere Angaben benötigt, kommt man mit der Ausgabe der Konsole aber nicht weiter. In diesem Fall sollten Sie den Parameter `--coverage-html` ausprobieren. An der Ausgabe in der Konsole ändert sich hierdurch nichts. Wenn Sie aber nach dem Absetzen des Befehls einen Blick in das Verzeichnis `joomla/tests/_output/` werfen, werden Sie dort das Verzeichnis `coverage` vorfinden. Aber zunächst müssen Sie den Befehl `vendor/bin/codecept run unit --coverage-html` eingeben:

```
/var/www/html/joomla$ vendor/bin/codecept run unit -- coverage-html
```

```
Codeception PHP Testing Framework v2.2.9
```

```
Powered by PHPUnit 5.7.15 by Sebastian Bergmann and contributors.
```

```

Unit Tests (7) -----
...
Summary:
Classes: 100.00% (1/1)
Methods: 100.00% (3/3)
Lines: 54.29% (19/35)
PlgContentAgpaypal
Methods: 100.00% ( 3/ 3) Lines: 100.00% ( 18/ 18)
Remote CodeCoverage reports are not printed to console
HTML report generated in coverage

```

Und nun können Sie sich das Verzeichnis `joomla/tests/_output/` ansehen. Oder Sie öffnen gleich in Ihrem Internetbrowser die Adresse
`file:///var/www/html/joomla/tests/_output/coverage/index.html`.

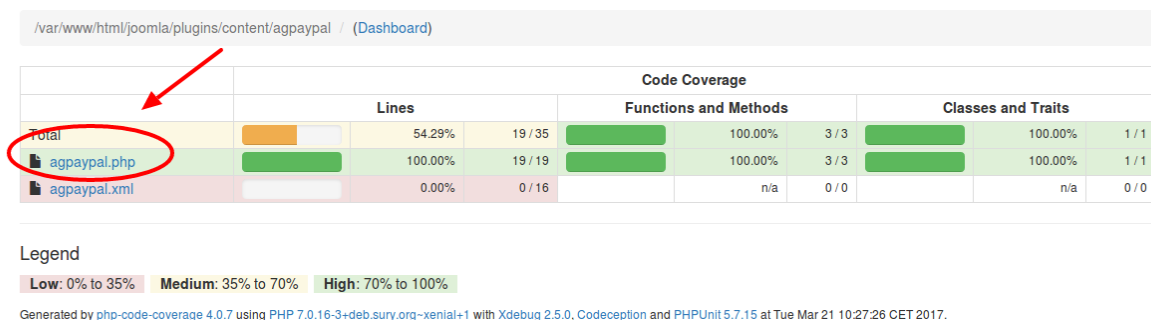


Abbildung 58: Der HTML Report zur Testabdeckung nach dem Ausführen des Befehls `vendor/bin/codecept run unit --coverage-html. 957a.png`

Nun ist alles klar. Wir hatten das Verzeichnis `plugins/content/agpaypal/` komplett zur Messung in der Konfigurationsdatei angegeben. Die Datei `agpaypal.xml` gehört hier aber eigentlich gar nicht dazu. Als XML-Datei können wir sie nicht in PHPUnittests ausführen. Die Zeilen in dieser Datei schlagen aber trotzdem negativ zu Buche.

Bevor wir diese XML-Datei nun aus der Messung der Testabdeckung ausschließen sehen wir uns die Datei `agpaypal.php` etwas genauer an. Klicken Sie dazu einfach auf den als Hyperlink hervorgehobenen Namen der Datei im Report.

/var/www/html/joomla/plugins/content/agpaypal / agpaypal.php

	Code Coverage									
	Classes and Traits			Functions and Methods				Lines		
Total	<div></div>	100.00%	1 / 1	<div></div>	100.00%	3 / 3	CRAP	<div></div>	100.00%	19 / 19
PlgContentAgpaypal	<div></div>	100.00%	1 / 1	<div></div>	100.00%	3 / 3	7	<div></div>	100.00%	18 / 18
onContentPrepare				<div></div>	100.00%	1 / 1	2	<div></div>	100.00%	4 / 4
startCreateButtons				<div></div>	100.00%	1 / 1	2	<div></div>	100.00%	4 / 4
createButtons				<div></div>	100.00%	1 / 1	3	<div></div>	100.00%	10 / 10

```

1 <?php
2
3 defined('_JEXEC') or die;
4
5 class PlgContentAgpaypal extends JPlugin
6 {
7
8     public function onContentPrepare($context, &$row, $params, $page = 0)
9     {
10         if (is_object($row))
11         {
12             $this->startCreateButtons($row->text);
13         }
14         else
15         {
16             $this->startCreateButtons($row);
17         }
18
19         return true;
20

```

Abbildung 59: Die Detailansicht der Datei agpaypal.php im HTML-Report. 957b.png

In der Detailansicht können Sie sich genau ansehen, welche Zeilen beim Testdurchlauf ausgeführt wurden und welche nicht. In unserem Falle ist alles mit grüner Farbe hinterlegt. Die Zeilen sind mit einem Test abgedeckt. Zeilen, die während keines Tests aufgerufen wurden, wären mit roter Farbe hinterlegt.

Weiter geht es. Wir wollten unsere Konfiguration so abändern, dass die XML-Datei nicht mehr in die Messung der Testabdeckung eingeschlossen wird. Eigentlich wollen wir nur PHP-Dateien testen. Deshalb ändern wir die Konfiguration so ab, dass nur noch Dateien mit der Endung PHP in die Messung der Testabdeckung aufgenommen werden. Im nachfolgenden Beispiel sehen Sie den relevanten Teil der Konfigurationsdatei /var/www/html/joomla/codeception.yml. Ich habe die Zeile -plugins/content/agpaypal/* in -plugins/content/agpaypal/*.php abgeändert.

```

coverage:
  enabled: true
  include:
    - plugins/content/agpaypal/*.php

```

Probieren Sie es aus. Nach der Änderung der Konfiguration ist bei einem erneuten Testdurchlauf jede zu testende Zeile grün und die Testabdeckung beträgt 100 %.

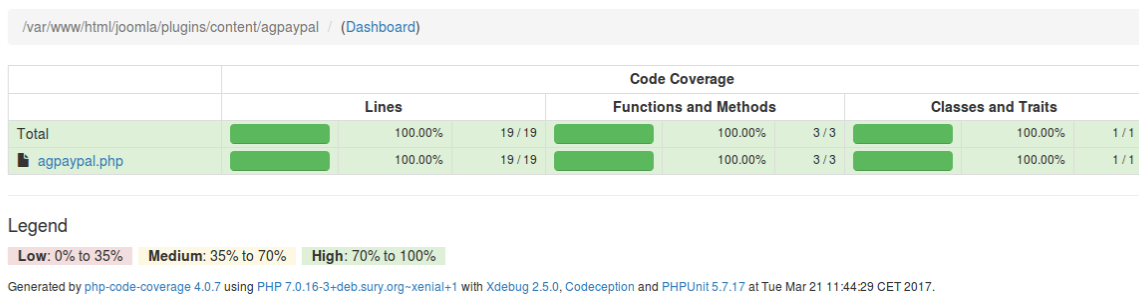


Abbildung 60: Die Testabdeckung ist 100 %. 957c

Sie wissen nun, wie Sie die Testabdeckung von Unittest messen und analysieren können. Um die Testabdeckung für Funktionstests und Akzeptanztests auszuwerten, benötigen wir ein zusätzliches Werkzeug. Dieses zusätzliche Werkzeug heißt [c3](#).

Testabdeckung von Akzeptanztests und Funktionstests

In den beiden letzten Kapiteln haben Sie gesehen, dass Funktionstests und Akzeptanztests mithilfe eines anderen Moduls ausgeführt werden. Unsere Funktionstests verwendeten das Modul PHPBrowser und unsere Akzeptanztests verwendeten WebDriver. Mithilfe der Codeception-Erweiterung [c3](#) können auch dann Daten zur Testabdeckung gesammelt werden, wenn die Anwendung über ein anderes Modul ausgeführt wird. Die Daten werden während des Testablaufs in einem temporären Verzeichnis gespeichert und von Codeception weiterverarbeitet, wenn der Test beendet ist. Zunächst müssen wir aber c3 installieren. Dies können wir mithilfe von Composer. Fügen dazu die Zeile "codeception/c3": "2.*" in die Datei composer.json ein.

```
{
  "require": {
    "codeception/codeception": "*",
    "codeception/specify": "*",
    "codeception/verify": "*",
    "codeception/c3": "2.*"
  }
}
```

Wenn Sie die Zeile "codeception/c3": "2.*" eingefügt haben, führen Sie den Befehl `composer update` aus.

```
/var/www/html/joomla$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing codeception/c3 (2.0.10)
  Downloading: 100%
[codeception/c3] Copying c3.php to the root of your project...
[codeception/c3] Include c3.php into index.php in order to collect codecoverage from server scripts
- Removing guzzlehttp/psr7 (1.4.1)
- Installing guzzlehttp/psr7 (1.4.2)
...
Writing lock file
Generating autoload files
```

Alle notwendigen Pakete werden nun heruntergeladen. Am Ende befindet sich eine Datei mit dem Namen `c3.php` im Stammverzeichnis Ihres Projekts. Wenn Sie meinem Beispiel gefolgt sind, heißt dies konkret, dass Sie die Datei `c3.php` im Verzeichnis `/var/www/html/joomla/` vorfinden.

Um eine Verbindung zwischen der Codeception-Erweiterung `c3` und dem `PhpBrowser` oder `WebDriver` Modul herzustellen, müssen Sie in der Konfiguration von Codeception den Parameter `c3_url` definieren. Fügen Sie dazu am Ende der Konfigurationsdatei `/var/www/html/joomla/codeception.yml` die Zeile `c3_url: 'http://localhost/joomla/index.php'` ein, denn in die Datei `joomla/index.php` werden wir im nächsten Schritt die Datei `joomla/c3.php` einbinden.

```
actor: Tester
paths:
  tests: tests
  log: tests/_output
  data: tests/_data
  support: tests/_support
  envs: tests/_envs
settings:
  bootstrap: _bootstrap.php
  colors: true
  memory_limit: 1024M
extensions:
```

```
enabled:
    - Codeception\Extension\RunFailed
modules:
    config:
        Db:
            dsn: "
            user: "
            password: "
            dump: tests/_data/dump.sql
coverage:
    enabled: true
    include:
        include:
            - plugins/content/agpaypal/*.php
    c3_url: 'http://localhost/joomla/index.php'
```

Als Nächstes fügen Sie am Anfang der Datei `joomla/index.php` den Befehl zum Inkludieren der Datei `c3.php` ein.

RANDBEMERKUNG:

Die Datei `joomla/index.php` gehört zum Kern von Joomla! und wird bei einer Aktualisierung des Content Management Systems eventuell überschrieben. Wenn Sie Joomla! anpassen möchten, sollten Sie eigentlich nur Änderungen in eigenen Erweiterungen vornehmen. Da es sich um eine Testinstallation handelt, mache ich hier aber eine Ausnahme.

```
<?php
if (file_exists('c3.php')) {
    include_once 'c3.php';
}
/**
 * @package Joomla.Site
 *
 * @copyright Copyright (C) 2005 - 2016 Open Source Matters, Inc. All rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
```

*/

...

Testabdeckung bei Akzeptanztests

Als Nächstes führen wir unsere Akzeptanztests aus und überprüfen, ob eine Testabdeckung aufgezeichnet wird. Die Testabdeckung muss 0 % betragen, denn der bisher erstellte Akzeptanztest nutzt unser Plugin nicht. Mit diesem Akzeptanztest erstellen wir einen Beitrag im Backend. Unser Plugin wird nur ausgeführt, wenn im Frontend ein PayPal Jetzt-kaufen-Button angezeigt wird. Bisher hatten wir folgende Tests in der Feature-Datei `/var/www/html/joomla/tests/acceptance/paypal.feature`.

Feature: paypal

In order to manage content articles with paypal links

I need to create a content article with a paypal link

Background:

When I login into Joomla administrator

And I see the administrator dashboard

Scenario: Create an Article

Given There is a add content link

When I create new content with field title as "Beitrag mit PayPal-Button" and content as a "Dies ist mein erster Beitrag."

And I save an article

Then I should see the article "Beitrag mit PayPal-Button" is created

Zur Erinnerung: Vor dem Start der Akzeptanztests müssen Sie den Selenium Standalone Server, wie in Kapitel *Akzeptanztests | Selenium WebDriver – eine Einführung | Selenium und ChromeDriver installieren* beschrieben, starten – es sei denn, dieser ist aktiv. Mit dem Befehl `vendor/bin/codecept run acceptance --coverage-html` starten Sie dann den Testlauf.

```
vendor/bin/codecept run acceptance --coverage-html
```

```
Codeception PHP Testing Framework v2.2.9
```

```
Powered by PHPUnit 5.7.17 by Sebastian Bergmann and contributors.
```

```
Acceptance Tests (1) -----
```

```
✓ paypal: Create an Article (6.82s)
```

```
-----  
Time: 14.88 seconds, Memory: 12.00MB  
OK (1 test, 2 assertions)  
Code Coverage Report:  
2017-04-03 10:47:45  
Summary:  
Classes: 0.00% (0/1)  
Methods: 0.00% (0/3)  
Lines: 0.00% (0/40)  
Remote CodeCoverage reports are not printed to console  
HTML report generated in coverage
```

Auf den ersten Blick sieht alles gut aus. Klassen, Methoden und Zeilen zeigen eine Testabdeckung von 0 % an und wenn Sie einen Blick in das Verzeichnis `tests/_output/coverage` werfen, finden Sie Daten.

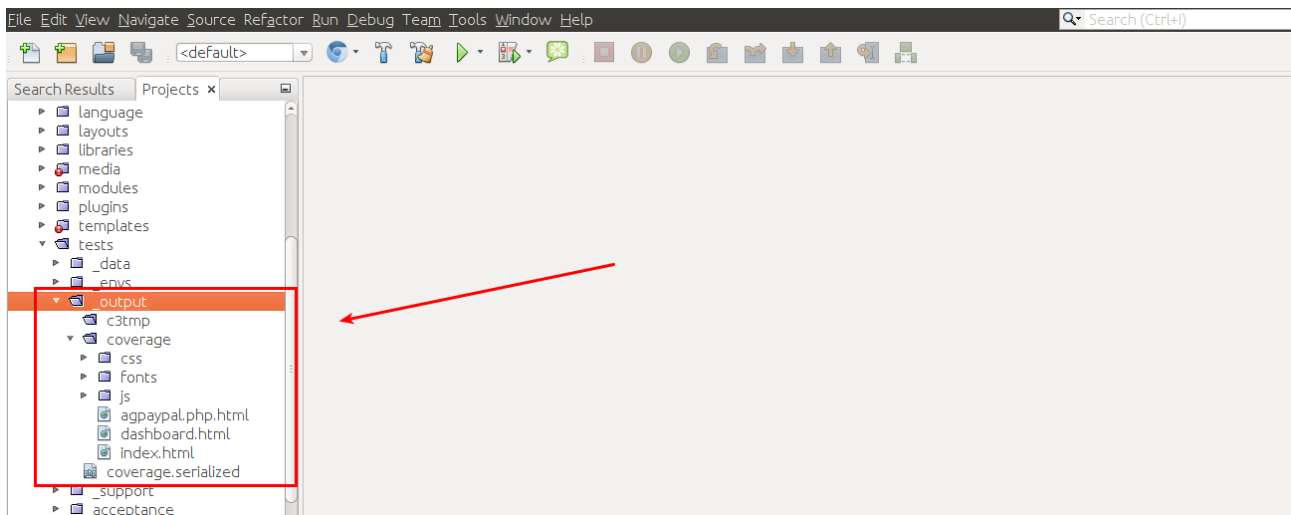


Abbildung 61: 950.png

Den HTML Report im Verzeichnis `tests/_output/coverage` können Sie sich, genauso wie eben bei den Unittests beschrieben, im Browser ansehen. Navigieren Sie dazu in Ihrem Browser zur Adresse `file:///var/www/html/joomla/tests/_output/coverage/index.html`.

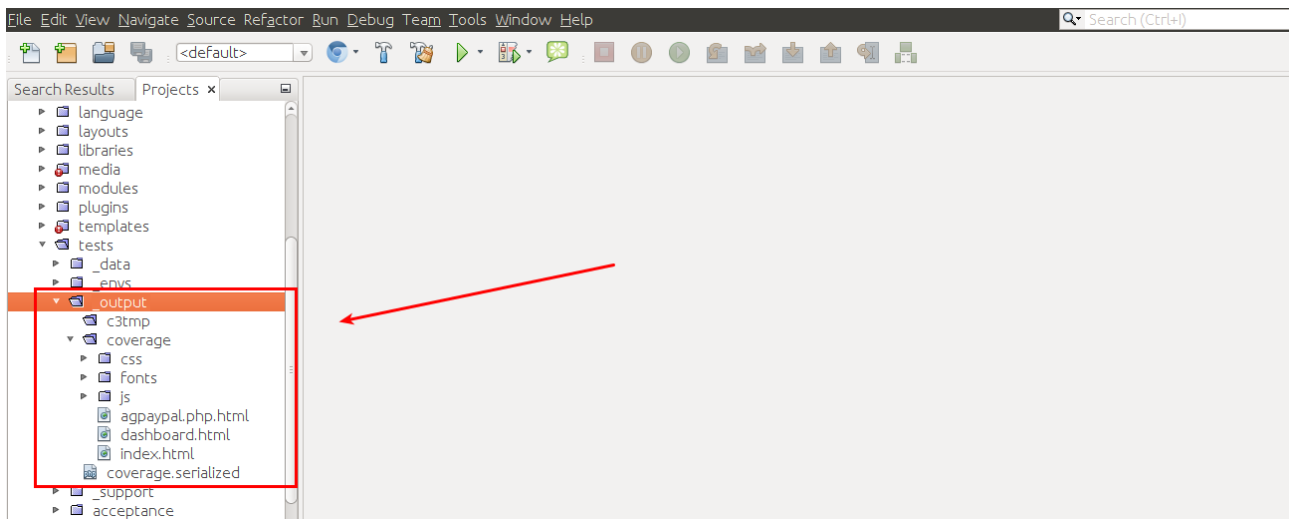


Abbildung 62: Den HTML-Report finden Sie im Verzeichnis tests/_output/coverage.950a.png

Auch hier ist alles in roter Farbe dargestellt – es wurde also kein Programmcode in der Datei plugins/content/agpaypal/agpaypal.php mit einem Test ausgeführt. Eine Testabdeckung von 0 % wird also richtig ausgegeben. Uns interessiert aber viel mehr, ob auch eine positive Testabdeckung richtig berechnet wird. Dafür erweitern wir die Feature-Datei um einen Step. In dem Step öffnen wir nur die Startseite unserer Joomla! Installation im Browser. Hier wird der PayPal Jetzt-kaufen-Button angezeigt und somit wird Programmcode in unserem Plugin – zumindest teilweise – ausgeführt. Bitte fügen Sie den neuen Step in Ihre Feature-Datei ein. Sie sehen den dafür notwendigen Text im nachfolgenden Programmcode am Ende fett hervorgehoben.

Feature: paypal

In order to manage content articles with paypal links

I need to create a content article with a paypal link

Background:

When I login into Joomla administrator

And I see the administrator dashboard

Scenario: Create an Article

Given There is a add content link

When I create new content with field title as "Beitrag mit PayPal-Button" and content as a "Dies ist mein erster Beitrag."

And I save an article

Then I should see the article "Beitrag mit PayPal-Button" is created

Scenario: Show an Article

Given I am on home page

Die Testmethode, die diesen Step ausführen, müssen wir dann natürlich auch noch in der CEST-Datei implementieren. Fügen Sie also die Methode `iAmOnHomePage()` in die Datei `/joomla/tests/_support/Step/Acceptance/Paypal.php` ein. Falls Sie unsicher sind, können Sie die einzelnen Schritte hierzu in den Kapiteln *Akzeptanztests | Gherkin* und *Akzeptanztests | Implementatierung der Testmethoden* nachlesen.

```
<?php
namespace Step\Acceptance;
class Paypal extends \AcceptanceTester{
    ...
    /**
     * @Given I am on home page
     */
    public function iAmOnHomePage(){
        $I = $this;
        $I->amOnPage("/index.php");
    }
}
```

Nun passt auch alles. Zumindest sind die Prozentwerte zur Testabdeckung nun teilweise mit positiven Werten gefüllt.

```
/var/www/html/joomla$ vendor/bin/codecept run acceptance --coverage-html
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.17 by Sebastian Bergmann and contributors.
Acceptance Tests (2) -----
✓ paypal: Create an Article (6.28s)
✓ paypal: Show an Article (2.17s)
-----
Time: 18.83 seconds, Memory: 12.00MB
OK (2 tests, 3 assertions)
Code Coverage Report:
2017-04-03 11:29:07
Summary:
Classes: 0.00% (0/1)
Methods: 33.33% (1/3)
```

Lines: 88.89% (16/18)
PlgContentAgpaypal
Methods: 33.33% (1/ 3) Lines: 88.24% (15/ 17)
Remote CodeCoverage reports are not printed to console
HTML report generated in coverage

Im HTML Report können Sie sich die Informationen detaillierter anzeigen lassen. Wenn Sie die Adresse `file:///var/www/html/joomla/tests/_output/coverage/index.html` öffnen und hier auf den Link `agpaypal.php` klicken, sehen Sie ganz genau, welche Zeilen bei einem Testdurchlauf nicht ausgeführt werden: Diese Zeilen haben eine rote Hintergrundfarbe.

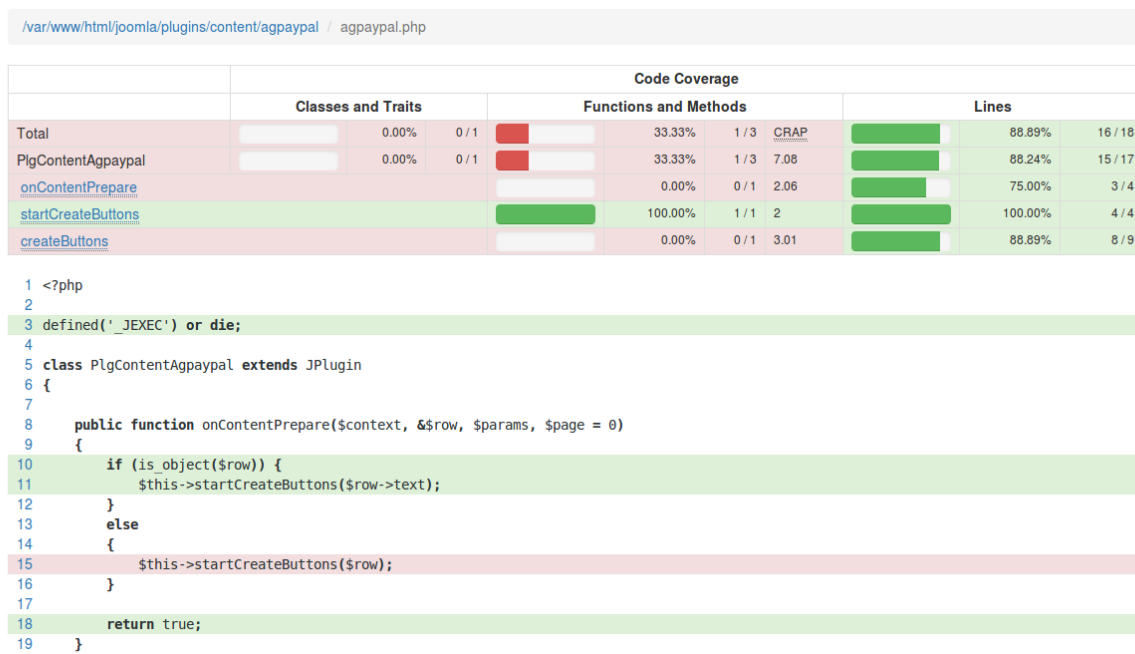


Abbildung 63: Die Zeilen, die bei einem Testdurchlauf nicht ausgeführt wurden, haben eine rote Hintergrundfarbe. 950b.png

Testabdeckung bei Funktionstests

Die Testabdeckung Ihrer Funktionstests prüfen Sie genauso, wie Sie es bei den Akzeptanztests gemacht haben. Mit dem Befehl `vendor/bin/codecept run functional --coverage-html` starten Sie den Testlauf.

Programmcodestandards

Mit dem Begriff Programmcodestandard bezeichnet man in der Programmierung einen [Programmierstil](#) der die Erstellung von Quellcode regelt. Programmcodestandards legen fest, wie der Quellcode eines Programms gestaltet sein soll. Warum gibt es

Programmcodestandards? Wenn mehrere Menschen zusammen ein Programm erstellen, helfen allgemeine Regeln die Wartbarkeit und Verständlichkeit des Programms positiv zu beeinflussen.

Auch wenn viele Softwareentwickler die Vorteile von Programmcodestandards kennen, verwenden Sie diese Regeln nicht gerne. Genau wie das Erstellen von Tests sieht man den Nutzen dieser Regeln erst auf den zweiten Blick. Auf den ersten Blick ist die Arbeit und die Zeit die man in das Ändern des Programmcodes investiert, einfach nur lästig. Dabei kann man gerade in diesem Bereich viele Dinge automatisieren.

RANDBEMERKUNG:

In vielen Softwareprojekten gibt es spezielle Programmcodestandards. Die Standards des Joomla! Projekts finden Sie auf [Github](#).

Ich zeige Ihnen im folgenden wie Sie [PHP_CodeSniffer](#) des Projektes <http://www.squizlabs.com/> mithilfe von Composer installieren. PHP_CodeSniffer beinhaltet zwei Werkzeuge. phpcs findet Verstöße gegen definierte Programmcodestandards und phpcbf korrigiert die meisten Standardverletzungen automatisch.

Wenn Sie sich dieses nun praktisch ansehen möchten, müssen Sie PHP_CodeSniffer installieren. Fügen Sie dazu als Nächstes die Zeile "squizlabs/php_codesniffer": "2.*" in die Datei composer.json ein.

```
{
  "require": {
    "codeception/codeception": "*",
    "codeception/specify": "*",
    "codeception/verify": "*",
    "codeception/c3": "2.*",
    "squizlabs/php_codesniffer": "2.*"
  }
}
```

Mit dem Befehl `composer update` fügt Composer dann das Paket `squizlabs/php_codesniffer` zu Ihrem Projekt hinzu und macht es bekannt.

```
/var/www/html/joomla$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing squizlabs/php_codesniffer (2.8.1)
  Downloading: 100%
...
Writing lock file
Generating autoload files
```

Das war es auch schon. Sie können nun sofort mit der Analyse Ihrer Dateien beginnen. Setzen Sie doch gleich einmal den Befehl `./vendor/bin/phpcs ./plugins/content/agpaypal/agpaypal.php` auf Ihrer Kommandozeile ab.

```
/var/www/html/joomla$ ./vendor/bin/phpcs ./plugins/content/agpaypal/agpaypal.php
FILE: /var/www/html/joomla/plugins/content/agpaypal/agpaypal.php
-----
FOUND 103 ERRORS AND 2 WARNINGS AFFECTING 46 LINES
-----
 2 | ERROR   | [ ] Missing file doc comment
 5 | ERROR   | [ ] Expected "if (...) {\n"; found "if (...) \n {\n"
...
58 | ERROR   | [x] Spaces must be used to indent lines; tabs are not |   | allowed
58 | ERROR   | [x] Line indented incorrectly; expected 4 spaces, |   | found 1
-----
PHPCBF CAN FIX THE 90 MARKED SNIFF VIOLATIONS AUTOMATICALLY
-----
Time: 52ms; Memory: 6Mb
```

Bei mir wurden 103 Fehler und 2 Warnungen in 46 Zeilen Programmcode gefunden. PHP_CodeSniffer verwendet standardmäßig den PEAR-Standard. Beim Codieren unseres Beispiel-Plugins hatte ich zugegebenermaßen auch nicht auf Standards geachtet. Mit 103 Fehlern hatte ich allerdings nicht gerechnet und wenn ich diese alle von Hand korrigieren müsste, würde ich mich sehr schwer tun. Deshalb bin ich froh, dass ich das Skript `phpcbf` kenne.

RANDBEMERKUNG:

Eine kurze Dokumentation zu PHP_CodeSniffer können Sie auf [Github](#) einsehen. Über den Befehl `./vendor/bin/phpcs -h` können Sie sich einen Überblick über alle möglichen Optionen verschaffen.

Wurden bei Ihnen auch Standardverletzungen angezeigt? Dann verwenden Sie am besten, genau wie ich, den Befehl `./vendor/bin/phpcbf ./plugins/content/agpaypal/agpaypal.php`.

```
/var/www/html/joomla$ ./vendor/bin/phpcbf ./plugins/content/agpaypal/agpaypal.php
Changing into directory /var/www/html/joomla/plugins/content/agpaypal
Processing agpaypal.php [PHP => 401 tokens in 59 lines]... DONE in 25ms (90 fixable violations)
=> Fixing file: 0/90 violations remaining [made 3 passes]... DONE in 61ms
Patched 1 file
Time: 123ms; Memory: 6Mb
```

So schnell kann das gehen. Eine erneute Fehlersuche zeigt, dass nun nur noch neun Fehler vorhanden sind. Alles andere hat phpcbf automatisch korrigiert. Die neun restlichen Fehler sind Fehler, die nicht automatisch bereinigt werden können.

```
/var/www/html/joomla$ ./vendor/bin/phpcs ./plugins/content/agpaypal/agpaypal.php
FILE: /var/www/html/joomla/plugins/content/agpaypal/agpaypal.php
-----
FOUND 9 ERRORS AND 3 WARNINGS AFFECTING 11 LINES
-----
 2 | ERROR   | Missing file doc comment
 9 | ERROR   | Missing class doc comment
12 | ERROR   | Missing function doc comment
17 | ERROR   | Expected "} else {\n"; found "}\n     else\n
   |         | {\n"
25 | ERROR   | Missing function doc comment
27 | WARNING | Line exceeds 85 characters; contains 86 characters

35 | ERROR   | Private method name
   |         | "PlgContentAgpaypal::createButtons" must be prefixed
   |         | with an underscore
```

```
35 | ERROR | Missing function doc comment
37 | ERROR | Expected "for (...) {\n"; found "for (...) \n
    |      | {\n"
43 | ERROR | Expected "} else {\n"; found "} \n      else \n
    |      | {\n"
47 | WARNING | Line exceeds 85 characters; contains 110 characters
49 | WARNING | Line exceeds 85 characters; contains 119 characters
-----
Time: 46ms; Memory: 4Mb
```

Ich hoffe, dass ich Ihnen in diesem kurzen Kapitel die Verwendung von Programmcodestandards etwas schmackhafter gemacht habe.

Kurzgefasst

Man kann fast alles noch besser machen. In diesem Kapitel haben Sie erfahren, wie Sie Ihre Tests analysieren und eventuell verbessern können. Sie wissen nun, wie Sie die Testabdeckung für Ihre Software messen und als Report ausgeben können.

Außerdem haben Sie ein Werkzeug, das Sie bei der Verwendung eines Programmierstils unterstützt, kennengelernt. Und der CRAP-Index gibt Ihnen eine Antwort auf die Frage, ob Sie die Struktur Ihres Programmcodes verbessern sollten.

Automatisierung – Gestatten mein Name ist Jenkins

Bei einer kontinuierlichen Integration werden einzelne Bestandteile der Software permanent integriert. Die Software wird in kleinen Zyklen immer wieder erstellt und getestet. Sie haben sich sehr viel Arbeit mit dem Erstellen von Tests gemacht und möchten nicht, dass diese in Vergessenheit geraten. Die Software Jenkins unterstützt Sie bei der Automatisierung Ihrer Tests und auch bei der kontinuierlichen Integration Ihrer Softwarebausteine – das Programm ist einfach einzurichten und zu handhaben. Außerdem können Sie Jenkins flexibel erweitern und an Ihre Bedürfnisse anpassen. Hierzu gibt es jede Menge verschiedener Plugins.

Warum sollten Sie sich Jenkins – oder ein anderes Werkzeug, dass sie bei der Integration Ihrer Software unterstützt – ansehen? Integrationsprobleme oder fehlerhafte Tests finden Sie frühzeitig und nicht erst Tage oder Wochen später. Bei einer kontinuierlichen Integration ist die Fehlerbehebung wesentlich einfacher, weil die

Fehler zeitnah zur Programmierung auftauchen und so in der Regel nur ein kleiner Programmteil betroffen ist.

Installation

Ich habe Jenkins anhand der [Installationsanleitung](#), die auf der Jenkins-Website veröffentlicht ist, installiert.

Das Jenkins Projekt betreibt eigene Repositories. Dies macht die Installation relativ einfach. Wir müssen dazu vor der Paketinstallation den Schlüssel über den Befehl `wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -` hinzufügen.

```
~$ wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -  
[sudo] Passwort für meinName:  
OK
```

Nach dem Hinzufügen des Schlüssels sollte Ihnen der Status OK angezeigt werden. Zum Hinzufügen der Paketquellliste geben Sie nun den Befehl `sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'` ein.

```
~$ sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
```

Als Nächstes lesen Sie mit `sudo apt-get update` die, in der Datei `/etc/apt/sources.list` und die in der Datei `/etc/apt/sources.list.d/` eingetragenen, Paketquellen neu ein. Diesen Befehl sollten Sie nach dem Hinzufügen einer neuen Quelle immer ausführen, um die aktuellsten Informationen zu den verfügbaren Paketen zu erhalten.

```
meinName@acer:~$ sudo apt-get update  
...  
Es wurden 271 kB in 2 s geholt (101 kB/s).  
Paketlisten werden gelesen... Fertig
```

Sobald Ihr System aktualisiert ist, führen Sie den Befehl `sudo apt-get install jenkins` für die Installation von Jenkins aus. Während der Installation müssen Sie an einer Stelle mit der Eingabe des Buchstaben J bestätigen, dass Sie die Installation fortsetzen möchten.


```
~$ sudo apt-get install jenkins
Paketlisten werden gelesen... Fertig
Abhängigkeitsbaum wird aufgebaut.
Statusinformationen werden eingelesen.... Fertig
Die folgenden zusätzlichen Pakete werden Installiert
daemon
Die folgenden NEUEN Pakete werden installiert:
daemon jenkins
0 aktualisiert, 2 neu installiert, 0 zu entfernen und 316 nicht aktualisiert.
Es müssen 69,6 MB an Archiven heruntergeladen werden.
Nach dieser Operation werden 70,2 MB Plattenplatz zusätzlich benutzt.
Möchten Sie fortfahren? [J/n] J
Holen:1 http://de.archive.ubuntu.com/ubuntu xenial/universe amd64 daemon amd64 0.6.4-1 [98,2 kB]
Holen:2 http://pkg.jenkins.io/debian-stable binary/ jenkins 2.32.3 [69,5 MB]
Es wurden 69,6 MB in 57 s geholt (1.210 kB/s).
Vormals nicht ausgewähltes Paket daemon wird gewählt.
(Lese Datenbank ... 241914 Dateien und Verzeichnisse sind derzeit installiert.)
Vorbereitung zum Entpacken von .../daemon_0.6.4-1_amd64.deb ...
Entpacken von daemon (0.6.4-1) ...
Vormals nicht ausgewähltes Paket jenkins wird gewählt.
Vorbereitung zum Entpacken von .../jenkins_2.32.3_all.deb ...
Entpacken von jenkins (2.32.3) ...
Trigger für man-db (2.7.5-1) werden verarbeitet ...
Trigger für systemd (229-4ubuntu10) werden verarbeitet ...
Trigger für ureadahead (0.100.0-19) werden verarbeitet ...
daemon (0.6.4-1) wird eingerichtet ...
jenkins (2.32.3) wird eingerichtet ...
Trigger für systemd (229-4ubuntu10) werden verarbeitet ...
Trigger für ureadahead (0.100.0-19) werden verarbeitet ...
```

Sie haben nun Jenkins installiert. Am Ende des Installationsvorgangs wurde der Jenkins Dienst gestartet. Den Status des Jenkins Dienstes können Sie sich mithilfe des Befehls `service jenkins status` anzeigen lassen. Bei der Installation wurde der Benutzer Jenkins angelegt, unter dessen Namen der das Programm Jenkins ausgeführt wird.

```
service jenkins status
● jenkins.service - LSB: Start Jenkins at boot time
```

```
Loaded: loaded (/etc/init.d/jenkins; bad; vendor preset: enabled)
Active: active (exited) since Mo 2017-04-03 20:58:46 CEST; 12h ago
Docs: man:systemd-sysv-generator(8)
Tasks: 0
Memory: 0B
CPU: 0
Apr 03 20:58:45 acer systemd[1]: Starting LSB: Start Jenkins at boot time...
Apr 03 20:58:45 acer jenkins[2242]: * Starting Jenkins Continuous Integration Server jenkins
Apr 03 20:58:45 acer su[2291]: Successful su for jenkins by root
Apr 03 20:58:45 acer su[2291]: + ??? root:jenkins
Apr 03 20:58:45 acer su[2291]: pam_unix(su:session): session opened for user jen
Apr 03 20:58:46 acer jenkins[2242]: ...done.
Apr 03 20:58:46 acer systemd[1]: Started LSB: Start Jenkins at boot time.
...
```

Die Konfigurationsdatei finden Sie im Verzeichnis `/etc/default/`. Mit dem Befehl `cat /etc/default/jenkins` können Sie einen Blick in diese Datei werfen.

```
/etc/default$ cat /etc/default/jenkins
# defaults for jenkins continuous integration server
# pulled in from the init script; makes things easier.
NAME=jenkins
# location of java
JAVA=/usr/bin/java
...
```

Jenkins startet nach der Installation automatisch auf dem Port 8080. Öffnen Sie nun die Adresse `http://localhost:8080/` in Ihrem Webbrowser, damit wir uns im nächsten Kapitel die Weboberfläche von Jenkins genauer ansehen können. Die meisten Einstellungen können Sie hier konfigurieren. Bei der allerersten Anmeldung müssen Sie einen Benutzer anlegen. Jenkins erklärt Ihnen alle hierfür notwendigen Schritte.

Konfiguration von Jenkins für die Verwendung mit Codeception

Öffnen Sie in Ihrem Internetbrowser die Adresse `http://localhost:8080/`. Melden Sie sich dann mit Ihrem Benutzer an. Klicken Sie als Nächstes links auf den Menüpunkt New Item.

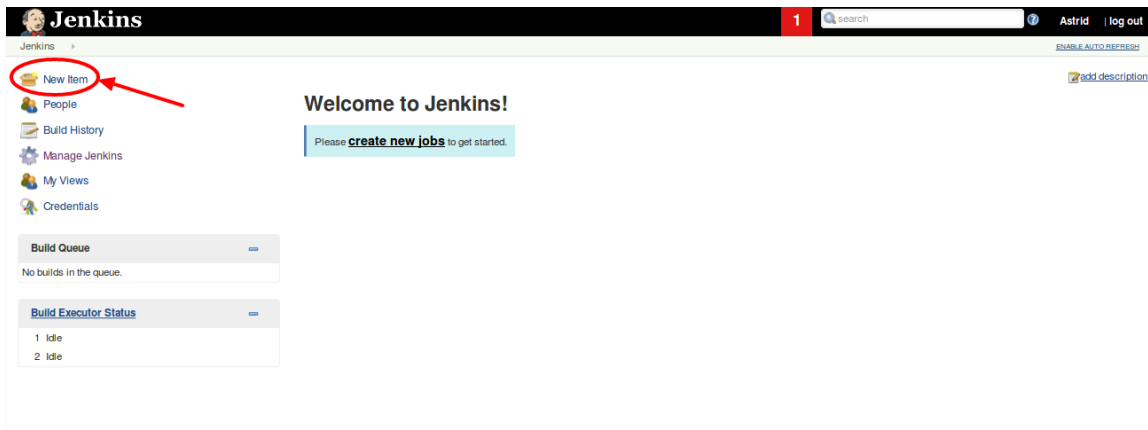


Abbildung 64: Die Abbildung zeigt die Oberfläche des webbasiertes Software-Systems Jenkins nach der ersten Anmeldung. 955b.png

Geben Sie dem Item einen Namen, ich habe den Namen Agpaypal gewählt. Wählen Sie dann den Eintrag *Freestyle Projekt* und klicken danach ganz unten auf die Schaltfläche OK.

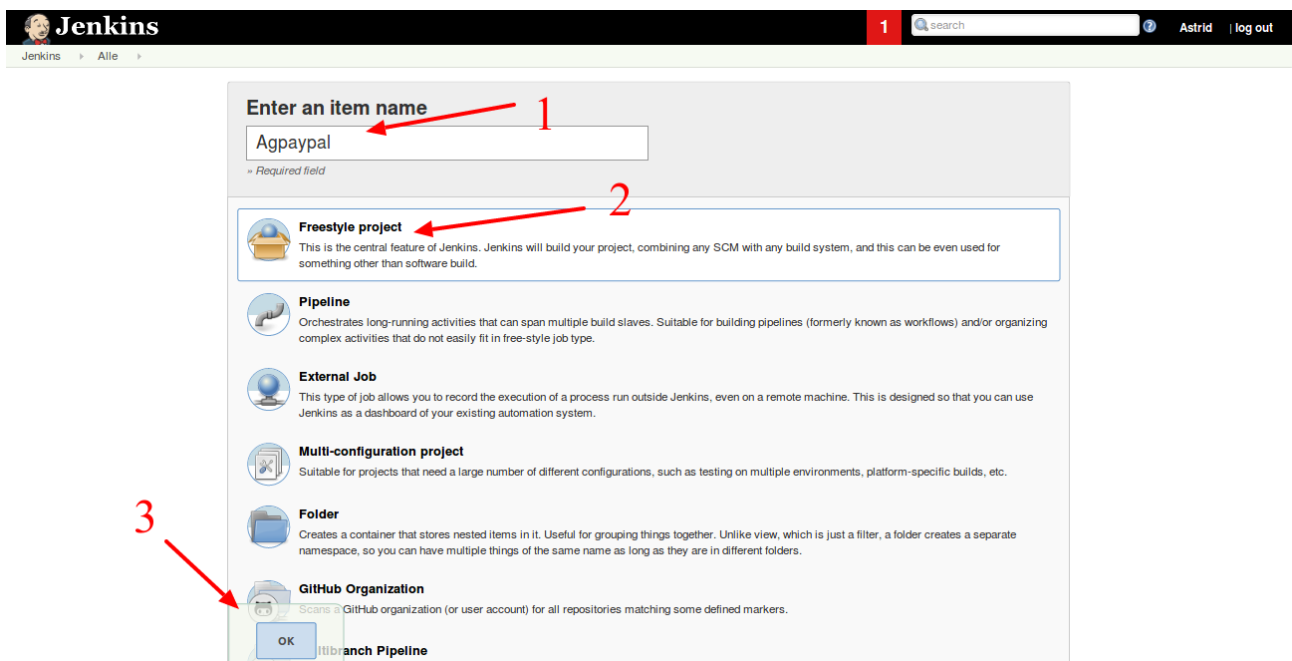


Abbildung 65: Die Abbildung zeigt die Weboberfläche zum Anlegen eines neuen Items in Jenkins. 955g

Und hier werden Sie nun von Einstellmöglichkeiten fast erschlagen. Wir wollen, dass unser Unittests in regelmäßigen Abständen automatisch ausgeführt werden. Deshalb wechsele ich sofort in das Register Build, wähle im Auswahlfeld Add Build Step den

Eintrag **Execut Shell** und trage im sich nun öffnenden Textbereich die Kommandos `cd /var/www/html/joomla/` und `vendor/bin/codecept run unit` ein. Sodann speichere ich diese Einträge.

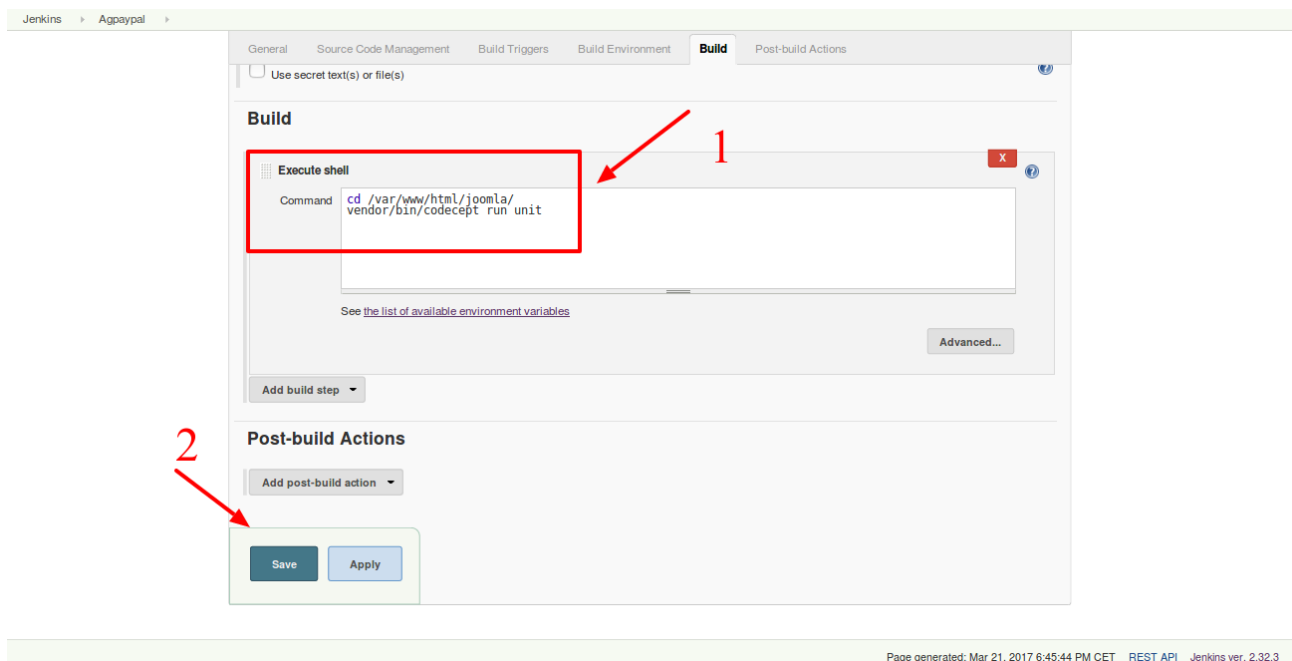


Abbildung 66: Die Konfiguration eines Items in Jenkins – Register Build. 955c.png

Um die automatische Ausführung zu konfigurieren, wechsele ich nun in das Register *Source Code Management*. Hier kann ich im Bereich *Build Triggers* eine regelmäßige Ausführung einstellen. Dazu muss ich die Option *Build periodically* auswählen, um im Textfeld, das mit *Schedule* beschrieben ist, die Zeitpunkte eingeben zu können, zu denen meine Kommandos automatisch ausgeführt werden sollen. Hierzu muss ich fünf Werte eingeben.

1. Minute (Mögliche Werte: 0–59)
2. Stunde (Mögliche Werte: 0–23)
3. Tag im Monat (Mögliche Werte: 1-31)
4. Monat (Mögliche Werte: 1-12)
5. Tag der Woche (Mögliche Werte: 0-7)

Für jeden Wert kann anstelle einer Zahl auch der Buchstabe **H** oder das Zeichen ***** eingesetzt werden. **H** steht für einen beliebigen möglichen Wert und ***** steht für jeden möglichen Wert.

Ich möchte, dass meine Tests täglich ausgeführt werden. Die genaue Uhrzeit ist mir egal. Deshalb gebe ich `H H * * *` in das Testfeld, das mit *Schedule* beschrieben ist, ein. Am Anfang ist diese Art einen Zeitpunkt zu beschreiben etwas gewöhnungsbedürftig. Vielleicht schreibt Jenkins deshalb unterhalb des Feldes das Datum, zu dem das Kommando das nächste Mal ausgeführt wird. Hier können Sie also Ihre Eingabe noch einmal überprüfen.

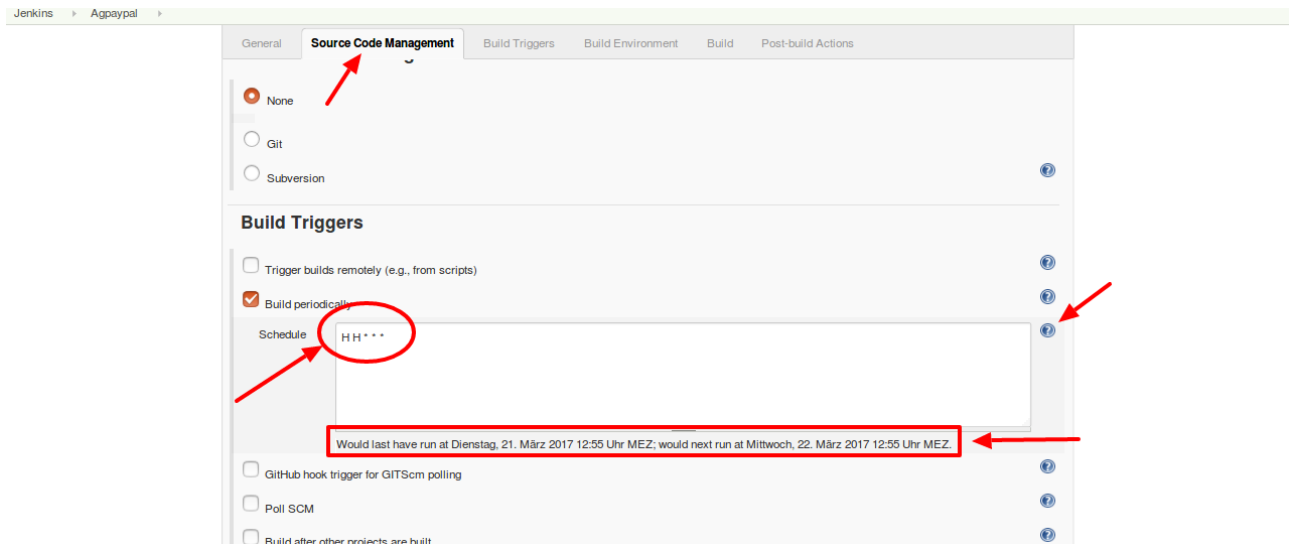


Abbildung 67: Die Konfiguration eines Items in Jenkins – Register Source Code Management. 955e.png

Wenn Sie Ihr Item nun speichern, werden Sie zu einer Übersichtsseite weitergeleitet. Sie möchten nun sicherlich nicht einen Tag warten, um zusehen, ob Ihre Kommandos auch ausgeführt werden. Starten Sie deshalb nun einen manuellen Lauf. Sie können dies jederzeit tun, indem Sie links im Menü auf den Eintrag *Build now* klicken.

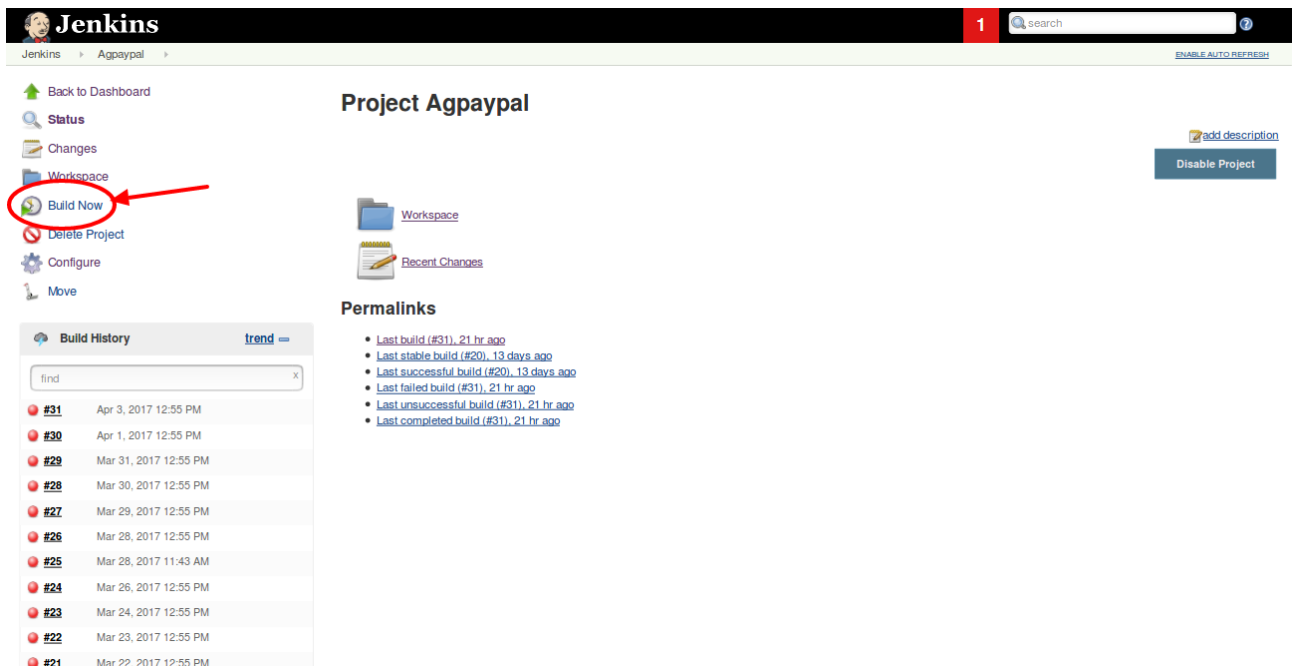


Abbildung 68: Übersichtsseite zum Item Agpaypal in Jenkins. 949.png

Unmittelbar nach dem Klick auf Build Now sehen Sie im Bereich Build History einen weiteren Eintrag. Bei mir ist dieser grün. Das heißt, alles ist fehlerfrei gelaufen. Im Fehlerfall sehen Sie einen roten Punkt.

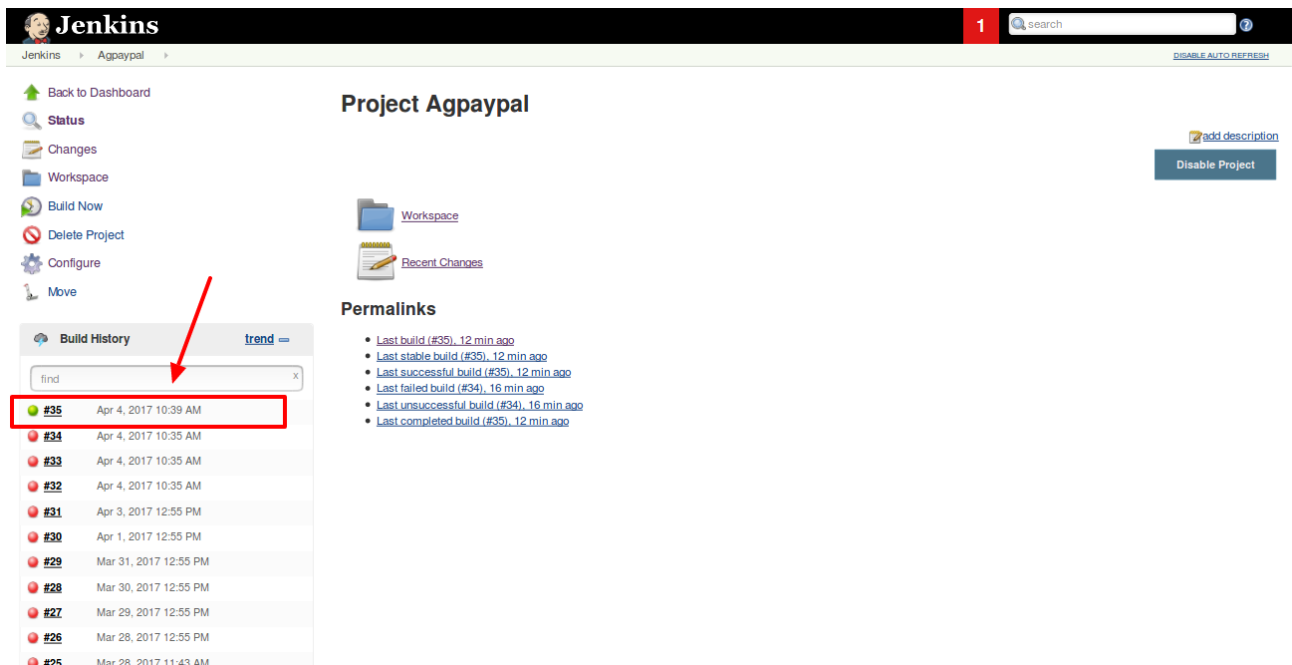


Abbildung 69: In der Build History der Übersichtsseite sehen Sie ob der Build fehlerfrei gelaufen ist. 949a.png

Wenn alles fehlerfrei gelaufen ist, müssen Sie gar nichts weiter tun. Wenn Fehler aufgetreten sind, möchten Sie diese sicher beheben. Sie können sich die Ausgabe der

Konsolen zu einem Build ansehen, indem Sie zunächst auf den entsprechenden Eintrag im Bereich *Build History* klicken und danach links auf den Eintrag *Console Output*.

The screenshot shows the Jenkins web interface. On the left sidebar, the 'Console Output' link is highlighted with a red circle and a red arrow. The main content area displays the 'Console Output' for a build. The output text includes: 'Gestartet durch Benutzer anonymous', 'Baue in Arbeitsbereich /var/lib/jenkins/workspace/Agpaypal', '[Agpaypal] \$ /bin/sh -xe /tmp/hudson4825283706863069123.sh', '+ cd /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.', a table of 7 PHPUnit tests, 'Time: 256 ms, Memory: 10.00MB', '[30;42mOK (7 tests, 74 assertions)](0m', '[htmlpublisher] Archiving HTML reports...', and 'Finished: SUCCESS'. At the bottom right, it says 'Page generated: Apr 4, 2017 10:59:36 AM CEST', 'REST API', and 'Jenkins ver. 2.32.3'.

Abbildung 70: Die Ausgabe der Konsole bei einem erfolgreichen Jenkins Build. 949b.png

Bei mir sind die ersten Versuche fehlgeschlagen, weil Jenkins keinen Zugriff auf das Verzeichnis `tests/_output/` hatte. Ich die Rechte für dieses Verzeichnis dann so gesetzt, dass jeder schreiben und lesen darf: `chmod 666 -R tests/_output/`.

The screenshot shows the Jenkins web interface. On the left sidebar, the 'Console Output' link is highlighted with a red circle and a red arrow. The main content area displays the 'Console Output' for a build. The output text includes: 'Gestartet durch Benutzer Astrid', 'Baue in Arbeitsbereich /var/lib/jenkins/workspace/Agpaypal', '[Agpaypal] \$ /bin/sh -xe /tmp/hudson3307182246216858583.sh', '+ cd /var/www/html/joomla/', '+ vendor/bin/codecept run unit --xml', 'Codeception PHP Testing Framework v2.2.9', 'Powered by PHPUnit 5.7.17 by Sebastian Bergmann and contributors.', a red-bordered box containing the error message: '[Codeception\Exception\ConfigurationException] Path for output is not writable. Please, set appropriate access mode for output path.', a list of command-line options for PHPUnit, 'Build step \'Shell ausführen\' marked build as failure', and 'Finished: FAILURE'. At the bottom right, it says 'Astrid', 'log out', and 'Jenkins ver. 2.32.3'.

Abbildung 71: Die Ausgabe der Konsole bei einem fehlerhaften Jenkins Build. 955a.png

Kurzgefasst

Nun haben Sie Jenkins installiert und können sich eine Meinung zu dem Werkzeug bilden. Nur darum ging es mir in diesem Kapitel. Vielleicht arbeiten Sie schon mit der freien Software zur verteilten Versionsverwaltung [Git](#). Lesen Sie dann in der Dokumentation von [Codeception](#) wie Sie mit Jenkins automatisch ein Repository von Git klonen, Composer Abhängigkeiten installieren und Codeception Tests durchführen können.

Auf Wiedersehen

Ich hoffe, dass Ihnen der Rundgang durch Codeception gefallen hat und Sie jede Menge für sich mitnehmen konnten.

Literatur

Eike Riedemann: Testmethoden für sequentielle und nebenläufige Software-Systeme
Teubner, Stuttgart, 1997

Andreas Zeller: Why Programs Fail A GuideTo Systematic Debugging dpunkt.verlag,
Heidelberg, 2005

W.E. Howden; [Symbolic Testing and the DISSECT Symbolic Evaluation System](#), 1977

Achim Feyhl; [Management und Controlling von Softwareprojekten: Software wirtschaftlich auswählen, entwickeln, einsetzen und nutzen](#), Ausgabe 2, 2013

James A. Whittaker, Jason Arbon, Jeff Carollo; [How Google Tests Software](#), 2012

Matthias Daigl, Rolf Glunz; ISO 29119, 2016

Helmut Balzert: Lehrbuch der Software-Technik. Spektrum Verlag 2008, ISBN 978-3-8274-1161-7.

Andreas Spillner, Tilo Linz: Basiswissen Softwaretest. dpunkt.Verlag 2005, ISBN 3-89864-358-1.

Harry M. Sneed, Mario Winter: Testen objektorientierter Software. Hanser Verlag 2002, ISBN 3-446-21820-3.

Ernest Wallmüller: Software Qualitätssicherung in der Praxis. Hanser Verlag 2001, ISBN 3-446-21367-8.