

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 gelöst wurden.

Ich habe sehr viel Zeit mit dem Testen von Software - und früher noch mehr mit 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 kosten. Und ich gehe sogar noch weiter: 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 Lernweg gerne zur Verfügung gehabt hätte.

Randbemerkung: Test-Methoden sind nachhaltig

Testmethoden 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.

Die Techniken des Testens sind im Gegensatz zu Programmiertechniken, die Modeerscheinungen sind, zeitlos.

Welche Themen behandelt dieses Buch?

Das Kapitel Softwaretests - eine Einstellungssache? behandelte die Frage, warum man Zeit in Softwaretests investieren sollte. Dabei erläutere ich auch die wichtigsten Testkonzepte.

Praxisteil: Die Testumgebung einrichten Todo

Codeception – ein Überblick

Unit Tests

Testduplikate

Funktionstest

Acceptancetests

Analyse

Was Sie zur Bearbeitung dieses Buchs benötigen

Welche Ausstattung brauchen Sie? Sie müssen nicht sehr viele Voraussetzungen erfüllen, um dieses Buch zu bearbeiten. Natürlich müssen Sie über einen heute üblichen Computer verfügen. Auf diesem sollte eine Entwicklungsumgebung und ein lokaler Webserver installiert sein. Hilfen zur Einrichtung finden Sie im Kapitel Praxisteil: Die Testumgebung einrichten.

Was sollten Sie persönlich für Kenntnisse mitbringen? Sie sollten grundlegende PHP-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 im Browser aufrufen. Auf einem lokal installierten Webserver geht dies meist über eine URL in der Form `http://localhost/datei.php`.

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

Wer sollte dieses Buch lesen

Jeder, der der Meinung ist, dass Softwaretests reine Zeitverschwendung sind, sollte einen Blick in dieses Buch werfen. Insbesondere möchte ich die Entwickler einladen das Buch zu lesen, die wie ich, schon immer Tests für ihre Software schreiben wollten – es dann aber doch aus den unterschiedlichsten Gründen nie zu Ende gebracht haben. **Codeception** könnte ein Weg sein, Hindernisse aus dem Weg zu räumen.

Infos zu Formatierungen Buch

Ein Buch im Bereich Programmierung enthält oft **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
    );

```

Kommandozeileingaben 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. Zum Verständnis der Inhalte dieses Buches sind diese Texte nicht notwendig. Ich habe Randbemerkungen eingerückt und mit einem Strich am linken Rand versehen.

Exkurs - Was bedeutet das Zeichen & vor dem Parameter \$response

Mithilfe des vorangestellten &-Zeichens können Sie eine Variablen an eine Methode per Referenz übergeben, so dass die Methode ihre Argumente modifizieren kann. Beispiel:

```

function add (&$num)
{
    $num++;
}
$number = 0;
add($number);
echo $number;

```

Wichtige Merksätze habe ich zum leichteren Wiederfinden grau hinterlegt.

WICHTIG!

final, private und static Methoden können nicht mit PHPUnit Stub Objekten genutzt werden. PHPUnit unterstützt diese Methoden 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 das erste Kapitel zunächst links liegen lassen und mit dem praktischen zweiten Kapitel beginnen. Ich verweise an passender Stelle immer mal wieder auf diesen ersten Theorieteil.

Ich habe bewusst Frameworks, hier konkret Joomla! und Codeception, als Beispiele für Erklärungen gewählt. Die Verwendung von Frameworks anstelle von kleinen selbst erstellen Codebeispielen hat Vorteile und Nachteile. Ein Framework stellt einen Rahmen zur Verfügung, innerhalb dessen der Programmierer eine Anwendung erstellt. Aus diesem Rahmen kann ich Testbeispiele wählen. Ich muss also nicht immer das Rad selbst neu erfinden. Nachteilig ist, dass dieser Rahmen teilweise selbst erklärungsbedürftig ist.

Ich hoffe, dass Ihnen nach der Lektüre dieses Kapitels klar ist, warum Softwaretests in einem Projekt eingeplant werden sollten.

Ihnen wird bewusst werden, welchen Einfluss Softwaretests auf Ihre Arbeit haben. Mir ging es so,

- dass ich sicherer in meiner Arbeit wurde,
- Fehler in Spezifikationen eher gefunden und korrigiert habe,
- meine Vorgehensweise selbst im Vorhinein überdacht habe und

- so qualitativ bessere und fehlerfreie Programme erstellt habe.

Um dies mit praktischen Beispielen zu veranschaulichen tauchen wir am Ende dieses Abschnitts in die Techniken Testgetriebene Programmierung (Test-Driven-Development, kurz TDD) und verhaltensgetriebene Softwareentwicklung (Behavior-Driven-Development (BDD) ein.

Dieses Kapitel umfasst die Themen

- Warum sollten Sie Software testen?
- Projektmanagement
- Testen ist wirklich wichtig!

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. Das der ideale Test nicht berechenbar ist hat [Howden](#) schon 1977 bewiesen.

Dann habe ich aber das Gegenteil erfahren. Außerdem hat mich der Satz im [Google Testing Blog](#)

„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 getestet werden kann, ist es ebenso offensichtlich, dass es unmöglich ist, ohne zu testen etwas qualitativ Gutes zu entwickeln. Es gibt sogar Entwickler, die noch weiter gehen und sagen: „Softwareentwicklung ohne Tests ist wie Klettern ohne Seil und Haken“.

EXKURS: Warum ist Software fehlerhaft?

Ursachen fehlerhafter Software sind menschliche Fehlleistungen beim Austauschen, Verarbeiten oder Speichern von Informationen, die zur Software-

Entwicklung benötigt werden.

Austausch von Informationen Hauptformen der Kommunikation und mögliche Fehlerursachen sind

-Intrapersonale Kommunikation – Informationsverarbeitung innerhalb einer Person Irrtum beim Denken und Wahrnehmen: Informationsverarbeitung innerhalb des Menschen. Identitätsirrtum: Den Wert einer Variablen brutto statt netto zuweisen.

-Interpersonale Kommunikation - Informationsaustausch zwischen Gesprächspartnern Erklärungsirrtum: Hat hat A gesagt, aber B gemeint.

-Irrtum bei der Übermittlung: Mitarbeiter gibt entgegengenommen Anruf falsch wieder

-Irrtum beim Entschlüsseln: Information falsch gelesen oder gehört.

Unterschiedliche Sprache / Fachsprache

Inhaltsirrtum: Missverständnis über qualitative oder quantitative Eigenschaften

Medien-gebundene Kommunikation – Informationsaustausch über Medien wie Bücher oder Zeitungen Es treten die gleiche Probleme wie bei interpersonaler Kommunikation auf.

Typische Ausprägungen bei der Softwareentwicklung: große Projekte mit Projektleitern und Programmierern Online-Medien (Wikis, Foren, Online-Dokumentationen) Kognitive Einschränkungen

Wir Menschen haben zu wenig RAM im Gehirn! Unser Langzeitgedächtnis kann zwar viele Informationen speichern. Im Kurzzeitgedächtnis ist aber nur wenig Speicherplatz. Oft reicht dieser nicht dafür 2 Schleifendurchläufe inklusive Kontext nachzuvollziehen! Genau wie ein Maurer das Haus Stein für Stein baut und schreiben wir Programme Zeile für Zeile. Wir betrachten und manipulieren Programme durch ein extrem kleines kognitives Fenster!

Nicht-kommunikative Fehlerquellen

Komplexität der Systeme „Denken wie ein Computer“ Verstehen von Nebenläufigkeiten mangelndes Problemverständnis algorithmische Lösung unzutreffend / unbekannt

fehlende Information der Beteiligten

Stress, Übermüdung, fehlende Motivation

Probieren Sie es 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 Tests Ihnen eine mühsame Fehlersuche erspart hat. 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? Dieses Kapitel hat Sie sicher davon überzeugt, dass dies ohne Tests nicht möglich ist.

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

Projektmanagement

EXKURS: Testen im Software-Lebenszyklus

Analyse

Korrektheit, Vollständigkeit, Konsistenz

Testfälle vorbereiten

Entwurf

Konsistenz, Vollständigkeit Systemstruktur

fehlende Fälle, Schnittstellen, fehlerhafte Logik

Testfälle für interne Funktionen angeben

Implementierung

Code überprüfen und ausführen

Wartung = Beginn klassischen Debugging

Fehler beseitigen ohne neue zu erzeugen

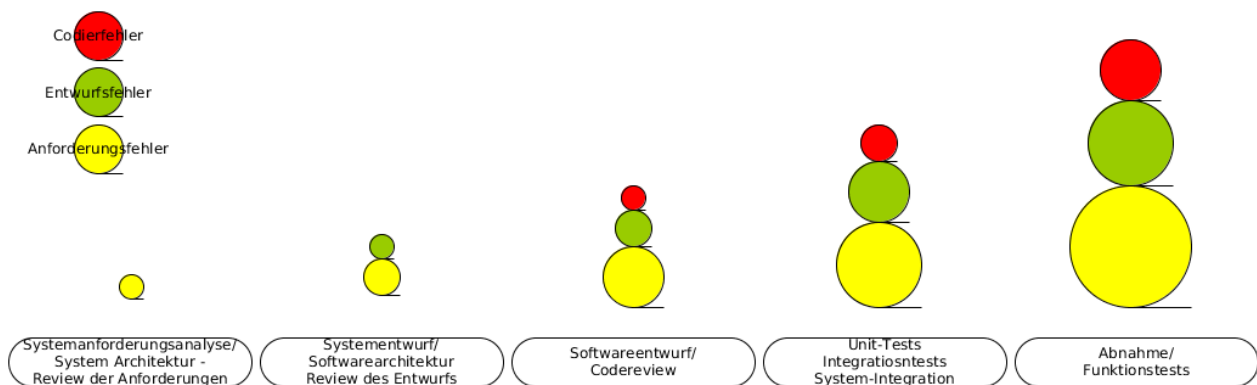


Abbildung 1: Relative Kosten für die Fehlerbehebung 997.png

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 schon in anderen Bereichen von diesem Spannungsverhältnis gehört. Es bei fast allen betrieblichen Abläufen in einem Unternehmen ein wichtiges Thema.

ToDo Bild machen von Spannungsverhältnis

Zum Beispiel werden Überstunden geleistet, um einen Termin einzuhalten; dies erhöht die Kosten. Oder bei geforderte Kosteneinsparungen werden Leistungen gestrichen, um die Kosten zu halten; dies senkt die Qualität des Ergebnisses. Ein letztes Beispiel: Um die Qualität einer Software sicherzustellen, wird zusätzliche Zeit in Tests investiert und der Termin nach hinten verschoben.

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

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

Exkurs - Technische Schulden

Der Begriff Technische Schuld steht für die möglichen Konsequenzen schlechter technischer Umsetzung von Software. Unter der technischen Schuld versteht man den zusätzlichen Aufwand, den man für Änderungen und Erweiterungen an schlecht geschriebener Software im Vergleich zu gut geschriebener Software 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 technischer Schuld.

	bewusst	ungewollt
umsichtig	So sollte es sein :)	Nun haben wir etwas gelernt.
risikofreudig	Wir haben keine Zeit!	Was ist OOP?

(Todo Wie bekomme ich Tabelle in Exkurs?)

Und nun sind wir bei einem Thema angekommen, dass sehr unterschiedlich diskutiert wird. Wie schaffen wir es Kosten in der Planung genau zu berechnen und im zweiten Schritt Kosten und Nutzen in idealer Weise zu verbinden? (Todo Formulierung)

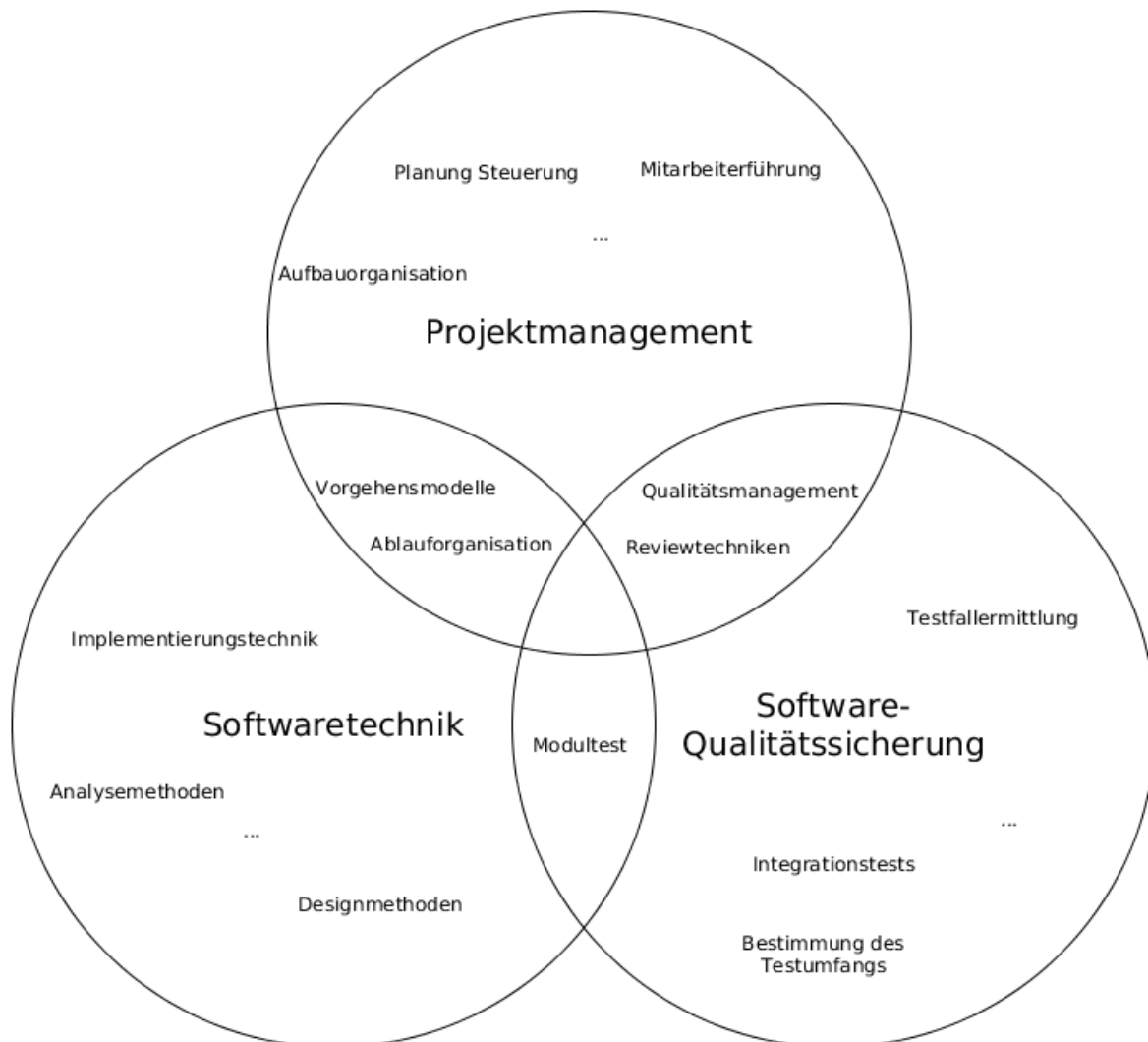


Abbildung 2: Softwareprojekt Todo Bild noch in Zusammenhang bringen 995.png

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 deren Kostenabweichung ermittelt: 70% der Projekte wiesen eine Kostenabweichung von mindestens 50% gegenüber der ursprünglichen Planung auf!

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

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

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

(Todo erklären wie ich meine wie das geht – Gründe warum Softwareprojekte schlecht planbar sind (wissen von Entwicklern, immer wieder neue Techniken) – wie kann Wissen über Kosten verbessert und damit planbarer machen -)

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, die beim Auffinden eines Fehlers für dessen Behebung notwendig sind, an.

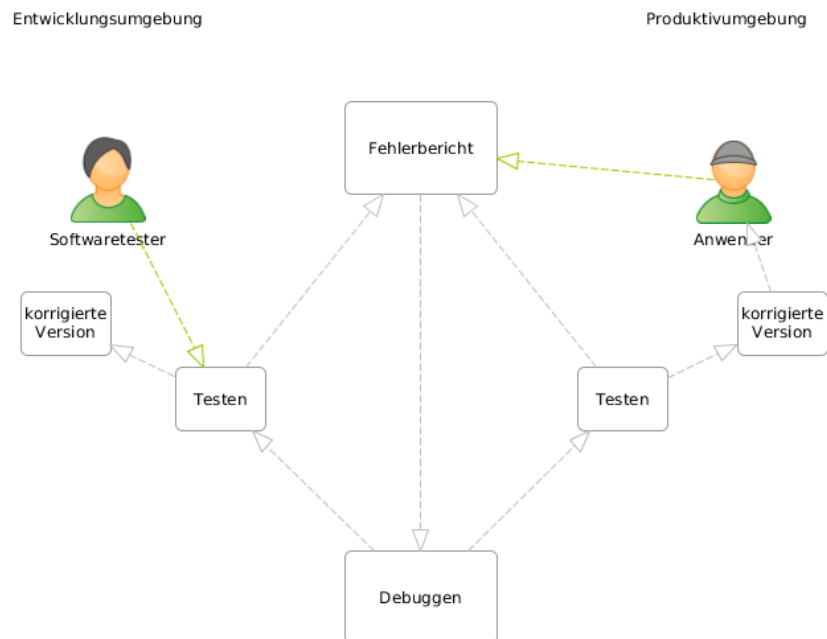
Exkurs: Testen und Debuggen

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 haben gemein, dass Sie Fehlfunktionen aufdecken. Es gibt aber auch Unterschiede in der Bedeutung.

Testmethoden finden unbekannte Fehlfunktionen während der Entwicklung. Dabei ist das Finden der Fehlfunktion aufwendig und teuer, die Lokalisation und Behebung des Fehlers ist hingegen billig. Das Erkennen der Fehlfunktion ist quasi ein Nebenprodukt, dass sich aus den Testfällen ergeben hat.

Debugger beheben bekannte Fehlfunktion nach Fertigstellung des Produktes.

Dabei ist das Finden der Fehlfunktion gratis, die Lokalisation und Behebung des Fehlers aber teuer. Hauptaufwand: Reproduktion und Lokalisierung.



Kosten beim Auffinden eines Fehlers in den Unterschiedlichen Projektphasen

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

(Todo einfügen von dem was ich weiter hintenschon geschrieben habe)

Kontinuierliches Testes

(Todo einfügen von dem was ich schon geschrieben habe)

Stellen Sie sich folgendes Szenario vor. Die neue Version eines beliebten Content Management Systems soll veröffentlicht werden. Alles das, was die Entwickler des Teams seit dem letzten Release 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 der Software!

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, bevor 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 immer wieder erstellt und getestet. Integrationsprobleme oder fehlerhafte Tests finden Sie frühzeitig und nicht erst Tage oder Wochen später. Bei einer kontinuierlichen Integration ist die Fehlerbehebung wesentlich leichter, weil der Fehler zeitnah zur Programmierung erkannt auftaucht und in der Regel nur ein kleiner Programmteil betroffen ist.

(Todo Jenkins Kapitel 9?)

Damit Sie bei der kontinuierlichen Integration auch jederzeit Tests zur Verfügung haben, sollten Sie testgetrieben entwickeln.

Testgetriebene Entwicklung (TDD)

Die Testgetriebene Entwicklung ist eine Technik, bei der in kleinen Schritten entwickelt wird. Dabei schreiben Sie als erste den Testcode. Erst danach erstellen Sie den zu testenden Programmcode. Jede Änderung am Programm wird erst vorgenommen, nachdem der Test für diese Änderung erstellt wurde. Tests müssen also unmittelbar nach der Erstellung fehlschlagen. Die geforderte Funktion ist ja noch nicht im Programm implementiert. Nun erst erstellen Sie den Programmcode, der den Test erfüllt. Die Tests helfen Ihnen also zusätzlich dabei, das **Programm richtige** 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 wertvoll.

(Todo Link Theorie ist immer schwer vorstellbar. Praktisches Beispiel Kapitel)

EXKURS Regressionstest

Ein **Regressionstest** ist ein wiederholt durchgeführter Test. Durch die Wiederholung wird sicher gestellt, dass Modifikationen in bereits getesteten Teilen der Software keinen neuen Fehler oder keine neue Regression verursachen.

Warum sollten Sie Regressionstests machen? Sie sollten Tests wiederholt durchführen weil bestimmte Fehlfunktionen manchmal plötzlich wieder auftauchen. Zum Beispiel

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

Behavior-Driven-Development (BDD)

BDD ist eine Art Best Practices für das Schreiben von Tests. BDD kommt idealerweise gemeinsam mit TDD und Unit Tests zum Einsatz kommen. Im Prinzip steht Behavior-Driven-Development dafür nicht die Implementierung, sondern das Verhalten des Codes zu testen. (Todo Formulierung)

Akzeptanztests prüfen ob die Spezifikation, also die Anforderung des Kunden, erfüllt ist. Akzeptanztests helfen Ihnen dabei, das **richtige Programm** zu schreiben.

(Todo BDD mehr erklären – gilt nicht nur für Akzeptanztest ? Vielleicht Verweis?, welche Werkzeuge gibt es gherkin)

EXKURS: Grundlegende Teststrategien

Grundsätzlich können Sie **spezifikationsorientierte** und **implementationsorientierte** 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 Programm-Internen ohne System oder systematisch durchgeführt.

Vorteile spezifikationsorientierten Tests sind

- fehlende Programmteile werden entdeckt
- Integritätsbedingungen können aufgestellt werden

- Portierungen werden unterstützt

Nachteile spezifikationsorientierter Tests sind

- Probleme bei nicht repräsentativer Spezifikation
- Zusätzliche Fälle die nicht in der Spezifikation berücksichtigt sind werden nicht entdeckt / getestet
- hilft nicht beim Aufdecken von Nebenläufigkeiten
- hilft nicht bei Inkompatibilität zur Konfiguration

Implementationsorientierte Tests gewinnen Testfälle durch strukturelle Analyse des Programms. Diese Testvariante kennen Sie vielleicht unter dem Namen **White Box-Tests**.

Vorteile implementationsorientierter Tests sind

- bis zu einem gewissen Grade automatisierbar
- geht auch bei fehlender Spezifikation
- es wird bemerkt ob im Programm etwas zuviel ist
- Verfeinerungen und Implementationsdetails können gezielt getestet werden

Nachteile implementationsorientierter Tests sind

- Übertragung auf nebenläufige Systeme nicht trivial, aber machbar
- nicht implementierte Teile werden nicht entdeckt.

	implementationsbezogen	spezifikationsbezogen
statisch	Statische Code-Analyse	Entwurfsphase
dynamisch	Profiler	Unser Bereich

Einordnung von Tests in die Softwaretechnologie

Konstruktive Methoden (Erstellung)

- Anforderungsdefinition, Entwurf, Programmierung

Analytische Methoden (Messen, Bewerten)

- Qualitätssicherung, Testen

Wichtig: Konstruieren Sie nicht erst und testen im Anschluss daran. Führen Sie begleitend zu allen Ebenen des Softwarezyklus Tests durch.

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. In der [ISO 29119-2](#) umfasst er neun umfangreiche Aktivitäten. (Todo Aktivitäten auflisten?)

Es gibt aber auch andere Vorgehensweisen. Whittaker beschreibt in seinem Buch [How Google Tests Software](#) die sehr an der Praxis orientierte 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.

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.

EXKURS Testfälle

Was sind Testfälle und wie erstellen Sie systematisch? (Todo Beispiel auf meine papal Beispiel abändern)

Nehmen wir die Menge der möglichen Testfälle ist D.

Um ein Beispiel zu nenne: Bei der Anmeldung an einem System kann ein Benutzer einen korrekten Benutzernamen in Kombination mit einem korrekten Passwort eingeben. Ein weiterer Testfall wäre die Eingabe eines ungültigen Benutzer mit einem Passwort.

D = Eingabebereich

Nehmen wir weiter an, dass es eine Menge an möglichen Ausgabemöglichkeiten gibt und nennen diese R. Bei der Anmeldung an einem System könnte eine Ausgabemöglichkeit die Bestätigung der korrekten Eingabe der Anmeldedaten sein. Eine andere Ausgabe könnte den Benutzer darauf hinweisen, dass seine Daten nicht korrekt sind und er es erneut versuchen soll.

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.

P: D -> R

Ein Spezialfall von P liegt vor, wenn das Programm die Daten so verarbeitet, wie es in der Spezifikation festgelegt wurde. Beispiel: Der Benutzer erhält eine Bestätigung, wenn seine Anmeldedaten korrekt sind. Bei fehlerhafter Eingabe

muss er darauf hingewiesen werden, dass etwas nicht stimmt. Im folgenden Nenne diesen Spezialfall F.

$F: D \rightarrow R$ (Ausgabe-Anforderung für P ist erfüllt)

Das Programm arbeitet also formal gesehen korrekt und fehlerfrei, wenn die Menge P gleich der Menge F für alle mögliche Eingaben ist.

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

Eine endliche Teilmenge $T \subseteq D$ ist eine Menge von Testfällen.

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 ideale 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.

- Wenn die Programmausführung fehlerhaft ist gilt: $\exists d \in D: P(d) \neq F(d)$ - Für den idealen Test gilt $T = \{d\}$

- 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 das so wäre gäbe es sicherlich ausschließlich korrekte Programme. Der ideale Test ist nicht berechenbar! Todo Verweis auf Howden (Symbolic Testing and the DISSECT Symbolic Evaluation System)

Der gleichförmig ideale Test: Ein idealer Test gilt für ein konkretes Programm mit einer konkreten Fehlfunktionen. Ein gleichförmig idealer Test findet Fehlfunktionen in allen Programmen P die die Ausgabe-Anforderung F berechnen. Nun ist es aber so, dass es in der Menge aller möglichen Testfälle D ein Programms P gibt, das für einen Testfall d nicht die korrekte Ausgabe berechnet. $\forall d \in D$ gibt es P, das nur für d falsch ist.

Dies bedeutet, dass nur der erschöpfender Test gleichförmig ideal ist. Das heißt, dass ein Test bei dem die Menge der Testfälle gleich der möglichen Eingaben ist, also $T = D$ gilt, ein erschöpfend idealer Test ist.

Das Testen aller möglichen Eingabeparameter ist in der Realität unmöglich. Ein systematisches stichprobenartiges Testen ist die einzig praktikable Lösung!

(Todo Beispiel umschrieben)

Passwort korrekt	Passwort leer	Benutzer korrekt	Benutzer leer	
1	1	1	1	Passwort leer und korrekt schließt sich aus
0	1	1	1	Benutzer leer und korrekt nicht möglich
1	0	1	1	Benutzer leer und korrekt nicht möglich
0	0	1	1	Benutzer leer und korrekt nicht möglich
1	1	0	1	Passwort leer und korrekt nicht möglich
0	1	0	1	JAuthentication:: STATUS_FAILURE Message:: JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED
1	0	0	1	JAuthentication:: STATUS_FAILURE Message:: JGLOBAL_AUTH_NO_USER
0	0	0	1	JAuthentication:: STATUS_FAILURE Message:: JGLOBAL_AUTH_NO_USER
1	1	1	0	Passwort leer und korrekt nicht möglich
0	1	1	0	JAuthentication:: STATUS_FAILURE Message:: JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED
1	0	1	0	JAuthentication:: STATUS_SUCCESS
0	0	1	0	JAuthentication:: STATUS_FAILURE

				Message:: JGLOBAL_AUTH_INVALID_PASS
1	1	0	0	Passwort leer und korrekt nicht möglich
0	1	0	0	JAAuthentication:: STATUS_FAILURE Message:: JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED
1	0	0	0	JAAuthentication:: STATUS_FAILURE Message:: JGLOBAL_AUTH_No_USER
0	0	0	0	JAAuthentication:: STATUS_FAILURE Message:: JGLOBAL_AUTH_NO_USER

Tests generieren

Sie haben nun sehr viel Theoretisches zum Thema Softwaretests gelesen. In diesem Buch werde ich ihnen einige Werkzeuge erklären, die Sie in der Praxis unterstützen. Also beim generieren von Tests. Wenn Sie diese Tools einsetzen, werden Sie selbst erfahren, wie Sie Ihre Tests am besten schreiben. Da jeder Entwickler 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 sollen **wiederholbar** sein. (kein Faker)
2. Ein Test sollten **einfach** gehalten sein. (nur eine Einheit)
3. Ein Test sollten **unabhängig** von anderen Tests sein.

Testen ist wirklich wichtig!

Todo

Wie starten Sie

Bottom up (erst unit tests) top down (erst integrationstest)

Zusammenfassung

Das 1. Kapitel beschreibt, was Softwaretest sind und warum Software Tests wichtig sind. Dann plane ich verschiedene Konzepte zu erklären. Unter anderem die Begriffe Test-Driven-Development und Behavior-Driven-Development. Ich möchte auch darauf eingehen, wie Tests kontinuierlich integriert werden können/sollen.

Außerdem möchte ich hier wichtige Prinzipien wie Einfachheit, Wiederholbarkeit und Unabhängigkeit eines Tests erläutern.

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 Tests 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 eine Paypal Schaltfläche in einen Beitrag zu integrieren. Der Benutzer muss hierzu nur ein bestimmtes Textkürzel kennen und dies 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 Textkürzel stößt wandelt sie dieses in ein HTML Element um, welches eine Paypal Schaltfläche anzeigt.

Entwicklungsumgebung und Arbeitsweise

Entwickeln Sie Software? Dann haben Sie Ihre persönliche 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 auch sicherlich keinen Spaß an Ihrer Arbeit haben.

(Todo hier fehlt noch was)

- 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. Mit Netbeans können Sie beispielsweise Ihre Software [Debuggen](#), Programmcode automatisch vervollständigen oder Werkzeuge, die eine Versionskontrolle unterstützen, nutzen.
- Ich verwende die Programmkombination LAMP. LAMP ist ein Akronym. Die einzelnen Buchstaben des Akronyms stehen für die verwendeten Komponenten Linux (Betriebssystem), Apache (Webserver), MySQL (Datenbank) und PHP (Programmiersprache). 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 das 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 sehr einfach und intuitiv ist. Zumindest dann, wenn alle Systemvoraussetzungen erfüllt sind.

Ich beschreibe hier die Installation unter Ubuntu Linux und einer Standard LAMP Installation. Falls Sie mit einem anderen Betriebssystem arbeiten passen Sie die Beschreibung bitte an Ihre Systemumgebung an.

Was ist Joomla! eigentlich?

Bevor Sie 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 zahlen. Außerdem können Sie den Quellcode einsehen. Joomla! ist eine Open Source Software die unter

der Lizenz GNU General Public License Version 2 or later veröffentlicht ist. (ToDo Opensource hier oder im nächsten Kapitel)

Voraussetzungen

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

Download des Joomla! Installationspaketes

Besorgen Sie sich als erstes das aktuelle Joomla! Installationspaket. Die neueste Version finden Sie immer unter der Adresse <https://www.joomla.org/download.html>. Eine Installationsdatei, die die deutschen Sprachpakete enthält finden Sie auf der Website <https://www.jgerman.de/>. Da ich ein deutsches Buch schreiben habe ich das deutsche Installationspaket in der Version 3.6.5 - also die Datei Joomla_3.6.5-Stable-Full_Package_German.zip - installiert. Sie werden wahrscheinlich eine aktuellere Version herunterladen können.

Upload der Joomla! Installationsdateien auf den lokalen Webserver

Verschieben Sie das heruntergeladene Installationspaket auf Ihren lokalen Webserver und entpacke es dort. Wenn Sie wie ich die Standardinstallation von LAMP nutzen, sollten Sie das Installationspaket also 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_German`. Der Einfachheit halber benennen Sie dieses Verzeichnis bitte um in `/var/www/html/joomla`.

Ab nun können Sie 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



The screenshot shows the Joomla! main configuration interface. At the top, the Joomla! logo is displayed, followed by the text "Joomla!® ist freie Software. Veröffentlicht unter der GNU General Public License." Below this, there are three tabs: "1 Konfiguration" (selected), "2 Datenbank", and "3 Überblick". A language selection dropdown is set to "German (DE-CH-AT)", and a "Weiter" button is visible. The main section is titled "Hauptkonfiguration" and contains several input fields with labels and instructions:

- Name der Website ***: A text input field. Below it, the instruction reads: "Den Namen der Joomla!-Website eingeben."
- Beschreibung**: A larger text input field. Below it, the instruction reads: "Eine Beschreibung der gesamte Website für Suchmaschinen eingeben. Üblicherweise ist ein Maximum von 20 Wörtern optimal."
- Administrator-E-Mail ***: A text input field. Below it, the instruction reads: "Bitte eine E-Mail-Adresse eingeben, die für den Super Administrator der Website genutzt werden soll."
- Administrator-Benutzername ***: A text input field. Below it, the instruction reads: "Den Benutzernamen für das Konto des Super Administrators eingeben."
- Administrator-Passwort ***: A text input field. Below it, the instruction reads: "Das Passwort für das Super Administrator Konto eingeben. Im Feld darunter bitte die Passworteingabe"

Abbildung 3: Joomla! Hauptkonfiguration 992

Der weitere Ablauf der Konfiguraiton 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.



Joomla! ist freie Software. Veröffentlicht unter der GNU General Public License.

1 Konfiguration 2 Datenbank 3 Überblick

Zusammenfassung [← Zurück](#) [→ Installieren](#)

Beispieldaten installieren: ☒ Keine **Benötigt für eine automatisch standardmäßig eingerichtete multilinguale Webseitenerstellung**

☐ Englische (GB) Beispieldaten: Bloginhalte

☐ Englische (GB) Beispieldaten: Prospektinhalte

☐ Englische (GB) Beispieldaten: Standardinhalte

☐ Englische (GB) Beispieldaten: Joomla! erlernen

Anfängern wird dringend empfohlen diese Daten zu installieren. Hiermit werden die Beispieldaten eingefügt, die dem Installationspaket von Joomla! beiliegen.

Überblick

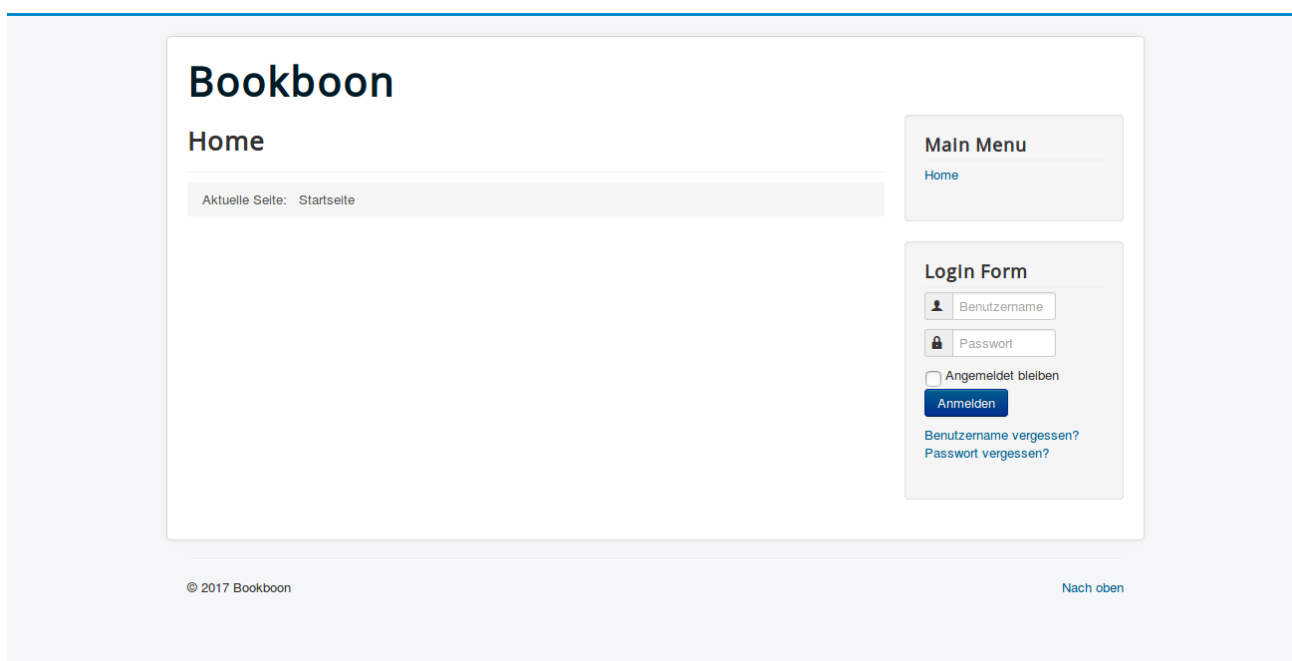
Konfiguration senden:

Konfigurationseinstellungen nach der Installation an `admin@example.de` per E-Mail senden.

Hauptkonfiguration **Konfiguration der Datenbank**

Abbildung 4: Zusammenfassung der Joomla! Installation - Hier bitte auf Beispieldaten verzichten 991.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.



Bookboon

Home

Aktuelle Seite: Startseite

Main Menu

[Home](#)

Login Form

☐ Angemeldet bleiben

[Benutzername vergessen?](#)

[Passwort vergessen?](#)

© 2017 Bookboon [Nach oben](#)

Abbildung 5: Joomla! Frontend unmittelbar nach der Installation 988.png

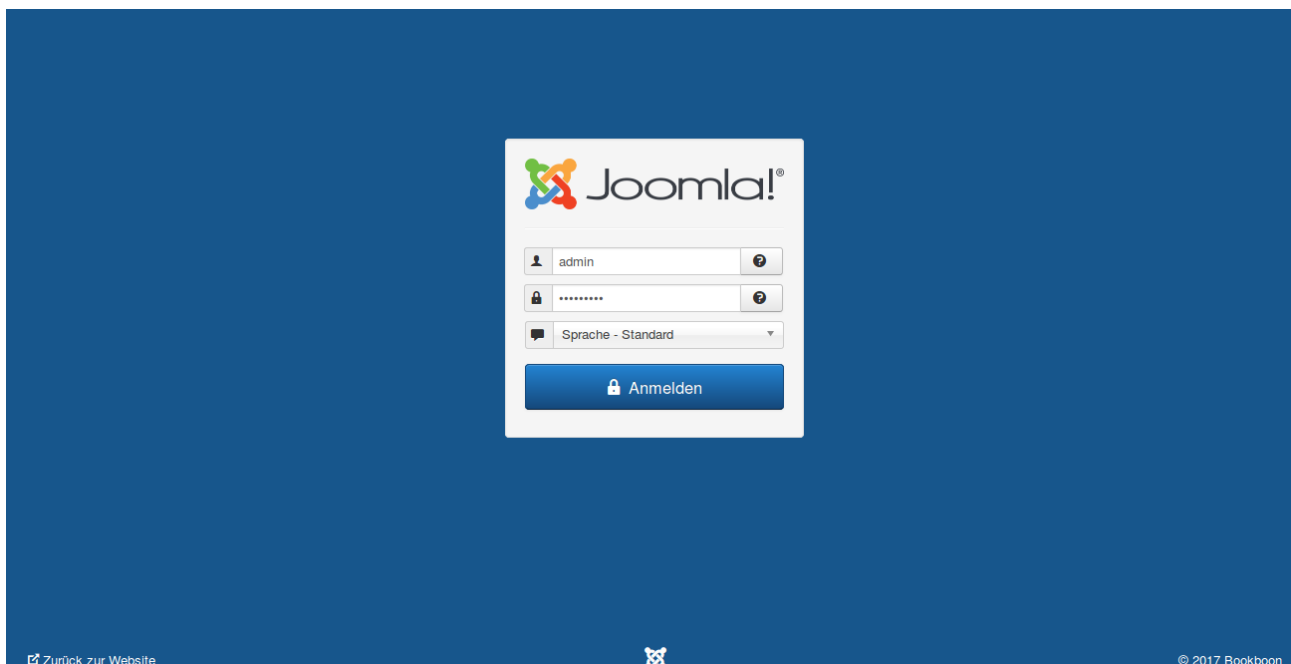


Abbildung 6: Anmeldemaske zum Joomla! Administrationsbereich 987.png

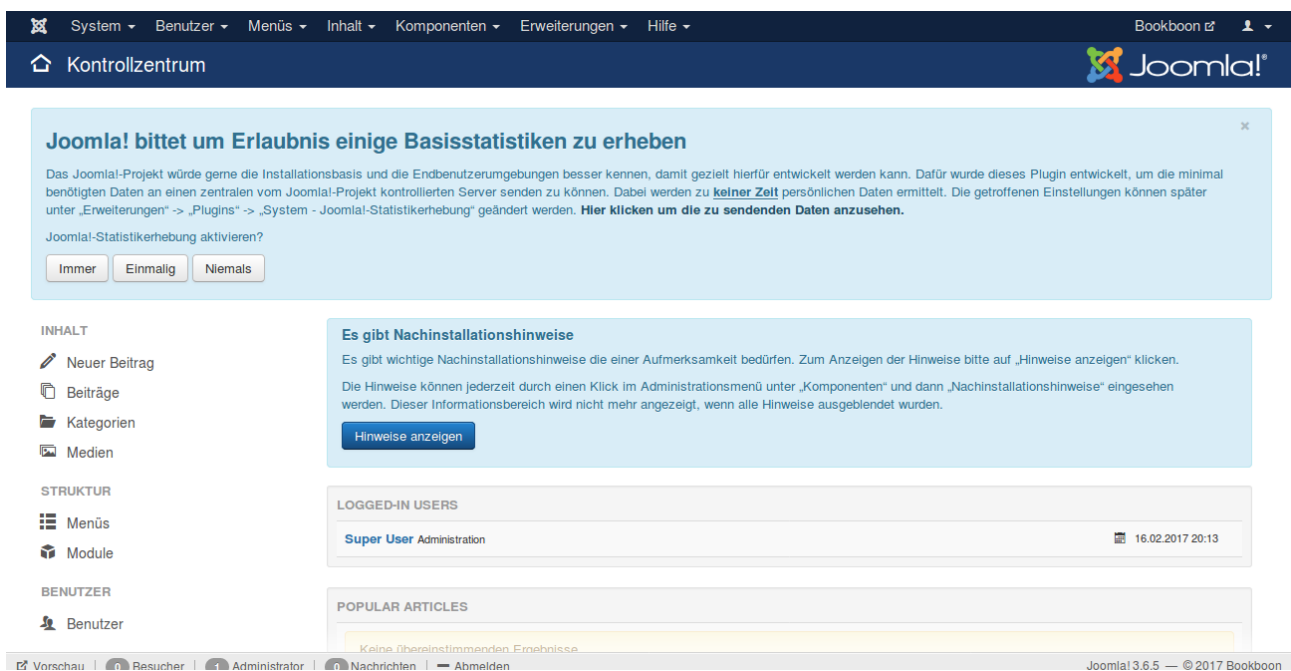


Abbildung 7: Joomla!: Die erste Anmeldung im Administrationsbereich 987.png

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
```

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 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!

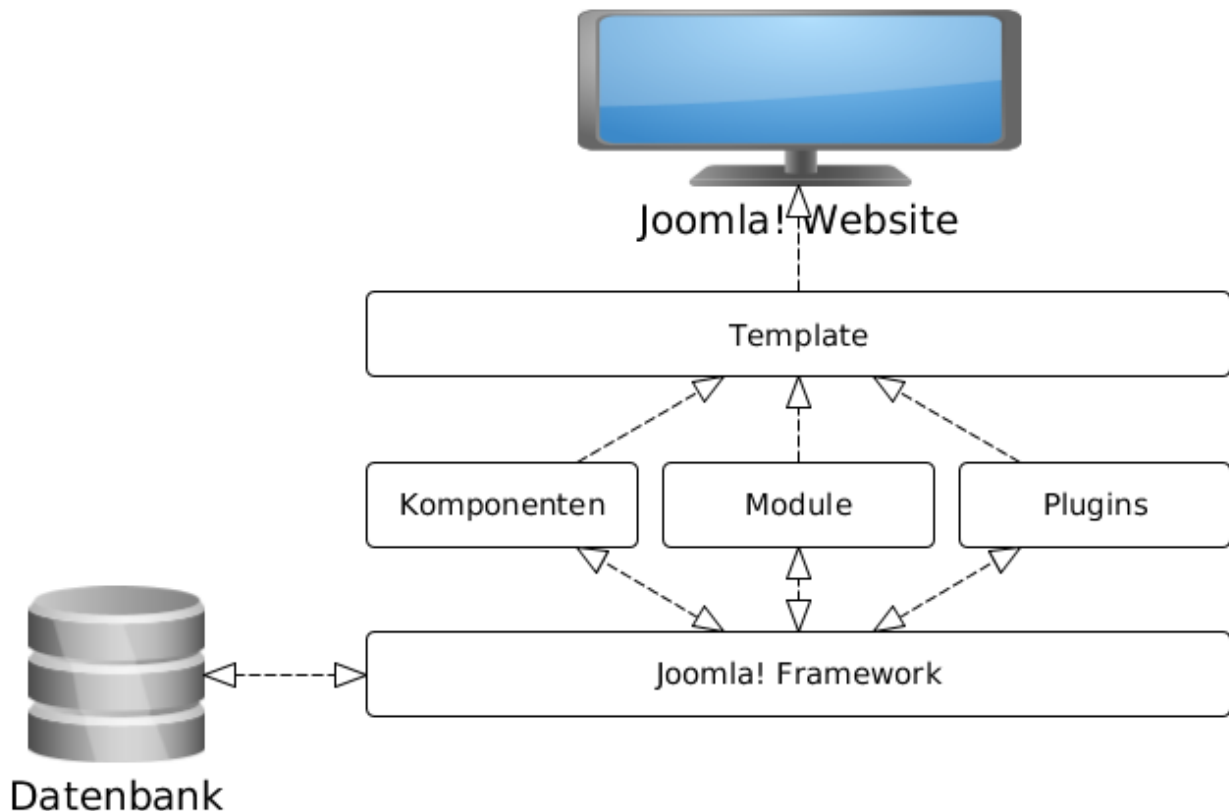


Abbildung 8: Joomla! Architektur 996

Todo schöner formulieren.

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 aufbauen.

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! unterscheiden zum einen die

- geschützten Erweiterungen. Das sind Erweiterungen, die Sie bei der Standardinstallation von Joomla! mit installiert haben.

- Außerdem gibt es für fast jede Problemstellung Erweiterungen von Drittanbietern.

Todo link JED

Erweiterungstypen

ToDo Einleitung und wo in der Verzeichnisstruktur zu finden ..

- Komponente

Unter einer Komponente können Sie sich eine kleine Anwendung vorstellen.

Diese Anwendung erfordert das Joomla! Framework als Grundlage, ansonsten können Sie diese aber eigenständig nutzen und mit ihr interagieren. Ein Beispiel für eine geschützte Komponente ist der Benutzermanager. Komponenten von Drittanbietern finden Sie im Joomla! Extension Directory

<https://extensions.joomla.org>.

- Modul

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, der ein weiteres Beispiel für eine geschützte Komponente ist, konfigurieren Sie ein Modul. Das wohl bekannteste Modul ist *Eigenes HTML*, mit dem Sie individuelle Text mittels der Hypertext-Auszeichnungssprache HTML auf Ihrer Website anzeigen können.

- Plugin

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. Das Ereignis das wir in unserem Beispiel Plugin agpaypal ausnutzen werden ist der erste Schritt in der Aufbereitung der Anzeige eines Beitrags im Frontend. Ein Plugin ist eine einfache aber sehr effektive Art das Joomla! Framework zu erweitern.

- Template

Das Template bestimmt das Aussehen Ihrer Joomla! Website.

Joomla! mit einem eigenen Plugin erweitern

Zu Beginn dieses Kapitels haben Sie Joomla! installiert. Danach haben ich Ihnen die wichtigsten Bestandteile von Joomla! erläutert. In diesem Abschnitt werden wir nun eine einfache Erweiterung schreiben. Aufgabe dieser Erweiterung ist es, einen bestimmten Text in eine PayPal Schaltfläche umzuwandeln. Wir werden dazu das

Ereignis OnContentPrepare nutzen. Dieses Ereignis wird in Joomla! beim Vorbereiten eines Beitrags für die Anzeige im Browser ausgelöst.

Ein Joomla! Plugin muss im Grunde genommen 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 mit folgendem Inhalt.

```
/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;
}
}

```

(https://www.paypal.com/de/cgi-bin/webscr?cmd=_pdn_xclick_techview_outside,
https://docs.joomla.org/J3.x:Creating_a_Plugin_for_Joomla Todo)

Danach erstellen Sie im gleichen Verzeichnis die Datei agpaypal.xml mit folgendem Inhalt.

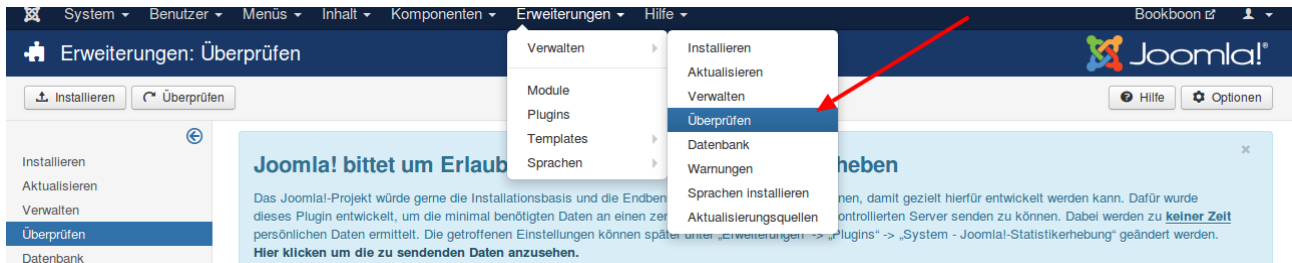
```

<?xml version="1.0" encoding="utf-8"?>
<extension version="3.5" type="plugin" group="content" method="upgrade">
  <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>
    <folder>language</folder>
  </files>
</extension>

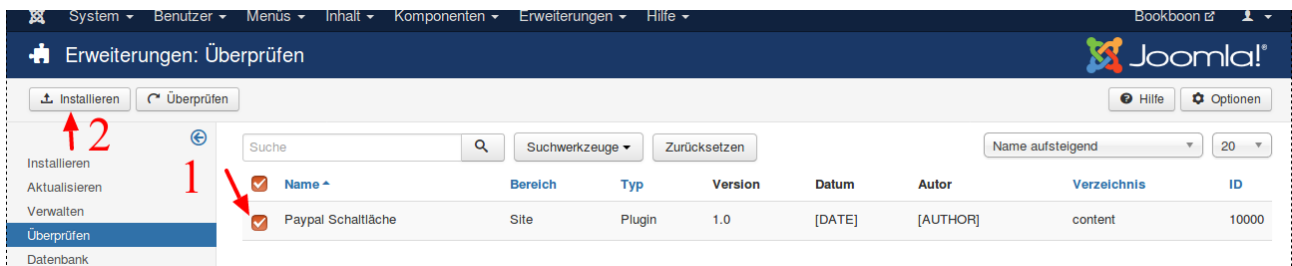
```

(Todo Bevor wir die Erweiterung in Joomla! ausprobieren erklären ich Ihnen kurz die bisher verwendeten Programmcode-teile. Danach werden wir die Erweiterung testgetrieben weiter bearbeiten. Irgendwann wollen Sie sicherlich einmal etwas anderes als einen Teddy für 12,99 Euro verkaufen, oder?)

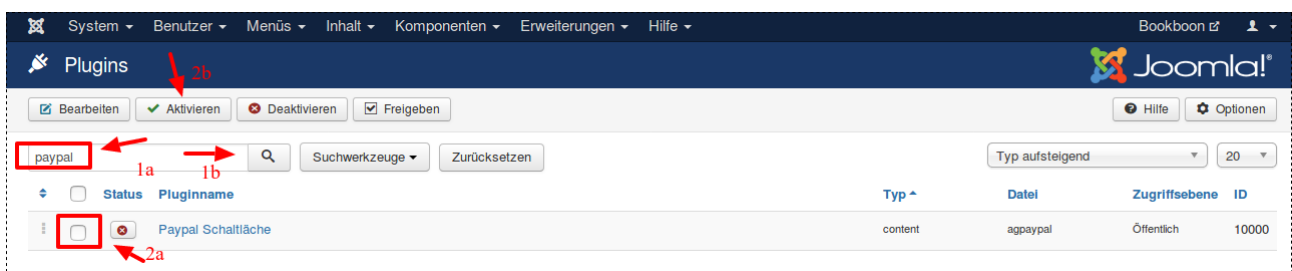
Damit Joomla! Ihre Erweiterung kennenlernen muss das Content Management System diese noch entdecken. Öffnen Sie dazu bitte im Administrationsbereich das Menü Erweiterungen|Verwalten|Überprüfen und klicken dann links oben auf die Schaltfläche Überprüfen.



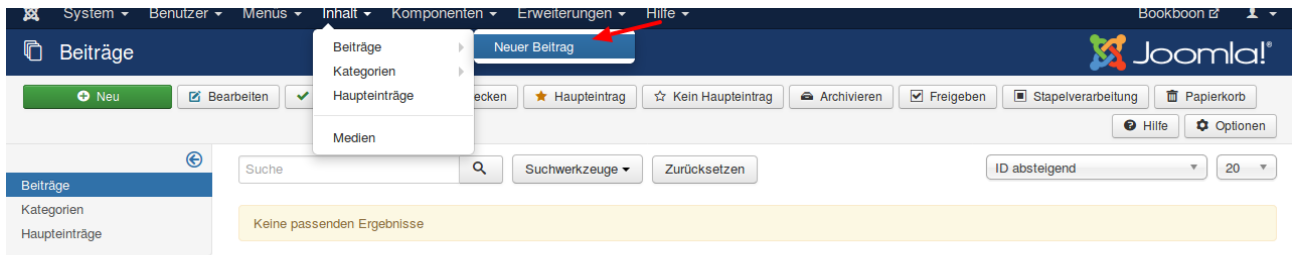
Wenn Sie die Dateien genau wie ich es Ihnen beschrieben habe erstellt haben sehen Sie nun im Hauptbereich einen Eintrag, der Ihr eben erstelltes Plugin beschreibt. Wählen Sie diesen Eintrag aus und klicken danach links oben auf die Schaltfläche Installieren.



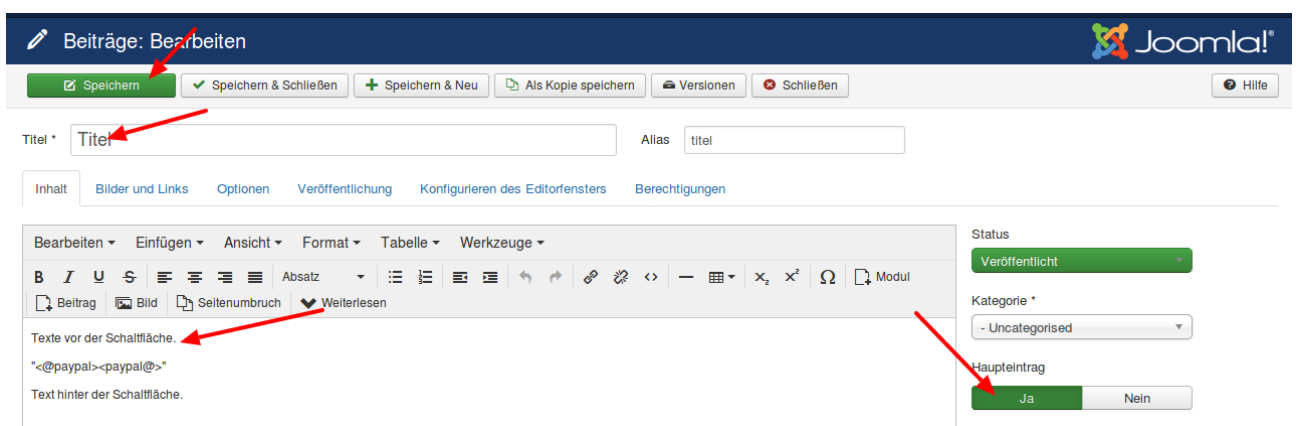
Wenn alles richtig läuft wird Ihnen nun gemeldet, dass das Installationspaket installiert wurde. Überprüfen Sie über das Menü Erweiterungen|Plugins ob Joomla! Ihr Plugin nun wirklich kennt und aktivieren Sie es im nächsten Schritt, indem Sie die Checkbox vor dem Plugineintrag selektieren und in der Werkzeugleiste auf die Schaltfläche Aktivieren klicken.



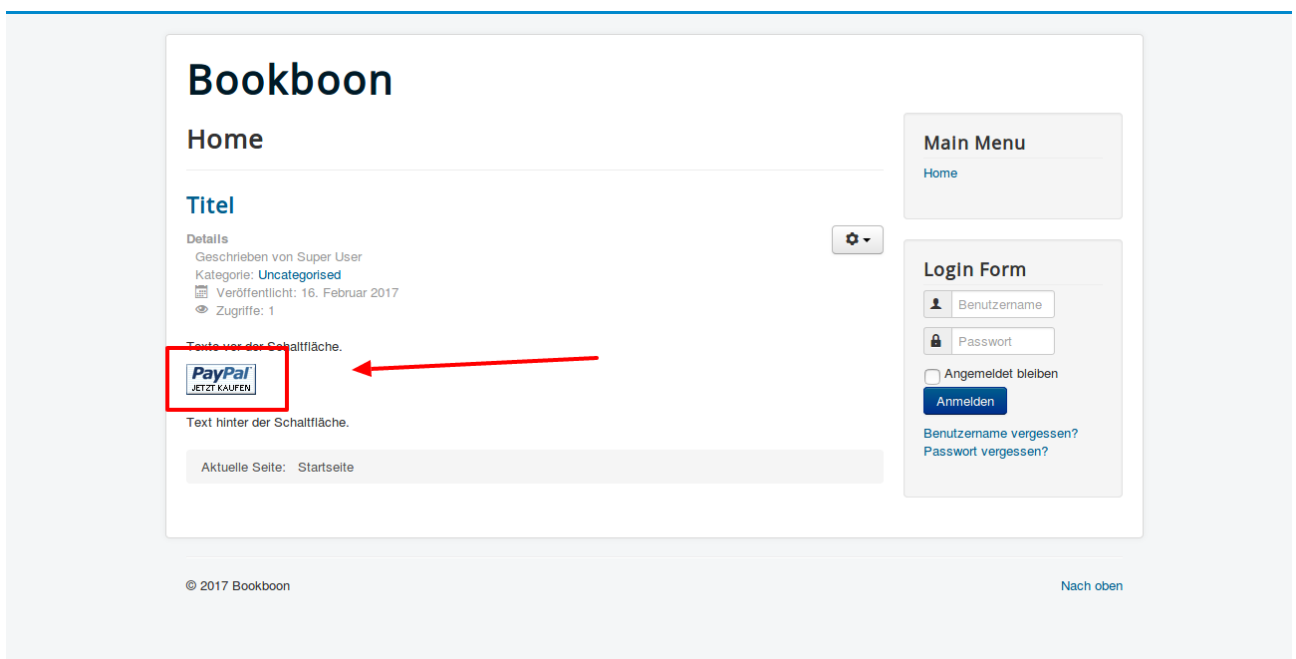
Ihr Plugin ist nun aktiv und wenn Sie einen neuen Beitrag erstellen, in den Sie den Text "@paypalpaypal@" einfügen, erscheint im Frontend anstelle des Textes eine PayPal „Jetzt kaufen“-Schaltfläche. Probieren Sie es aus. Erstellen Sie als erstes einen Beitrag, indem Sie im Administrationsbereich das Menü Inhalt|Beiträge|Neuer Beitrag öffnen.



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



Wenn Sie im Browser das Frontend aufrufen.



So die Erweiterung funktioniert. Da wir testgetrieben entwickeln wollen, installieren wir als nächstes Codeception im nächsten Kapitel. Vorher sehen wir uns aber die Testmöglichkeiten mit Codeception und die Philosophie die dahinter steckt kurz theoretisch an.

Unsere Tests mit Codeception planen

Die Aufgabenstellung der Software, die wir erstellen wollen, ist definiert. Wie integrieren wir am besten welche Tests? Was bietet Codeception uns für Möglichkeiten?

Testtypen

Codeception unterstützt Sie beim Erstellen von

- Unittests
- Integrationstests
- Akzeptanztests

Da zu Beginn eines Softwareprojektes noch nicht sicher ist, wie das Programm am Ende genau aussieht, fällt das Planen von Tests schwer. Man tappe sozusagen im Dunkeln. Sinnvoll ist es, die Aufgabenstellung in einzelne unabhängige Module zu unterteilen.

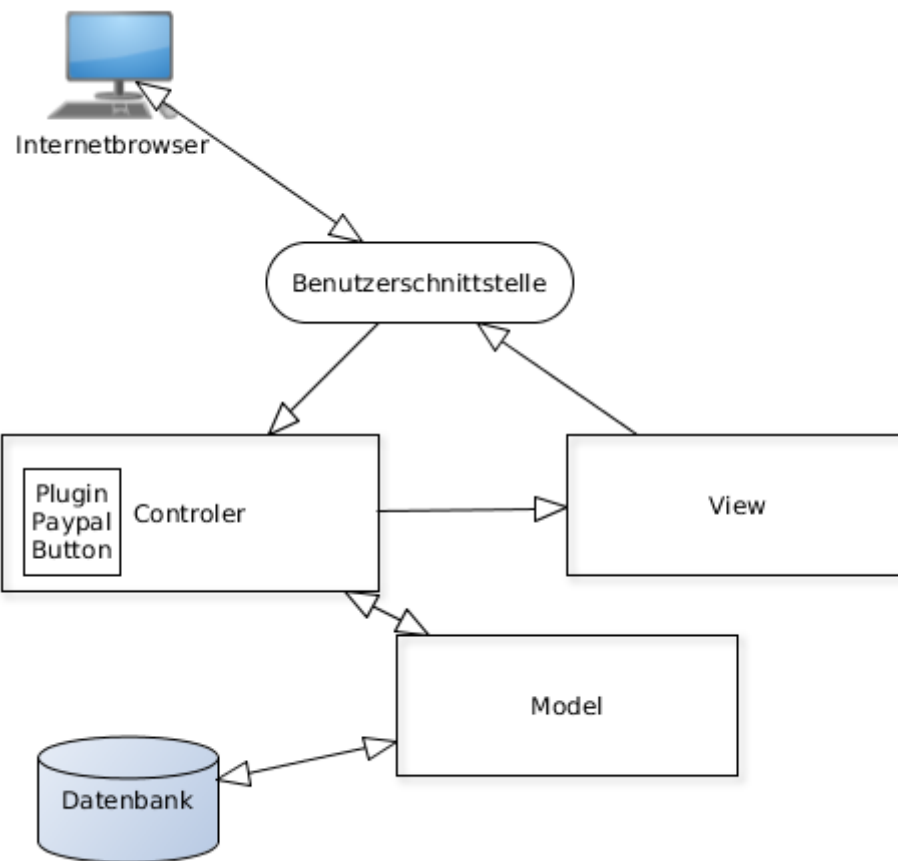


Abbildung 9: Eine Webanwendung mit Datenbank und Browerausgabe 990.png

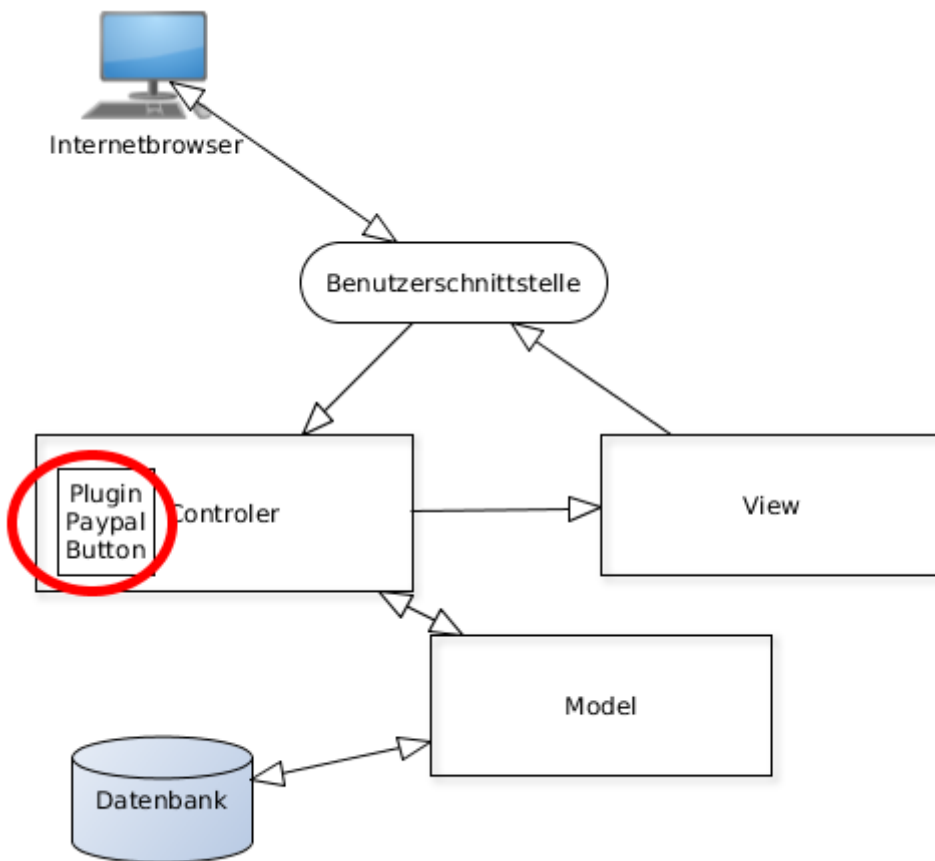


Abbildung 10: Unittest - Eine kleine Einheit innerhalb eines Systems testen. 9990aUnittest.png

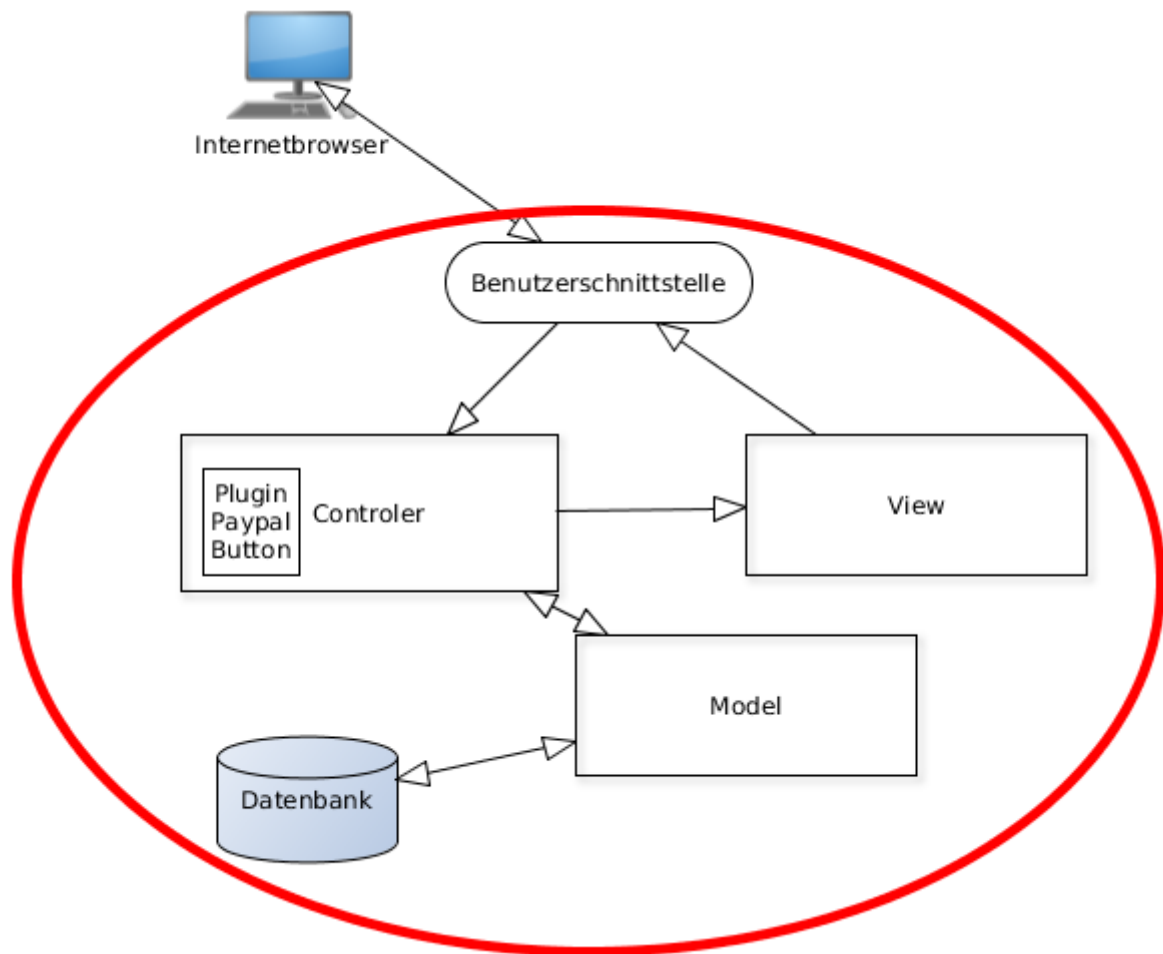


Abbildung 11: Funktionstests oder Integrationstests; Das Zusammenspiel der einzelnen Einheiten testen. Ein Anwender wird nicht simuliert.

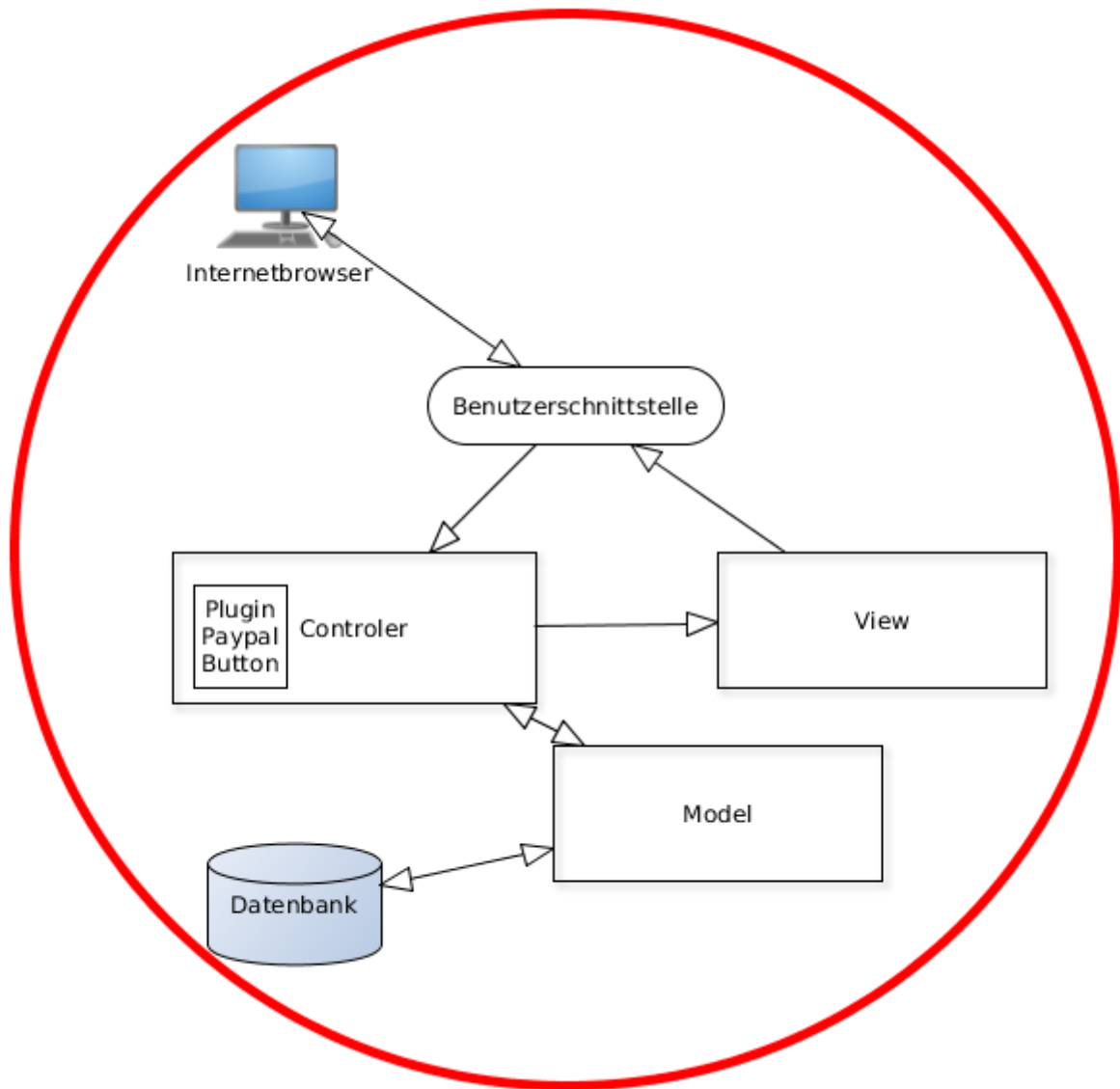
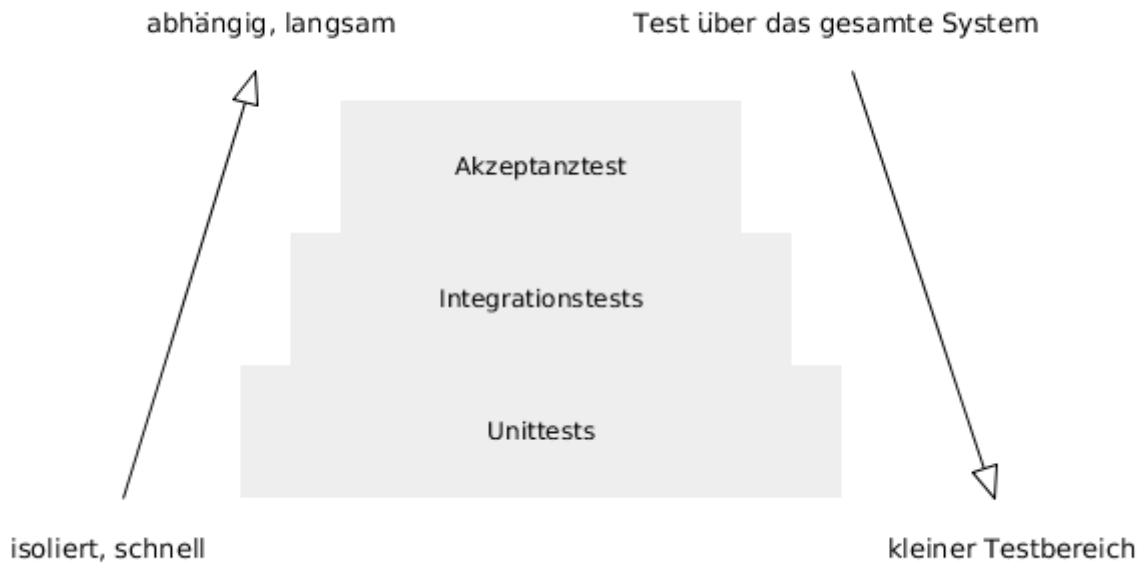


Abbildung 12: Akzeptanztests; Anwendungsfälle werden anhand von (automatisierten) Benutzereingaben getestet.

Teststrategie

Top-Down- und Bottom-Up-Testen

top-down: Haupt- vor Detailfunktionen testen; untergeordnete Routinen werden beim Test zunächst ignoriert oder (mittels "Stubs") simuliert



bottom-up: Detailfunktionen zuerst testen; übergeordnete Funktionen oder Aufrufe werden mittels "Testdriver" simuliert

Testplan

Im 2. Kapitel möchte ich erklären, wie die Testumgebung am Beispiel des Content Management Systems CMS Joomla! eingerichtet werden kann (<https://github.com/joomla/joomla-cms>). Hier gebe ich auch eine kleine Einführung in den Paket Manager Composer (<https://getcomposer.org/>). Außerdem erkläre ich die Struktur des CMS Joomla! kurz. Ein weiteres Thema wird die Planung der Tests sein: Was soll wie getestet werden.

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]

(todo <http://codeception.com/install>)

Wir möchten Tests für eine Joomla! Erweiterung mit Codeception schreiben. Dafür installieren wir Codeception im Joomla! Projekt selbst. Alternativ könnte Codeception auch global installiert werden. Ich empfehle und zeige Ihnen hier aber die projektspezifische Installation. Um Codeception zu installieren benötigen zunächst die Software [Composer](#).

Composer

Wer oder was ist Composer und wofür wird Composer gebraucht? Um diese Frage zu beantworten, ist es wichtig sich in 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 Projekt die eine oder andere externe Software einsetzen. Ihr PHP-Projekt ist also 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 dies Gott sei Dank 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 von PHP in einem **Gesamtsystem** 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 eigene Pakete Abhängigkeiten hat, automatisch installiert. Welche PHP-Anwendungen verfügbar sind, können Sie über die Plattform [Packagist](#) herausfinden. Composer ist noch recht neu. Die erste Version wurde im März 2012 veröffentlicht. An der Entwicklung von Composer können Sie sich beteiligen – Composer wird auf [Github](#) entwickelt.

Installation

Sie können Composer lokal in Ihrem aktuellen Joomla! Projekt installieren - oder global, etwa in im Verzeichnis `/usr/local/bin`. Diese Verzeichnis ist von Haus aus in der Umgebungsvariable `$PATH` hinterlegt, und wird bei einem Programmaufruf nach dem

entsprechenden Programm durchsucht. (Todo Um PATH dauerhaft und systemweit zu erweitern, muss die Datei /etc/environment editiert werden.)

Ich habe Composer unter Ubuntu 16.04 mithilfe im Verzeichnis /usr/local/bin installiert. Dazu habe ich die Datei composer.phar mittels

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

heruntergeladen. Eventuell müssen Sie vorher die Paketquellen in der /etc/apt/sources.list aktualisieren.

```
$ sudo apt-get update
```

Falls wget noch nicht auf Ihrem Rechner installiert ist, können Sie es mit dem Befehl

```
$ sudo apt-get install wget
```

installieren.

Danach habe ich die Datei composer.phar in composer umbenannt und ausführbar gemacht.

```
$ mv composer.phar composer  
$ chmod +x composer
```

Nun kann ich Composer lokal, also in dem Verzeichnis in dem die Datei abgelegt ist, ausführen.

```
$ ./composer
```

Um auch global auf den Paketmanager zugreifen zu können, habe ich die Datei in das Verzeichnis /usr/local/bin verschoben.

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

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

```
$ composer
```

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

Die Dateien `Composer.json` und `Composer.lock`

Wenn sie den Befehle Composer install ausführen, liest Composer die Datei `composer.json` im aktuellen Verzeichnis. Falls es diese Datei nicht gibt sieht die Ausgabe wie folgt aus.

```
$ composer install
```

```
Composer could not find a composer.json file in /home/astrid
```

```
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 eine `Composer.json`, die die Installation von Codeception erzwingt, anlegen. Sie könnten diese Datei von Hand manuell erstellen.

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

Sie schützen sich aber vor 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 diese Befehle im nächsten Kapitel anhand der Installation von Codeception ausführen. 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 soviel vorab: In diesem Fall werden alle Pakete, die zur Installation der in der Datei `composer.json` aufgeführten Programme notwendig sind, heruntergeladen und im Unterverzeichnis `/vendor` installiert. Außerdem wird die Datei `composer.lock` im Stammverzeichnis angelegt. Die Datei `composer.lock` dokumentiert die heruntergeladenen Versionsstände der einzelnen Pakete. Wenn Sie Ihr Projekt mit anderen Teilen möchten, können Sie mithilfe der `composer.lock` immer sicherstellen, dass alle Projektbeteiligten mit den gleichen Versionen arbeiten. Auch, dann wenn ein Teilpaket in der Zwischenzeit aktualisiert wurden.

Wenn Sie die neuere Version des zwischenzeitlich aktualisierten Paketes einsetzen möchten, laden Sie diese zunächst in Ihr Projekt und sehen sich in Ruhe an, ob die Änderungen negative Auswirkungen auf Ihr Projekt haben. Falls dies nicht so ist, können die Versionsnummer in der `composer.lock` hoch setzen. Dies bewirkt, dass in Ihrem Projekt zukünftig die neuere Version automatisch über Composer installiert wird.

Todo warum das `/vendor` verzeichnis nicht committen.

Paketverwaltung

Format	Trifft zu auf	Beispiel
<code>^1.0</code>	<code>>= 1.0 < 2.0</code>	1.0, 1.2.3, 1.9.9
<code>^1.1.0</code>	<code>>= 1.1.0 < 2.0</code>	1.1.0, 1.5.6, 1.9.9
<code>~1.0</code>	<code>>= 1.0 < 2.0.0</code>	1.0, 1.4.1, 1.9.9
<code>~1.0.0</code>	<code>>= 1.0.0 < 1.1</code>	1.0.0, 1.0.4, 1.0.9
<code>1.2.1</code>	<code>1.2.1</code>	1.2.1
<code>1.*</code>	<code>>= 1.0 < 2.0</code>	1.0.0, 1.4.5, 1.9.9
<code>1.2.*</code>	<code>>= 1.2 < 1.3</code>	1.2.0, 1.2.3, 1.2.9

Codeception

Installation

Als nächstes installieren wir Codeception so wie die Codeception Website dies [vorschlägt](#).

```
/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]
```

```
/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.
```

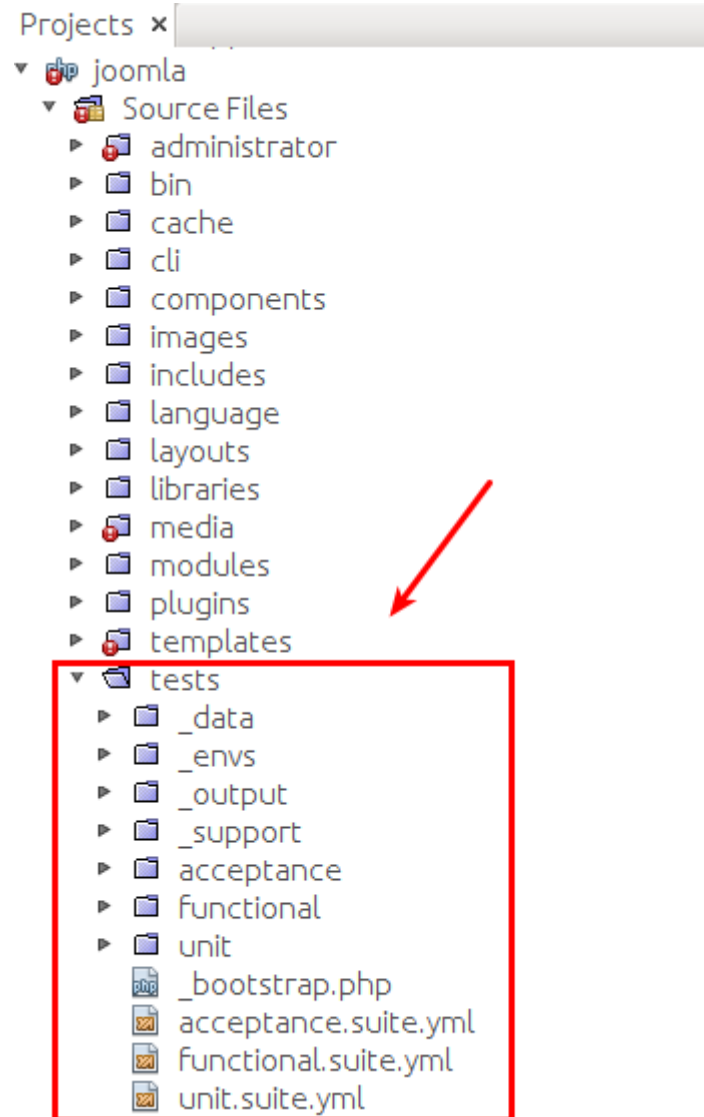
-> FunctionalTesterActions.php generated successfully. 0 methods added

\FunctionalTester includes modules: \Helper\Functional

FunctionalTester.php created.

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

-nun gibt es tests verzeichnis



```
/var/www/html/joomla$ composer require codeception/codeception
```

./composer.json has been updated

Loading composer repositories with package information

Updating dependencies (including require-dev)

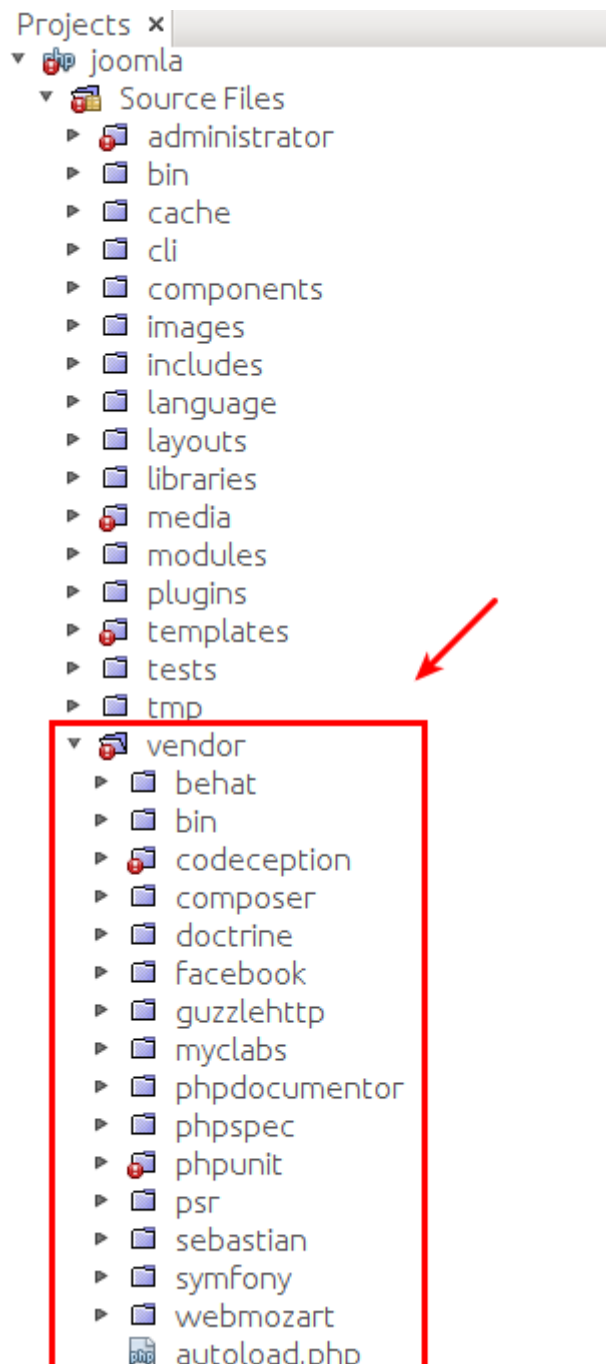
- Installing symfony/yaml (v3.2.4)

```

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

```

Nun gibt es auch vendorverzeichnis



Todo Die Dateien composer.json und composer.lock und wichtige Befehle als Wichtig formatieren.

```
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) -----
-----

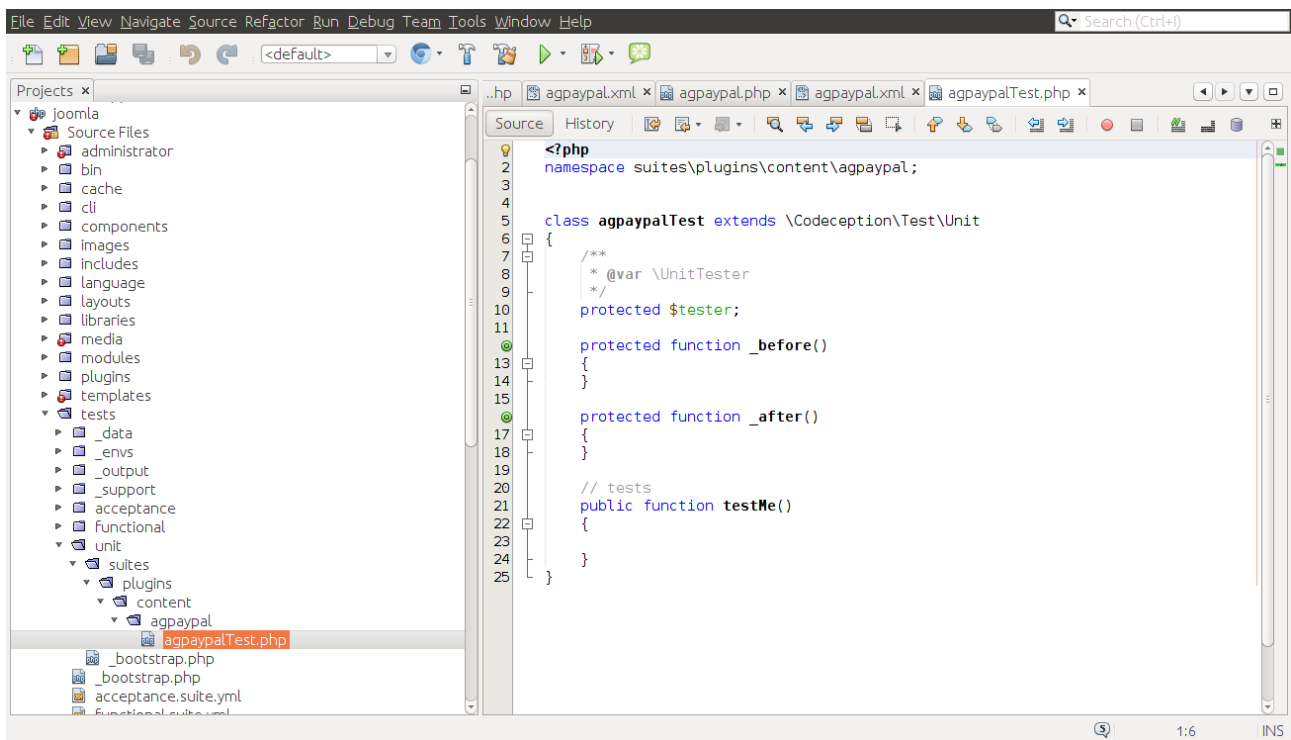
Time: 95 ms, Memory: 8.00MB
No tests executed!
/var/www/html/joomla$
```

Nun erstellen wir den ersten Test

```
/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
```

Die Datei agpaypalTest.php wurde befindet im Verzeichnis

/joomla/tests/unit/suites/plugins/content/agpaypal/ erstellt. Sie enthält nur leere Methoden.



Codeception – ein erster Rundgang

Nachdem nun alles installiert und lauffähig ist könnten wir sofort mit der Implementation der ersten realen Tests starten. Da eine gute Vorbereitung aber die halbe Miete ist, sehen wir uns zunächst das, was Codeception uns bietet genauer an. Ich führe Sie erst einmal durch die Verzeichnisstruktur von Codeception. Wenn Sie sich in dem Programm zurecht finden, ist der zweite Teil des Buches ein Kinderspiel. Sie wissen dann, wie Codeception intern Daten verarbeitet, welche Erweiterungen es gibt, welche Module es gibt und kennen die grundlegende Syntax.

Was steckt in Codeception?

Falls Sie bereits Tests mit einem Testwerkzeug schreiben, kennen Sie die Problematik sicherlich. Jedes Werkzeug hat seine Vorteile, seine Nachteile und seine Tücken. 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 das notwendige Geld um das Wissen einzukaufen.

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 als bekanntestes Tool hat PHPUnit seine Grenzen. Wenn Sie Funktionstests oder Integrationstests schreiben möchten oder gar beim Erstellen von Akzeptanztests werden Sie dieses selbst feststellen. Wie der Name auch 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. Jedes Tool hat seine eigenen individuellen Besonderheiten. Die Arbeit mit dem Programm, die Konfiguration des Programms, die Syntax und nicht zuletzt die Regeln beim Erstellen der Tests müssen neu gelernt werden. Codeception will hier Abhilfe schaffen.

Möchten Sie sich einen Überblick über die verschiedenen Testframeworks nur im Bereich Unittests verschaffen. Beginnen Sie bei [Wikipedia](#).

Codeception ist mehr als nur ein weiteres Werkzeug

Codeception ist kein weiteres Werkzeug. Warum sollte auch ein weiteres neues Werkzeug geschaffen werden? Es gibt ja genug Hilfsmittel. Die Neuschaffung wäre keine Lösung. Am Ende würde ein Tester wieder vor dem gleichen Problem stehen, nämlich unterschiedliche Programme nutzen, aufeinander abstimmen und erlernen müssen. Denn auch ein neues Werkzeug wäre höchstwahrscheinlich nicht die Eierlegende Wollmilchsau die alles gut kann.

Deshalb bündelt Codeception vielmehr verschiedene Tools. Es ist so etwas wie ein **Framework für Frameworks**.

Codeception bietet eine einheitliche Art Tests zu schreiben. Dabei unterstützt es unterschiedliche Testtypen. Die Logik und die Vorgehensweise beim Erstellen der Tests ist mithilfe von Codeception für alle Testtypen einheitlich. Damit wirkt die ganze Testinfrastruktur stimmig und auf schlüssige Weise zusammenhängend.

Todo einzelnen Module erklären?

Codeceptions Konzepte stellen sich vor

Codeception schreibt auf der eigenen Website [selbst über sich](#): Codeception wurde entwickelt um ein Tool zu bieten, dass 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 Projektbeteiligte, die nicht selbst programmieren, einen codierten Test lesen können und ohne weitere Erklärung den Testgegenstand verstehen.

Ich hatte etwas weiter vorne schon geschrieben, dass ein Problem bei der Verwendung unterschiedlicher Testtools die unterschiedliche Syntax ist. Codeception unterstützt beim Erstellen von Tests durch die Vereinheitlichung der Syntax. Tests sollen **einfach zu schreiben** sein!

Fehler sollen leicht gefunden werden. Dafür ist es wichtig, jederzeit die aktuelle Variablenbelegung ablesen zu können. Mit Codeception sind Tests **leicht zu Debuggen**.

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

Zusätzlich unterstützt Codeception die **Wiederverwendbarkeit** von Programmcode eines Tests in anderen Tests.

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 Testtype hat in Codeception seinen eigenen Bereich, sprich Ordner.

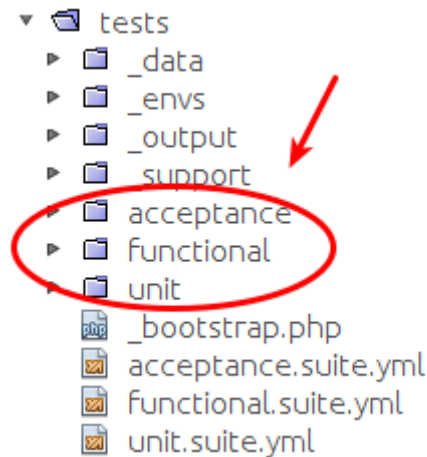


Abbildung 13: Verzeichnisse für die Testtypen innerhalb von Codeception. 976.png

Bevor Sie mit dem Schreiben von Tests beginnen, sollten Sie sicherstellen, dass alle notwendigen Codeception Klassen vorhanden sind. Dies können Sie jederzeit mithilfe des Befehls `vendor/bin/codecept build`.

```
/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 nun genau passiert? Drei Dateien, nämlich die Dateien `UnitTesterActions.php`, `AcceptanceTesterActions.php` und `FunctionalTesterActions.php`, wurden erfolgreich erstellt. Sehen Sie sich diese Klassen einmal genauer an. Diese sind prall gefüllt mit hilfreichen Funktionen die Sie verwenden können. Sie finden die Klassen im Verzeichnis

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

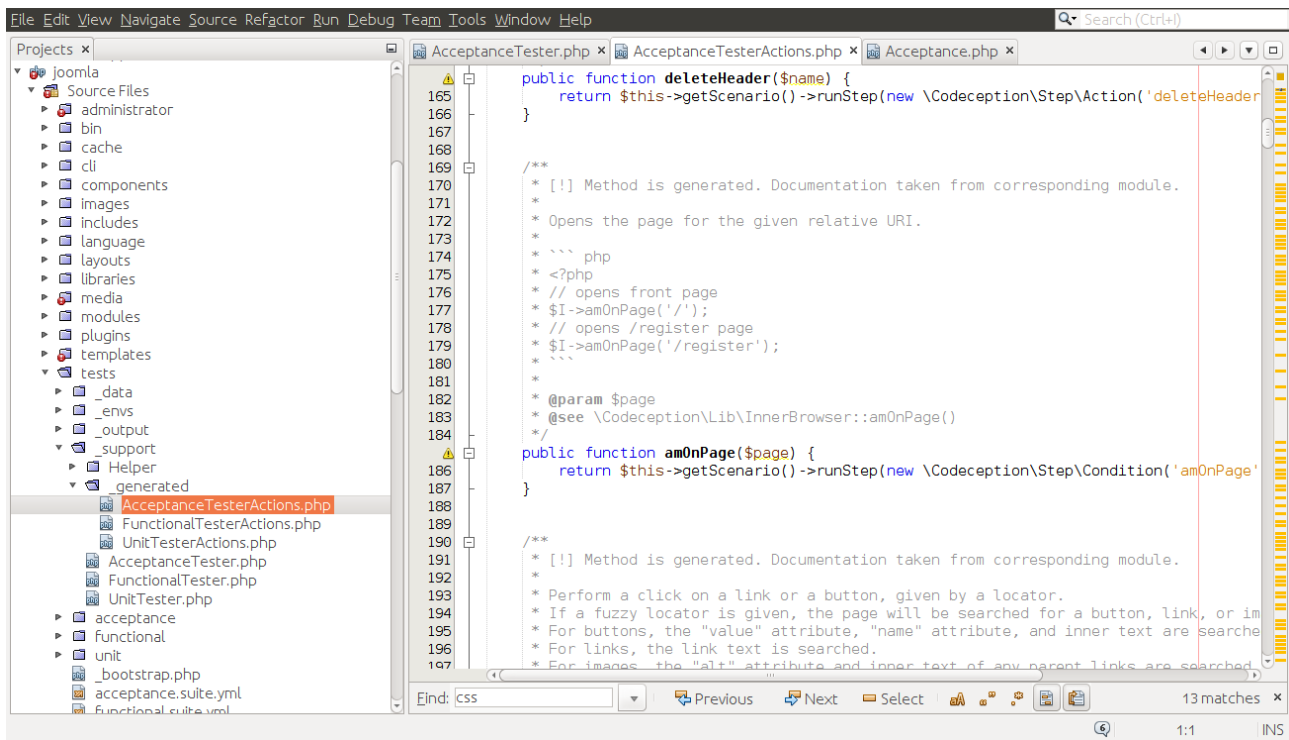


Abbildung 14: Die Klasse AcceptanceTesterAction.php unmittelbar nach der automatischen Generierung 975.png

WICHTIG: vendor/bin/codecept build

Das Kommando `vendor/bin/codecept build` müssen Sie nach jeder Änderung der Konfiguration ausführen. Sie können sich merken: Immer dann wenn Sie eine Datei die mit `.suite.yml` endet und sich innerhalb des Ordners `/tests` befindet ändern, müssen Sie Codeception neu „bilden“.

Sind Sie beim Stöbern im Codeception Ordner auch über die Klassen `UnitTester`, `AcceptanceTester` und `FunctionalTester` gestopert? 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 schafft mit dieser Namensgebung aber ganz nebenbei, dass man beim Schreiben der Tests einen wirklichen Tester, also einen Menschen, der das Programm ausführt, vor Augen hat.

Die verhaltensgetriebene Softwareentwicklung, oder das Behavior Driven Development (BDD) wird so ideal ergänzt. BDD hält idealerweise während der Anforderungsanalyse die geforderten Funktionen der Software in einer bestimmten Textform fest. Diese Funktionen werden dabei meist in „Wenn-dann“-Sätzen beschrieben. Diese „Wenn-dann“-Sätze können nun als Tests ausgeführt werden.

Wenn der **Tester** dies tut, **dann** sollte das passieren. So kann die Software auf ihre korrekte Implementierung getestet werden.

Getestet wird also nicht nur ob **das Programm richtig funktioniert**. Der Tester stellt auch sicher, dass das **richtige Programm erstellt wurde**. Diesen Satz hatte ich an anderer Stelle schon einmal ähnlich geschrieben. Ich finde aber, dass er so wichtig ist, dass man ihn nicht oft genug wiederholen kann.

Ich stelle Ihnen nun die Tester kurz vor, bevor Sie sie in den nachfolgenden Kapiteln in der Praxis kennen lernen.

AcceptanceTester

([Todo Link zu YouTubeVideo von Javier](#)) 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 wie in der Spezifikation festgelegt, funktionieren.

Beim Behavior Driven Development wurde die Spezifikation mittels Beispielen, sogenannten Szenarios, in der Entwurfsphase 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. Jedes Szenario beschreibt das Verhalten der Software in einem Teilbereich. Im Englischen werden Szenarien auch User Story 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

When I Login into Joomla administrator

Then I should see the administrator control pane

Dieses Szenario könnte wie folgt implementiert werden.

```
$I->amOnPage(ControlPanelPage::$url);  
$I->fillField(LoginPage::$usernameField, $username);  
$I->fillField(LoginPage::$passwordField, $password);  
$I->click(LoginPage::$loginButton);  
$I->see(ControlPanelPage::$url);
```

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 hatten Sie sich eben bereits angesehen. Todo Verweis wo praktisch gearbeitet wird.

RANDBEMERKUNG

Falls Ihnen der Name `AcceptanceTester` nicht gefällt, könne Sie diesen in der Konfigurationsdatei ändern.

Akzeptanztests sind, wie im Kapitel `todo` schon beschrieben Tests, die das Benutzerverhalten am realistischsten reproduzieren. Idealerweise laufen Sie unter den gleichen Bedingungen wie das Produktivprogramm ab. Testduplikate werden in der Regel nicht gebraucht. Nur ganz zu Beginn eines Tests sollte ein festgelegter initialer Zustand hergestellt werden. Andernfalls ist es nicht möglich unter den gleichen Testbedingungen den Test zu wiederholen. (todo Verweis zu Wiederholbarkeit, Testduplikaten.)

(Todo Verweis Webdriver, Selenium)

FunctionalTester

Die Bezeichnung Funktionstests ist eine Eigenart in `Codeception`. Im Grunde genommen sind Funktionstests das gleiche wie Integrationstest (todo Verweis).

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,

durchgeführt werden. Browser ohne grafische Benutzeroberfläche nennt man auch [Headless Browser](#).

Außerdem nutzen Funktionstests die einheitliche Schnittstelle des Architekturstil **Representational State Transfer (REST)**. Was können Sie sich genau unter einer REST-Schnittstelle vorstellen? Vereinfacht gesagt können Systeme über eine REST-Schnittstelle automatisch miteinander kommunizieren. Das bedeutet, dass auch Testdaten automatisch ablaufen können, ohne dass ein Mensch zwischendurch eingreifen muss. Im Internet ist ein Großteil der für REST nötigen Infrastruktur, zum Beispiel Webserver oder HTTP-fähige Clients, vorhanden.

So ist eine Website die statische Seiteninhalte nach dem Internetstandard HTTP anbietet REST-konform. Ein Content Management System wie Joomla! erzeugt seine Inhalte dynamisch. Der gleiche HTTP-Request kann, je nach Datenbankinhalt, geänderte Informationen anzeigen. Joomla! erfüllt das REST Paradigma somit nicht vollständig. Zur Veranschaulichung zeige ich Ihnen aber im praktischen Bereich einen Funktionstest, der einen HTTP-Zugriff nutzt. (todo link)

Wird über HTTP auf eine Anwendung zugegriffen, so gibt die verwendete HTTP-Methode an, welche Operation des Dienstes gewünscht ist. Die wichtigsten HTTP-Methoden sind GET, POST, PUT und DELETE. (Todo genauer)

Was im Zusammenhang mit Funktionstests wichtig ist zu erwähnen: Alle Funktionstests laufen im gleichen Speicherbereich ab. Für jeden Test müssen Sie explizit dafür sorgen, dass dieser Speicherbereich beim Abschluss des Tests wieder freigegeben wird. Falls hier etwas nicht richtig läuft, 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. Läuft der Test nun durch, wissen Sie, dass nicht der Test das Problem ist. Die Fehlerursache sollten Sie im Speichermanagement suchen.

UnitTester

Codeception nutzt das Framework PHPUnit für die Erstellung der Unittest. Hier an dieser Stelle gibt es nicht viel mehr dazu zu sagen. Wenn Sie PHPUnit schon verwenden können Sie das nächste Kapitel sicherlich schnell bearbeiten. 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 bietet Ihnen zwei verschiedene Formate. Das Cept-Format ist ein Szenario basiertes Format. Das Cest-Format basiert auf einer Klasse.

Mithilfe des Befehls

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

erstellen Sie eine **Cest**-Datei.

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

Jeder 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 Fehlerfalle.

Mithilfe des Befehls

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

erstellen Sie eine **Cept**-Datei.

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

Nicht-Entwickler bevorzugen oft das CEPT-Format. Entwickler entscheiden sich in der Regel für das CEPT-Format. Das CEPT-Format unterstützt mehrere Tests innerhalb einer Datei. Außerdem können Sie Codes mithilfe von zusätzlichen privaten Methoden leicht wiederverwenden.

EXKURS CEST-Format für Unittests

Meiner Meinung nach eignet sich das Cest-Format nur für Akzeptanztests und Funktionstests. Das Format bietet von Haus aus keine Assertionen und keine Methoden zum Erstellen von Testduplikaten.

Annotationen

Todo, die hatte ich doch schon einmal irgendwo beschrieben, oder?

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 Tests korrigieren und Ihnen ist sicherlich nun schon klar worauf ich hinaus will.

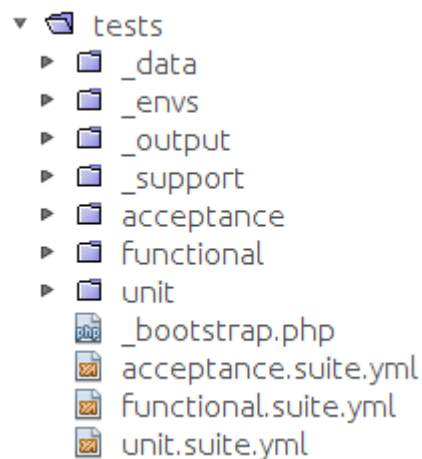
Sinnvoll ist es, Elemente des [Document Objekt Modell \(DOM\)](#) nur einmal hart zu codieren und dann immer wieder zu verwenden. Denn: Falls sich die Eigenschaft eines Elementes einmal ändert, muss nicht die gesamte Testsuite, sondern nur eine Datei geändert werden. Codeception hat hierfür **Page Objekte** 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, darstellen.

Step Objekte sind eine andere Art Programmcode wiederzuverwenden. Codeception hat Step Objekte dazu vorgesehen, Aktionen, die immer wieder vorkommen, anzubieten. 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 aufgenommen sein, der die Auswahl in einem Listefeld ausführt. Die Methode könnte `selectListValue(listenfeld $l, wert $w)` heißen. Immer dann wenn die Auswahl in einem Listefeld simuliert werden soll, würde ein Aufruf von `selectListValue(„tiere“, „hund“)` ausreichen, um die Belegung der Liste `tiere` mit der Option `hund` zu testen.

Codeception unter die Lupe genommen

Nach der Installation von Codeception (todo verweis) haben Sie den Ordner `tests` mit folgendem Inhalt vorgefunden. (todo vendor abhängerigkeiten)



Ordnerstruktur

_data

Im Verzeichnis `_data` können Sie einen Export aus einer Datenbank ablegen. Wenn Sie möchten, können Sie Codeception über die Konfiguration sagen, dass vor Ausführung der Tests eine Datenbank aus diesem Export erstellt wird und die Tests so in einem vordefinierten Zustand ausgeführt werden können. (todo verweis.)

_env

xxx

_output

xxx

_support

xxx

acceptance

xxx

functional

xxx

unit

xxx

Konfiguration

_bootstrap.php

todo

```
<?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';
```

acceptance.suite.yml

todo

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

functional.suite.yml

todo

```
class_name: FunctionalTester
modules:
```

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

unit.suit.yml

todo

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

Arbeiten mit Codeception

Das Sie Codeception über den Befehl codecept ausführen, haben Sie schon an ein paar Beispielen gesehen. Sie haben auch schon Steuerbefehle kennen gelernt, nämlich build, generate und run.

- Mit dem Befehl **build** veranlassen Sie Codeception dazu, Code anhand der Konfigurationsdateien zu konfigurieren. Unter anderem haben wir die Tester über mittels todo erstellt.

- 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 genau den einen Test `agpaypalTest` aus.

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

`tests/unit/suites/plugins/content/agpaypal/agpaypalTest` unterdrückt die Ausgabe in der Konsole. Die Option `q` steht für quite.

- Mit dem Befehl **generate** erstellen Sie Tests. Beim Befehl generate müssen Sie den Namen einer Suite und einen Namen für den Test 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 enthalten.
- `generate:test` erstellt einen PHPUnittest der von der Klasse `Codeception\Test\Unit` erbt und die Methoden `_before()` und `_after()` enthält.

(Todo vielleicht Bild, dass zeigt, dass Unit von PHPUnit erbt)

Die Datei `codecept.php` finden Sie im Verzeichnis `vendor/bin/`. Führen Sie die Datei doch einfach einmal mit der Option `-v` oder ganz ohne eine Option oder ein Argument aus. Dann werden Ihnen alle möglichen Optionen und Argumente angezeigt.

Tests erstellen

Zusammenfassung

Im 3. Kapitel soll es um das Framework Codeception

(<https://github.com/Codeception/Codeception>) gehen. Welche Konzepte beinhaltet es?

Wie erstellt man Tests? Welche Konfigurationsmöglichkeiten gibt es.

Unit Tests

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

[[Ernst Denert](#)]

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 oder, - wenn Sie testgetrieben entwickeln – vor dem eigentlichen Programmcode erstellen sollen. Jeder Test erfolgt, wie der Name schon

vermuten lässt, isoliert von anderen Programmteilen. Unittests testen kleinste Einheiten. Im Deutschen nennt man diese Tests auch Modultests. Bei der testgetriebenen Entwicklung erstellen Sie die Unittests vor jeder kleinen Programmänderung. Beim Erstellen eines Unit Tests kennen Sie die Implementierung der zu prüfenden Unit. Unit Tests zählen aber trotzdem zu den spezifikationsorientierten Tests, also den Black Box Tests. Das Ziel von Unit Tests ist es nicht, den Programmcode systematisch zu überprüfen und möglichst alle Programmcodeabschnitte zu testen. Dies tun die implementationsorientierten White Box Tests. Ziel von Unittests ist es, alle Testfälle abzudecken, egal ob dabei alle Programmcodeabschnitte durchlaufen werden oder nicht. (Verweis zu testfälle und Black und Whitebox tests)

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, Test in ihre eigene Software einzubauen. Falls Sie noch nicht testgetrieben entwickeln, kommt Ihnen diese Methode zunächst sicherlich sehr umständlich vor. Mir ging es anfangs zumindest so. Eine neue Funktionalität einfach einbauen 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 diese Erfahrung nicht mehr missen.

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 Zusatzfunktion von Codeception haben Sie schon benutzen, als Sie den ersten Test während der Installation von Codeception erstellten – nämlich den Testcode-Generator. Mit dem Befehl

```
/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/`

RANDBEMERKUNG (todo an andere Stelle)

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.

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 wird noch kein wirklicher Programmcode ausgeführt. 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() {}
}
```

Die Testklassen, im Beispiel hier die Klasse `agpaypalTest`, werden bewusst getrennt von der Produktionsklassen `plgContentAgpaypal` in einem separaten Verzeichnis abgelegt. Diese Klassen werden Sie später, wenn Ihr Programm fertig ist, nicht mit beim Kunden installieren. (todo andere Stelle?)

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.

Der erste selbst programmierte Test

Was wollen wir testen?

Todo Verweis zu Testfällen.

Die erste Frage ist nun: Was möchten wir genau testen? Dazu 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, 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 Punkte müssen wir testen, wenn wir die korrekte Bearbeitung bei einmaligem Vorkommen des Textes @paypalpaypal@ sicherstellen möchten. Ob andere Dinge, zum Beispiel die Anzeige der Schaltfläche im Beitrag auf der Website, korrekt gehandhabt werden, wird in den dafür zuständigen Klassen geprüft. Wir Testen hier nur diese Einheit unabhängig vom ganzen Programm!

Beim testgetriebenen Erstellen von Unit Tests haben sich in der Praxis die folgenden Regeln bewährt:

1. Prüfen Sie, ob alle schon vorhandenen Test fehlerfrei durchlaufen werden und korrigieren Sie etwaig Fehler.

2. Implementieren Sie Tests, die die kleinste mögliche Einheit im Programmcode testen.
3. Implementieren Sie die Tests unabhängig von einander.
4. Geben Sie den Tests sprechende Namen damit diese gut wartbar und aussagekräftig sind.

Die Implementierung des ersten Tests

Die Methoden in der generierten Testdatei sind noch leer. Dies ändern wir nun! Sie finden die Testdatei `agpaypalTest.php`, also die mit den leeren Methoden, im Verzeichnis `/joomla/tests/unit/suites/plugins/content/agpaypal/`. Zumindest dann wenn Sie diese vorher (todo Verweis wo) mit mir erstellt haben. Öffnen Sie diese Datei bitte nun in Ihrer Entwicklungsumgebung zur Bearbeitung.

Für den ersten Test erstellen wir die Methode

`testPaypalButtonSetCorrectIfSearchstringIsInContentOneTime()`. Jetzt wird es endlich konkret!

Sehen Sie sich zunächst einmal die fertige Methode an:

```
<?php
namespace suites\plugins\content\agpaypal;
class agpaypalTest extends \Codeception\Test\Unit {
    protected $tester;
    protected function _before() { }
    protected function _after() { }
    public function testPaypalButtonSetCorrectIfSearchstringIsInContentOneTime()
    {
        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 Code der Testmethode genauer an: Ich habe für die Bedingung dass der Text „@paypalpaypal@“ zur Anzeige im Frontend in den Text der eine Paypal-Schaltfläche anzeigt umgewandelt wird, eine Behauptung aufgestellt. Dies habe ich mithilfe der assertEquals() Methode getan. Hier musste ich die Parameter so setzten, dass die Bedingung erfüllt sind.

Die Variable \$contenttextafter enthält den Text den ich erwarte und \$contenttextbefore enthält den Eingabetest der zu Beginn den Text „@paypalpaypal@“ enthält. Nach durchlaufen der Methode assertEquals() aber gleich \$contenttextafter sein sollte. Ich behaupte also assertEquals(\$contenttextafter, \$contenttextbefore). Den Rest übernimmt das PHPUnit-Framework.

Der Satz „Hier musste ich nur die Parameter so setzten, dass die Bedingungen erfüllt sind.“ hört sich einfacher als er ist.

- Als erste habe ich ein Objekt des Typs PlgContentAgpaypal instanziiert und in der Variablen \$class gespeichert. Dazu musste ich die Objekte \$subject und \$config erzeugen. Diese Objekte benötigt die Klasse PlgContentAgpaypal bei der Instantiierung.
- 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 die die zu testende Methode aufgerufen. \$class->onContentPrepare("", \$contenttextbefore, \$params);

- Zum Schluss habe ich dann die im Kapitel Was wollen wir testen? auf Seite 62 beschriebenen Bedingungen als Parameter in die Methode `assertEquals()` geschrieben. (Todo) Diese Methode verifiziert die Gleichheit zweier Objekte.

Dann habe ich den Befehl

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

einggegeben um den Test auszuführen.

```
/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: Paypal button set correct 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 Variablen `$contenttextbefore` oder die Variable `$contenttextafter` einbauen. Dann müsste Ihnen nämlich ein Fehler angezeigt werden! (todo wenn platz hier fehler abdrucken)

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

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

```
$this->assertEquals($contenttextafter, $contenttextbefore, "@paypalpaypal@ wurde nicht richtig umgewandelt");
```

Der Text `"@paypalpaypal@ wurde nicht richtig umgewandelt"` wird im zweiten Fall bei einem fehlgeschlagenen Test neben der Nachricht von PHPUnit mit ausgegeben. Dieser kleine Mehraufwand kann auf lange Sicht bei der Testanalyse viel Zeit sparen.

In diesem Buch hier nutze ich diesen Parameter nicht. Die Codebeispiele sind ohne diesen Parameter 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 um ein Gefühl, für diese Methode zu bekommen. (todo testgetrieben immer groß oder klein schreiben?) In der testgetriebenen Entwicklung schreiben Sie den Test noch bevor Sie den Code schreiben, der diesen Test erfüllt. 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 den einfachsten Weg!**

Ein testgetriebenes Programmierbeispiel

(todo hier weiter vorne erklären das Beispiel abgekürzt.) Kommen wir auf unser Beispiel zurück. Bisher wird der Paypal-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? Diese Aufgabe kommt später vielleicht auf Sie zu. Weil dies momentan noch nicht gefordert ist. Wir gehen in kleinen Schritten vor. Beginnen wir mit dem Test.

Erst der Test ...

Indem wir mit zuerst den Test für eine Funktion erstellen, können wir nach jeder Änderung am Code sicherstellen, dass alles geforderten Funktionen erfüllt sind. Was ist nun genau der nächste Testfall? Fangen wir mit einem konkreten Beispiel an. In eine Website soll ein Paypalbutton (todo Paypalbutton immer gleich geschrieben?) eingefügt werden, über den ein Betrag von 15 Euro eingezogen würde. Wir haben uns schon dazu entschieden dem Suchtest einen Parameter mitzugeben. Eine mögliche Umsetzung wäre, den Betrag in die Mitte des Suchtestes 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 Variablen \$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
testPaypalButtonSetCorrectIfSearchstringIsInContentOneTimeAndContainsNoValue(){...}

public function
testPaypalButtonSetCorrectIfSearchstringIsInContentOneTimeAndContainsValueForAmount
()
{
    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.

```
/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: Paypal button set correct if searchstring is in content one time and contains no value
(0.00s)
✗ agpaypalTest: Paypal button set correct 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 Paypalbutton mitgegeben werden, manipulierbar sein sollen. Wahrscheinlich haben Sie auch hierfür schon eine Lösung im Kopf. Komplizierte Designelemente, die nicht unbedingt notwendig sind, sollen Sie aber aufschieben. Sie programmieren nur das, was Sie tatsächlich jetzt gerade benötigen. Nicht das, was später vielleicht notwendig ist.

Dazu suchen wir die Texte „@paypal“ und „paypal@“ mithilfe von `preg_match_all()`. `preg_match_all()` speichert alle Funde in der Variablen `$matches[0][0]` und der Suchtext `(.*)` extrahieren den Text zwischen `@paypal` und `paypal@` in der Variablen `$matches[1][0]`. Nun prüfen wir, ob `$matches[1][0]` den Text „amount=“ enthält. Falls ja, setzen wir den auszutauschenden Suchtext auf `$matches[0][0]` und passen den Wert für den Preis an. Falls nicht lassen wir alles beim Alten. So ist auch der erste Testfall noch erfüllt.

(todo `preg_match_all()`)

```

<?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. 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: Paypal button set correct if searchstring is in content one time and contains no value
(0.01s)
✓ agpaypalTest: Paypal button set correct 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 Paypalbutton einfügen will, sich vertippt?

Und weiter – erst testen, dann programmieren

Das Zusammenspiel von Testen und Implementieren bildet das fertige Programm. Das Programm wird in ganz kleinen Schritten geschrieben. Der vorhergehende Testfall wurde erfüllt. Nun gibt es eine neue Aufgabe.

Die Aufgabe ergibt sich entweder 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 froh 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 greife. 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
testPaypalButtonSetCorrectIfSearchstringIsInContentOneTimeAndContainsIncorrectValue()
{
    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 Paypalbutton eingefügt. Der Text 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: Paypal button set correct if searchstring is in content one time and contains no value (0.01s)
```

```
✓ agpaypalTest: Paypal button set correct if searchstring is in content one time and contains value for amount (0.00s)
```

```
✗ agpaypalTest: Paypal button set correct 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, setzten wir den auszutauschenden Text auf \$matches[0][0]. Den Preis belassen wir auf den Standardwert.

```
<?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: Paypal button set correct if searchstring is in content one time and contains no value
(0.00s)
✓ agpaypalTest: Paypal button set correct if searchstring is in content one time and contains value for
amount (0.00s)
✓ agpaypalTest: Paypal button set correct 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 diesem Programm gibt.

Weiter geht es Schritt für Schritt

Machen wir mit der nächsten Baustelle weiter. Laut Spezifikation kann es vorkommen, das in einem Text mehrere Paypalbuttons vorkommen. Was sagt ein Test dazu?

```

...
public function testPaypalButtonSetCorrectIfSearchstringIsInContentManyTimes()
{
    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);
    }

```

Mehrer Paypalbutton unterstützt das Plugin in der aktuellen Version leider noch nicht.

```

/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: Paypal button set correct if searchstring is in content one time and contains no value
(0.00s)
✓ agpaypalTest: Paypal button set correct if searchstring is in content one time and contains value for
amount (0.00s)
✓ agpaypalTest: Paypal button set correct if searchstring is in content one time and contains incorrect
value (0.00s)
✗ agpaypalTest: Paypal button set correct if searchstring is in content many times (0.00s)
...

```

FAILURES!

Tests: 4, Assertions: 4, Failures: 1.

Was müssen wir tun, um diese 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 das, was wir bisher codiert haben, 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: Paypal button set correct if searchstring is in content one time and contains no value
(0.00s)
✓ agpaypalTest: Paypal button set correct if searchstring is in content one time and contains value for
amount (0.00s)
✓ agpaypalTest: Paypal button set correct if searchstring is in content one time and contains incorrect
value (0.00s)
✓ agpaypalTest: Paypal button set correct 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 immer versucht die einfachste Lösung umzusetzen. Manchmal ist der erste Versuch aber nicht der einfachste. Das bedeutet, dass wir zusätzlich fortlaufend auch einen Blick auf das Design werfen sollten. Wir sollten auch dieses beständig verbessern. Der Code sollte so einfach und verständlich wie möglich bleiben. Redundanzen sollten schon im Keim erstickt werden. Prüfen Sie Ihren Code immer wieder und bringen ihn in eine einfache Form. Kommt Ihnen dies zu umständlich vor? Vertrauen Sie darauf, dass gut strukturierter Code während der ganzen Entwicklungsphase gut erweitert werden kann. (Todo Was habe ich gemacht)

```
<?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;
}
}

```

Das Plugin ist nun wieder übersichtlich. Es gibt aber noch viel zu tun. Welcher Aufgabe stellen Sie sich als nächstes? Beginnen Sie wieder mit dem Test

Im Buch hier legen wir nun aber den Schwerpunkt in Testsoftware. Wir implementieren keine neuen Funktionen ins Plugin. Wir sehen uns vielmehr an, wie wir besser Testen können und welche Möglichkeiten uns Codeception bietet.

Was bietet PHPUnit Ihnen

Fassen wir noch einmal zusammen. Wir haben eine Testdatei erstellt. Diese beinhaltet eine Klasse, die genau so heißt wie die Klasse, die sie überprüft. Am Ende des Namens haben wir lediglich das Wort Test angefügt. In dieser Testklasse werden alle Testfälle mit Bezug zu dieser Klasse aufgenommen. 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 läuft der Test erfolgreich durch. Zur Prüfung bietet PHPUnit (Todo PHPUnit überall gleich geschrieben?) in der Klasse Assert unterschiedliche Prüfmethode. Unsere Klasse erbt diese Prüfmethode. (Todo UML zur Vererbung.)

Die Testmethoden der Klasse Assert

Eine Liste aller Prüfmethode können Sie im Internet unter der Adresse <https://phpunit.de/manual/current/en/appendixes.assertions.html> abrufen.

Die wichtigsten Assertions sind meiner Meinung nach

- `assertTrue()`
- `assertEquals()` Todo

Der Rückgabewert dieser Methoden ist Ausschlaggebend dafür, ob ein Test erfolgreich ist oder nicht. Vermeiden Sie es Ausgabe einer Assert-Methode von mehreren Bedingungen gleichzeitig abhängig machen.

Annotationen

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

In PHP findet man 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 erwartet, welchen Rückgabewert es gibt oder welche Variablen-Typen verwendet werden. Außerdem nutzt der phpDocumentor <https://www.phpdoc.org/> die Kommentarblöcke zur Generierung einer technischen Dokumentation.

Ein doc-Kommentar in PHP muss mit `/**` beginnen und mit `*/` enden.
Annotationen in anderen Kommentaren werden ignoriert.

PHPUnit verwendet Annotationen zur Laufzeit. Wenn Sie mit Exceptions arbeiten, sollten Sie die Annotation `@expectedException` kennen.

Todo Wir nutzen die `@expectedException`-Annotation im PHPDoc-Block: Als Alternative zur Annotation können wir auch die Methode `setExpectedException()` nutzen:

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

Das erste Testbeispiel verbessern

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

Falls Sie etwas an der Konfiguration ändern, müssen Sie den Befehl `run codecept build` ausführen!

Bei Problemen ist oft hilfreich sind weitere Informationen mit dem Parameter `--debug` anzeigen zu lassen. Zum Beispiel

```
./tests/codeception/vendor/bin/codecept run unit --debug
```

Data Provider

Wir haben im Kapitel Der erste selbst programmierte Test auf Seite 62 den ersten Test erstellt. Mit dem Test haben wir sicherstellt, dass die Variablenbelegung in der Methode `onUserAuthenticate()` der Klasse `PlgAuthenticationLdap` im Falle von einer leeren Eingabe, richtig erfolgt. Die gleiche Variablenbelegung sollte als Ergebnis herauskommen, wenn das Passwortfeld leer bleibt aber ein ungültiger Benutzer eingegeben wird. Und auch dann, wenn das Passwortfeld leer bleibt aber ein Benutzername der in der Datenbank gespeichert ist, eingegeben wird. Erinnern Sie sich, diese Testfälle hatten wir in einer Tabelle im Kapitel Fehler: Referenz nicht gefunden ab Seite Fehler: Referenz nicht gefunden herausgearbeitet.

Es wäre langweilig, für jeden dieser Fälle eine eigene Testmethode zu schreiben. Sehen wir uns lieber an, wie wir die Testmethode mithilfe von *Data Providern* automatisch mit Daten füttern können.

Das nachfolgende Programmcodebeispiel enthält den abgeänderten Programmcode. Sie sehen hier einen Data Provider der in der Methode `provider_credentials_emptypassword` ausgegeben wird. Ein Data Provider ist ein

mehrdimensionales Array. Die zweite Ebene diese Arrays ruft die Methode, mit der der Data Provider über eine Annotation verknüpft ist, auf. Dabei ist der Wert in der zweiten Ebene gleichzeitig der Eingabeparameter der Methode.

```
<?php
namespace suites\plugins\authentication\joomla;
class PlgAuthenticationJoomlaTest extends \Codeception\Test\Unit
{
    protected $class;
    protected function _before() {}
    protected function _after() {}
}
/**
 * @dataProvider provider_credentials_emptypassword
 */
public function testonUserAuthenticate_EmptyPassword($credentials)
{
    require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
    $subject = \JeventDispatcher::getInstance();
    $config = array(
        'name' => 'joomla',
        'type' => 'authentication',
        'params' => new \JRegistry
    );
    require_once JINSTALL_PATH . '/plugins/authentication/joomla/joomla.php';
    $this->class = new \PlgAuthenticationJoomla($subject, $config);
    $options = array(
        'remember' => '',
        'return' => '',
        'entry_url' => '',
        'action' => ''
    );
    $response = new \JAuthenticationResponse;
    $this->class->onUserAuthenticate($credentials, $options, $response);
    $this->assertEquals($response->status, \JAuthentication::STATUS_FAILURE);
    $this->assertEquals($response->type, 'Joomla');
    $this->assertEquals($response->error_message,
        \Jtext::_('JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED'));
}
```

```

public function provider_credentials_emptypassword()
{
    return [
        [array('username' => "", 'password' => "", 'secretkey' => "")],
        [array('username' => "admin", 'password' => "", 'secretkey' => "")],
        [array('username' => "UserNotInDatabase", 'password' => "", 'secretkey' =>
        "")]
    ];
}
}

```

Was haben wir genau geändert? Zunächst haben wir die Methode `provider_credentials_emptypassword()` eingefügt. Diese Methode gibt einen Array zurück. Als nächstes haben wir die Annotation `@dataProvider provider_credentials` über unsere Testmethode gesetzt und dieser Methode einen Eingabeparameter hinzugefügt. Durch die Annotation `@dataProvider provider_credentials` wird dem Eingabeparameter der Rückgabewert der Methode `provider_credentials` zugeordnet.

Starten wir nun den Test über den Befehl

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

```
tests/codeception/unit/suites/plugins/authentication/joomla/PlgAuthenticationJoomlaTest
```

erneut, sieht die Ausgabe folgendermaßen aus:

```

Unit Tests (3) -----
✓ PlgAuthenticationJoomlaTest: Teston user authenticate_empty password | #0 (0.01s)
✓ PlgAuthenticationJoomlaTest: Teston user authenticate_empty password | #1 (0.00s)
✓ PlgAuthenticationJoomlaTest: Teston user authenticate_empty password | #2 (0.00s)
-----
Time: 2.76 seconds, Memory: 10.00MB
OK (3 tests, 9 assertions)

```

Die Verwendung von Data Providern ermöglichte es uns, eine Testmethode mit unterschiedlichen Daten aufzurufen. Wir können mit Data Providern komplexe Testfälle flexibel auf unterschiedliche Situationen anpassen.

Fixtures

Eine Fixtur setzen wir ein, wenn wir in einer gut bekannten und festen Umgebung testen möchten. Nur so ist ein Test übrigens auch wirklich wiederholbar. Anderenfalls können immer Situationen eintreten, die man nicht vorhersehen kann. Fixturen unterstützen Sie dabei vor einem Test bestimmte wohldefiniert 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 `onUserAuthenticate()` testen können, ohne sich darauf verlassen zu müssen, dass andere Werte vorher in durch andere Klassen in der Programmausführung richtig gesetzt wurden. Beispielweise die Variablen `$config` und `$options`. Das Setzen dieser Werte ist auch nicht Bestandteil dieses Tests. Wir testen hier ausschließlich die Methode `onUserAuthenticate()` der Klasse `PlgAuthenticationLdap`!

Eine Fixtur ist eine Variable die mit einer bestimmten Belegung gespeichert wird. Mit dieser Belegung kann sie mehrmals im Testablauf abgerufen werden. In der Regel wird diese Fixtur in einer Methode gespeichert, die vor jedem Test in einer Testdatei automatisch abläuft. Dazu können Sie im nächsten Abschnitt mehr lesen. Sie können Fixturen auch in einer bootstrap Datei vor allen Testdateien zusammen setzen. (Todo schöner formulieren.)

Beispiele für Fixturen:

- Das Laden einer Datenbank mit einem bestimmten, bekannten Datensatz. (Todo wir machend das beim Initialisieren.)
- Das Kopieren eines bestimmten bekannten Satzes von Daten
- Vorbereitung von Eingabedaten. Im weiteren Sinne können dies Stubs und Mocks sein.

Todo Verweis. Codeception unterscheidet grenzt diese aber voneinander ab.

(ToDo abgrenzung zu Stub Mock)

Als nächstes ändern wir unser Beispiel so ab, dass wir für die Optionen und die Konfiguration eine Fixtur nutzen.

```
<?php
namespace suites\plugins\authentication\joomla;
use \Codeception\Util\Fixtures;
class PlgAuthenticationJoomlaTest extends \Codeception\Test\Unit
```

```

{
    protected $class;
    protected function _before(){}
    protected function _after(){}
}
/**
 * @dataProvider provider_credentials_emptypassword
 */
public function testonUserAuthenticate_EmptyPassword($credentials)
{
    require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
    $subject = \JeventDispatcher::getInstance();
    Fixtures::add('config', [
        'name' => 'joomla',
        'type' => 'authentication',
        'params' => new \Jregistry
    ]);
    require_once JINSTALL_PATH . '/plugins/authentication/joomla/joomla.php';
    $this->class = new \PlgAuthenticationJoomla($subject, Fixtures::get('config'));
    Fixtures::add('options', [
        'remember' => "",
        'return' => "",
        'entry_url' => "",
        'action' => ""
    ]);
    $response = new \JAuthenticationResponse;
    $this->class->onUserAuthenticate($credentials, Fixtures::get('options'), $response);
    $this->assertEquals($response->status, \JAuthentication::STATUS_FAILURE);
    $this->assertEquals($response->type, 'Joomla');
    $this->assertEquals($response->error_message,
        \Jtext::_('JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED'));
}

public function provider_credentials_emptypassword()
{
    return [
        [array('username' => "", 'password' => "", 'secretkey' => "")],
        [array('username' => "admin", 'password' => "", 'secretkey' => "")],
        [array('username' => "UserNotInDatabase", 'password' => "", 'secretkey' => "")]
    ];
}

```

```
| }  
| }
```

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

Der Vorteil dieser Änderungen ist so noch nicht offensichtlich. Bisher benötigen diese Fixtur nur an einer Stelle. Wenn wir aber im weiteren Verlauf immer mal wieder die Konfiguration oder die Optionen mit dieser Belegung benötigen wird der Vorteil auch in der Praxis klar. (todo formulierung)

Sie können eine Fixture überschreiben, indem sie diese erneut speichern.

Haben Sie beispielsweise die Anweisung

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

in einer Testdatei und in der nächsten Testdatei setzten Sie

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

dann erhalten Sie über

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

den Benutzer **Paul**.

Allgemein gilt, dass alle Testfälle einer Testklasse von den gemeinsamen Fixturen Gebrauch machen sollten. Hat eine Testmethode für die Fixture-Objekte, dann sollten Sie prüfen, ob die Methode nicht besser in eine andere Testklasse passen würde. Oft ist dies ein Indiz dafür. Es kann durchaus vorkommen, dass zu einer Klasse mehrere korrespondierende Testfallklassen existieren. Jede von diesen besitzt ihre individuellen Fixturen.

Sie sehen schon. Die Verwendung von Fixturen sollte geplant werden. Schon allein deshalb ist es sinnvoll, Fixturen nur an bestimmten Stellen mit Werten zu belegen. Hierzu bieten sich Methoden an, die PHP Unit und Codeception Ihnen zur Planung der Tests zur Verfügung stellen. Zum Beispiel die Methoden, die ich Ihnen im nächsten Abschnitt näher bringen will.

Vor dem Test den Testkontext herstellen und hinterher aufräumen

Vorbereitungen sind mitunter lästig. Auch hinterher das Aufräumen mag niemand. Schön, dass man von PHP Unit und Codeception Methoden an die Hand bekommt, die bei diesen lästigen und oft routinemäßigen Arbeiten unterstützen. Vor und nach jedem Test werden bestimmte Methoden automatisch ausgeführt.

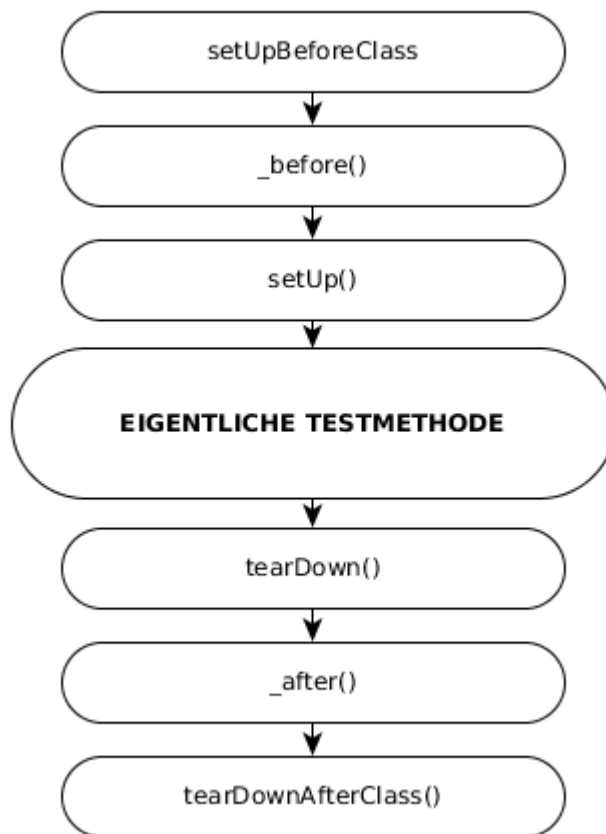


Abbildung 15: kk

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

1. `tests\codeception\unit\plugins\authentication\joomla\PlgAuthenticationJoomlaTest.setUpBeforeClasses()`
2. `tests\codeception\unit\plugins\authentication\joomla\PlgAuthenticationJoomlaTest._before()`
3. `tests\codeception\unit\plugins\authentication\joomla\PlgAuthenticationJoomlaTest.setUp()`
4. `tests\codeception\unit\plugins\authentication\joomla\PlgAuthenticationJoomlaTest.testonUserAuthenticate_EmptyPassword()`
5. `tests\codeception\unit\plugins\authentication\joomla\PlgAuthenticationJoomlaTest.tearDown()`
6. `tests\codeception\unit\plugins\authentication\joomla\PlgAuthenticationJoomlaTest._after()`


```
7. tests\codeception\unit\plugins\authentication\joomla\PlgAuthenticationJoomlaTest.tearDownAfterClass()
```

Das Design von Testfällen 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 Fixture-Variablen nicht im Konstruktor einer Testklasse initialisieren. Nutzen Sie hierfür die dafür vorgesehenen Methoden. Das sind die Methoden, die ich Ihnen eben aufgelistet habe.

`_before()` und `_after()` sind Methoden, die zu Codeception gehören. Alle anderen Methoden sind Standard PHPUnit-Methoden.

Für PHPUnit werden Sie sicherlich am häufigsten die Methoden `setUp()` und `tearDown()`. `setUp()` wird ausgeführt, bevor eine Testmethode aufgerufen wird. `tearDown()` wird aufgerufen, nachdem eine Testmethode ausgeführt wurde. Hier stellen Sie also sicher, dass eine wohldefinierte Umgebung für diesen speziellen Test erstellt wird und nach dem Test auch alles wieder in den Ursprungszustand zurück gesetzt wird. Falls Sie vorher Objekte erstellt haben oder etwas in einer Datenbank gespeichert haben, dann sollten Sie die Objekte und die Datenbankeinträge nach wieder löschen.

Es gibt wenige Gründe Dinge für mehrere Tests gleichzeitig vorzubereiten. Ein Grund könnte aber die Einrichtung der Datenbankverbindung sein, wenn Sie diese für alle Tests in dieser Testklasse benötigen. Natürlich könnten Sie diese in der `setUp()` Methode für jeden Test erstellen und nach jedem Test wieder trennen. Es bietet sich aber an, die Verbindung nur einmal vor allen Test aufzubauen. Während der Tests immer wieder auf diese Verbindung zuzugreifen und nach Abarbeitung aller Testfälle in der Testklasse die Verbindung erst zu trennen. Die Methoden `setUpBeforeClass()` und `tearDownAfterClass` sind die Methoden für diese Aufgabe. Sie bilden die äußersten Aufrufe. Ausgeführt werden Sie bevor die Testklasse instantiiert wird.

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 der Benutzer ein Passwort bei der Anmeldung eingibt. Bisher prüfen wir nur die Eingaben, bei denen das Passwortfeld leer ist. Bereiten wir trotzdem, da wir uns gerade die Methoden die vor und nach jedem Test ablaufen ansehen, schon einmal unser Testumgebung vor. Die Fixturen `$config` und `$options` werden wir in dem Test in unserer Testklasse verwenden. Deshalb verschieben wir die Erstellung der Fixturen in die `_before()` Methode. So können wir, auch wenn wir mehrere Testfälle oder Testmethoden nutzen, immer auf diese

Fixtures zurückgreifen. Für die die Methode setUpBeforeClass() haben wir keine Verwendung (todo wie wir Datenbank initialisieren.)

```
<?php
namespace suites\plugins\authentication\joomla;
use \Codeception\Util\Fixtures;
class PlgAuthenticationJoomlaTest extends \Codeception\Test\Unit
{
    protected $class;
    protected $response;
    protected function _before(){
        Fixtures::add('config', [
            'name' => 'joomla',
            'type' => 'authentication',
            'params' => new \Jregistry
        ]);
        Fixtures::add('options', [
            'remember' => '',
            'return' => '',
            'entry_url' => '',
            'action' => ''
        ]);
        require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
        $subject = \JeventDispatcher::getInstance();
        require_once JINSTALL_PATH . '/plugins/authentication/joomla/joomla.php';
        $this->class = new \PlgAuthenticationJoomla($subject, Fixtures::get('config'));
        $this->response = new \JauthenticationResponse;
    }
    protected function _after() {}
    /**
     * @dataProvider provider_credentials_emptypassword
     */
    public function testonUserAuthenticate_EmptyPassword($credentials)
    {
        $this->class->onUserAuthenticate($credentials, Fixtures::get('options'), $this->response);
        $this->assertEquals($this->response->
```

```

        $this->assertEquals($this->response->status, \Jauthentication::STATUS_FAILURE);
        $this->assertEquals($this->response->type, 'Joomla');
        $this->assertEquals($this->response->error_message,
        \Jtext::_('JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED'));
    }
    public function provider_credentials_emptypassword()
    {
        return [
            [array('username' => "", 'password' => "", 'secretkey' => "")],
            [array('username' => "admin", 'password' => "", 'secretkey' => "")],
            [array('username' => "UserNotInDatabase", 'password' => "", 'secretkey' => "")]
        ];
    }
}

```

Wir haben eine ganze Menge umgebaut. (Todo Was haben wir geändert)

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

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

Testduplikate

Mit Unit Tests testen Sie eine Funktionseinheit. Wichtig ist, dass dieser Test in Isolation zu anderen Einheiten erfolgt. (Todo Verweis zu Unabhängigkeit Testumgebung Merkmale unittest)

Zahlreiche Klassen lassen sich aber gar nicht einzeln testen, weil ihre Objekte in der Anwendung eng mit anderen Objekten zusammen arbeiten.

Wenn dieses Kapitel gelesen haben, sind Sie in der Lage diese Abhängigkeit mithilfe von Stub-Objekten und Mock-Objekten zu lösen.

Mit den Fixtures im vorausgehenden Kapitel haben wir schon mitwirkende Objekte, die selbst nicht direkt getestet werden, gesehen. Fixtures stehen für triviale *unechte* Implementierungen. Wie wir gesehen haben werden in der Regel vordefinierte Werte zurück gegeben. Fixtures sollen meist, komplexe Abläufe oder Berechnungen in der Anwendung zu ersetzen.

Für Testduplikate gibt unterschiedliche Bezeichnungen. [Martin Fowler](#) unterscheidet die Namen Dummy, Fake, Stub und Mock Objekte. Und dann gibt es den Begriff Fixtures. Für was welcher Begriff steht ist - wenn überhaupt - nur sehr schwammig definiert. Im Grunde genommen geht es aber bei allen Begriffen darum, komplexe Abhängigkeiten mithilfe von Testduplikaten aufzulösen. (Todo genauer)

In diesem Kapitel geht es nun um Stubs und Mocks. Diese sollen *eher reale* Objekte duplizieren.

(Todo: irgendwo schreiben, dass secretkey hier außer Acht gelassen wird.

Externe Abhängigkeiten auflösen

Kommen wir zurück auf unser Beispiel. Bisher haben wir ausschließlich eine Benutzereingabe, bei der das Passwortfeld leer gelassen wird, mit einem Unit Test versehen. Falls dieses Feld gefüllt wird, nutzt die Methode `onUserAuthenticate()` der Klasse `PlgAuthenticationLdap` andere Klassen. Es gibt also Abhängigkeiten, die wir hier auflösen müssen. Insbesondere werden statische Methoden in den Objekten `JFactory` und `JUserHelper` verwendet. Statische Methoden werden von PHPUnit nicht unterstützt. Hierfür müssen wir später eine andere Lösung für uns finden.

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

Fangen aber zuerst einmal mit einem einfachen Beispiel an. Dieses Beispiel soll die Erstellung von Stub Objekten auf einfache Art zeigen soll.

Das erste Stub Objekt

Ich hatte ja schon geschrieben, das Stub Objekte nach der allgemeinen Definition wirklichen Objekten entsprechen. In unserm Beispiel haben wir ein Objekt, also ein Objekt, dass es in unserer Anwendung Joomla! wirklich gibt. Der Einfachheit halber haben ich dieses einfach initialisiert und so als Eingabeparameter genutzt. Ich meine das Objekt `\JAuthenticationResponse`.

Wir könnten dieses auch als Stub Objekt erstellen und so verwenden. Für unser Beispiel bringt dies zwar keine Vorteil. Es zeigt aber, wie Sie ein Stub Objekt erstellen und die Rückgabewerte beeinflussen können.

```
<?php
namespace suites\plugins\authentication\joomla;
use \Codeception\Util\Fixtures;
class PlgAuthenticationJoomlaTest extends \Codeception\Test\Unit
{
    protected $class;
    protected $response;
    protected function _before(){
        Fixtures::add('config', [
            'name' => 'joomla',
            'type' => 'authentication',
            'params' => new \JRegistry
        ]);
        Fixtures::add('options', [
            'remember' => '',
            'return' => '',
            'entry_url' => '',
            'action' => ''
        ]);
        require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
        $subject = \JeventDispatcher::getInstance();
        require_once JINSTALL_PATH . '/plugins/authentication/joomla/joomla.php';
        $this->class = new \PlgAuthenticationJoomla($subject, Fixtures::get('config'));
        $this->response = $this->getMockBuilder(\JAuthenticationResponse::class)
    }
}
```

```

        ->getMock();
    }
    protected function _after() {}
    /**
     * @dataProvider provider_credentials_emptypassword
     */
    public function testonUserAuthenticate_EmptyPassword($credentials)
    {
        $this->class->onUserAuthenticate($credentials, Fixtures::get('options'), $this->response);
        $this->assertEquals($this->response->
        $this->assertEquals($this->response->status, \Jauthentication::STATUS_FAILURE);
        $this->assertEquals($this->response->type, 'Joomla');
        $this->assertEquals($this->response->error_message,
        \Jtext::_('JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED'));
    }
    public function provider_credentials_emptypassword()
    {
        return [
            [array('username' => "", 'password' => "", 'secretkey' => "")],
            [array('username' => "admin", 'password' => "", 'secretkey' => "")],
            [array('username' => "UserNotInDatabase", 'password' => "", 'secretkey' => "")]
        ];
    }
}

```

Was haben wir geändert? Wir haben die Zeile `$this->response = $this->getMockBuilder(\JAuthenticationResponse::class);` anstelle der Zeile `$this->response = new \JAuthenticationResponse;`.

Die Variable `$response` enthält vorher und nachher ein Objekt des Typs `JauthenticationResponse`. Allerdings können Sie dieses Objekt nach dieser Änderung beeinflussen.

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

Falls diese Objekt die Methode `getStatus()` hätte, dann könnte mit folgendem Code bewirken, dass das Objekt immer `true` ausgibt.

```
$this->response ->expects($this->any())
    ->method('getStatus')
    ->with(,username')
    ->willReturn(true);
```

Den Code können Sie so lesen: Immer (`$this->any()`) wenn die Methode `getStatus()` (`method('getStatus')`) mit dem Eingabeparameter `username` `with(,username')` aufgerufen wird, wird der Wert `true` `will(true)` zurückgegeben.

In unserem Beispiel setzten wir aber keinen fixen Rückgabewert. Wir wollen das Objekt hier ja nicht bewusst manipulieren, sondern wollen sicherstellen, dass es durch die Methode richtig belegt wird. Hier ist dies ein etwas missbrauchtes, konstruiertes Beispiel. Das Hauptziel dieses Beispiels ist es, Ihnen die Erstellung eines Stub Objektes zu zeigen.

Weitere Stub Objekten

Wenn wir uns die zu testende Methode weiter ansehen, finden wir hier noch andere Abhängigkeiten bei denen ein bestimmter Rückgabewert sichergestellt werden sollte. Zum Beispiel die Klasse `JFactory`.

(Todo Programmcode vom Plugin mit Markierung.)

Die Methoden die `JFactory` aufruft sind alle statische Methoden. Ich hatte eben schon geschrieben, dass PHPUnit statische Methoden nicht als Stub Objekt oder Mock Objekt unterstützt. Deshalb gehen wir hier einen Umweg.

```
<?php
namespace suites\plugins\authentication\joomla;
use \Codeception\Util\Fixtures;
class PlgAuthenticationJoomlaTest extends \Codeception\Test\Unit
{
    protected $class;
    protected $response;
    protected function _before(){
        Fixtures::add('config', [
            'name' => 'joomla',
```

```

        'type' => 'authentication',
        'params' => new \Jregistry
    ]);
    Fixtures::add('options', [
        'remember' => '',
        'return' => '',
        'entry_url' => '',
        'action' => ''
    ]);
    require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
    $subject = \JeventDispatcher::getInstance();
    require_once JINSTALL_PATH . '/plugins/authentication/joomla/joomla.php';
    $this->class = new \PlgAuthenticationJoomla($subject, Fixtures::get('config'));
    $this->response = $this->getMockBuilder(\JAuthenticationResponse::class)
        ->getMock();

    $this->app = $this->getMockBuilder(\JApplicationCms::class)
        ->getMock();

    $this->app
        ->expects($this->any())
        ->method('isAdmin')
        ->willReturn(1);

    \JFactory::$application = $this->app;
}

protected function _after() {}

/**
 * @dataProvider provider_credentials_emptypassword
 */
public function testonUserAuthenticate_EmptyPassword($credentials)
{
    $this->class->onUserAuthenticate($credentials, Fixtures::get('options'), $this->response);
    $this->assertEquals($this->response->
    $this->assertEquals($this->response->status, \Jauthentication::STATUS_FAILURE);
    $this->assertEquals($this->response->type, 'Joomla');
    $this->assertEquals($this->response->error_message,
    \Jtext::_('JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED'));
}

public function provider_credentials_emptypassword()
{

```



```

        return [
            [array('username' => "", 'password' => "", 'secretkey' => "")],
            [array('username' => "admin", 'password' => "", 'secretkey' => "")],
            [array('username' => "UserNotInDatabase", 'password' => "", 'secretkey' => "")]
        ];
    }
}

```

(Todo Was ist geändert)

Mit dieser Änderung können wir nun alle möglichen Eingaben testen. Vorher hat der Aufruf todo eine Ausnahme, also einen Fehler ausgelöst. Da wir die Anwendung nicht auf dem dafür vorgesehen Weg starten, wusste Joomla! nicht, ob wir uns im Administrationsbereich oder im Frontendbereich befinden. Die Antwort auf die Frage haben wir nun fest gesetzt.

Es gibt noch eine ganze Menge mehr Abhängigkeiten, die vorher definiert werden sollten. Aber hier ging es in erste Linie darum aufzuzeigen, wie Sie dies tun können. Sie haben nun die Grundlagen, um Ihre eigenen Anwendung mit Unit Tests zu bestücken.

Ein Mock Objekte

Bevor wir im nächsten Abschnitt den genauen Unterschied zwischen Mock Objekten und Stub Objekten klären, erstellen wir hier nun ein Mock Objekt. Mit einem Testfall für die Klasse PlgContentEmailcloak können wir ein einfaches Beispiel konstruieren. Sie finden den Programmcode zu diesem Plugin in der Datei /plugins/content/emailcloak/emailcloak.php. Sehen wir uns zunächst die Klasse und das was sie tun soll einmal genauer an. (todo)

```

<?php
defined('_JEXEC') or die;
use Joomla\String\StringHelper;
class PlgContentEmailcloak extends JPlugin
{
    public function onContentPrepare($context, &$row, &$params, $page = 0) {
        if ($context == 'com_finder.indexer') {
            return true;
        }
    }
}

```

```

        if (is_object($row)) {
            return $this->_cloak($row->text, $params);
        }
        return $this->_cloak($row, $params);
    }
    ...

```

Um sicherzugehen, dass die Methode `_cloak` ausgeführt wird, wenn nicht indiziert wird, könnten wir folgende Testmethode erstellen.

```

<?php
namespace suites\plugins\content\emailcloak;
class PlgContentEmailcloakTest extends \Codeception\Test\Unit{
protected function _before() {}
protected function _after() {}
public function testOnContentPrepare_RunCloak(){
    require_once JINSTALL_PATH . '/plugins/content/emailcloak/emailcloak.php';
    $emailcloak = $this->getMockBuilder(\PlgContentEmailcloak::class)
        ->disableOriginalConstructor()
        ->setMethods(['_cloak'])
        ->getMock();
    $emailcloak->expects($this->$emailcloak->expects($this->once()))
        ->method('_cloak');
    $emailcloak->onContentPrepare('com_finder.indexer', $row, $params);
}
}

```

Führen Sie diesen Test nun mit dem Befehl `tests/codeception/vendor/bin/codecept run unit /suites/plugins/content/PlgContentEmailcloakTest.php` aus. Er ist erfolgreich!

```

Unit Tests (1) -----
✓ PlgContentEmailcloakTest: On content prepare_run cloak (0.02s)
-----
Time: 3.17 seconds, Memory: 10.00MB

```

Wie Sie sehen müssen Sie hier keine assert-Methode verwenden. Die Zeile `$emailcloak->expects($this->$emailcloak->expects($this->once())->method('_cloak'))`; sorgt dafür, dass der Test nur erfolgreich ist, wenn die Methode `_cloak` auch ausgeführt wurde.

Probieren Sie es aus. Setzen Sie den Wert für `indexer` ein (todo).

```
<?php
namespace suites\plugins\content\emailcloak;
class PlgContentEmailcloakTest extends \Codeception\Test\Unit{
protected function _before() {}
protected function _after() {}
public function testOnContentPrepare_RunCloak(){
require_once JINSTALL_PATH . '/plugins/content/emailcloak/emailcloak.php';
$emailcloak = $this->getMockBuilder(\PlgContentEmailcloak::class)
    ->disableOriginalConstructor()
    ->setMethods(['_cloak'])
    ->getMock();
$emailcloak->expects($this->$emailcloak->expects($this->once())
    ->method('_cloak'));
$emailcloak->onContentPrepare('com_finder.indexer', $row, $params);
}
}
```

Nun meldet die Konsole Ihnen folgendes:

```
Unit Tests (1) -----
✘ PlgContentEmailcloakTest: On content prepare_run cloak (0.02s)
-----
Time: 3.22 seconds, Memory: 10.00MB
There was 1 failure:
-----
1) PlgContentEmailcloakTest: On content prepare_run cloak
Test
tests/codeception/unit/suites/plugins/content/PlgContentEmailcloakTest.php:testOnContentPrepare_RunCloak
Expectation failed for method name is equal to <string:_cloak> when invoked 1 time(s).
```

Method was expected to be called 1 times, actually called 0 times.

FAILURES!

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

Wie unterscheiden sich Mock Objekte von Stub Objekten

Nun haben Sie die unterschiedlichen Testduplikate einmal praktisch erlebt. Das Fixtures schwammig definiert keine realen Arbeitsobjekte darstellen und sie sich so von Mocks und Stubs unterscheiden, habe ich auch schon erwähnt. Sie fragen sich aber sicher nun was der Unterschied zwischen Mock und Stub Objekten ist.

In einem Satz kann ich es so beschreiben: Stubs prüfen den **Status** eines Objektes und Mock das **Verhalten**.

Jeder Testfall prüft eine Vermutung haben wir todo festgelegt. Für diesen Testfall können wir mehrere Stubs benötigen. In der Regel benötigen wir aber nur ein Mock Objekt.

Testlebenszyklus eines Stub Objektes

1. Bereiten Sie das zu prüfende Objekt und seine Stubs Objekte vor. In der Regel sollte dies in der Setup-Methode erfolgen.
2. Funktionalität testen.
3. Zustand prüfen/bestätigen
4. Aufräumen Ressourcen. Dies erfolgt in der Regel in der Teardown Methode.

Testlebenszyklus eines Mock Objektes

1. Bereiten Sie das zu prüfende Objekt. In der Regel sollte dies in der Setup-Methode erfolgen.
2. Bereiten Sie das Objekt vor, das im Zusammenspiel mit dem Testobjekt ein bestimmtes Verhalten zeigen soll.
3. Funktionalität testen.
4. Stellen Sie sicher, dass das erwartete Verhalten eingetreten ist.
5. Zustand prüfen/bestätigen
6. Aufräumen Ressourcen. Dies erfolgt in der Regel in der Teardown Methode.

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

BDD Spezifikationen

Bevor ich nun den Unit Test Teil beende, möchte ich Ihnen gerne noch ein nettes Feature zeigen. (Todo Verweis BDD)

Codeception möchte Sie dabei unterstützen Ihre Tests modular und flexibel aufzubauen. Bei den kurzen Beispielen hier ist die Notwendigkeit dazu noch nicht offensichtlich. Sehen sie sich den Ordner mit den vorhandenen Joomla! Unit Tests einmal an. Obwohl hier die Testabdeckung noch nicht sehr hoch ist, sind diese schon zahlreich. (todo erklären das unittest zu joomla noch alt sind.)

Was meinen Sie was passiert, wenn eine Änderung in der Anwendung Fehler bei der Ausführung der Test auslöst? Vielleicht sogar in einem Test der vor mehreren Jahren von jemandem geschrieben wurde, der heute nicht mehr im Projekt mitarbeitet? Ich denke es wird klar war ich meine.

Wichtig ist, dass es klare Regeln gibt. Jeder im Team sollte wissen, wie er einen Test schreiben soll. Diese Regel sollten präzise und einfach sein. Es bringt nichts, wenn ein Entwickler dabei ist, der zeigen will wie gut er programmieren. Auch dann nicht, wenn seine Beiträge wirklich gut sind. Wenn andere im Team diese guten Ansätze nicht verstehen wird der gute Code am Ende Wegwerfcode sein und im Team wird es viel Frustration geben.

Wirklich gut ist Programmcode meiner meiner Meinung nach erst, wenn andere diesen schnell nachvollziehen können ohne viel nachfragen zu müssen.

Codeception Werkzeuge

Todo Wie installieren

Specify

todoWegen trait muss ich use in class, Data provider muss example heißen, Nachteil mehr speicher

<https://github.com/Codeception/Specify>

```
<?php
namespace suites\plugins\authentication\joomla;
use \Codeception\Util\Fixtures;
class PlgAuthenticationJoomlaTest extends \Codeception\Test\Unit
```

```

{
use \Codeception\Specify;
protected $class;
protected $response;
protected $app;
protected function _before()
{
    Fixtures::add('config', [
        'name' => 'joomla',
        'type' => 'authentication',
        'params' => new \Jregistry
    ]);
    Fixtures::add('options', [
        'remember' => "",
        'return' => "",
        'entry_url' => "",
        'action' => ""
    ]);
    require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
    $subject = \JEventDispatcher::getInstance();

    require_once JINSTALL_PATH . '/plugins/authentication/joomla/joomla.php';

    $this->class = new \PlgAuthenticationJoomla($subject, Fixtures::get('config'));

    $this->response = $this->getMockBuilder(\JAuthenticationResponse::class)

        ->getMock();

    $this->app = $this->getMockBuilder(\JApplicationCms::class)

        ->getMock();

```

```

        $this->app->expects($this->any()
            ->method('isAdmin')
            ->willReturn(1);
        \JFactory::$Application = $this->app;
    }
    protected function _after() {}
    public function testAuthenticationPlugin()
    {
        $this->specify(
            "User leaves the password field blank",
            function($credentials) {
                $this->class->onUserAuthenticate($credentials, Fixtures::get('options'), $this->
                "User leaves the password field blank", function($credentials) {
                    $this->class->onUserAuthenticate($credentials, Fixtures::get('options'), $this->
                >response);

                $this->assertEquals($this->response->status,
                \JAuthentication::STATUS_FAILURE);

                $this->assertEquals($this->response->type, 'Joomla');

                $this->assertEquals($this->response->error_message,
                \JText::_('JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED'));

            }, ['examples' => [

                [array('username' => "", 'password' => "", 'secretkey' => "")],

                [array('username' => "admin", 'password' => "", 'secretkey' => "")],

                [array('username' => "UserNotInDatabase", 'password' => "", 'secretkey' =>
                "")]

            ]]);
    }
}

```

```
$this->specify("User enters correct login data", function($credentials) {

    $this->class->onUserAuthenticate($credentials, Fixtures::get('options'), $this->response);

    $this->assertEquals($this->response->status,
\JAuthentication::STATUS_SUCCESS);

    $this->assertEquals($this->response->type, 'Joomla');

    $this->assertEquals($this->response->error_message, "");

}, ['examples' => [

    [array('username' => "admin", 'password' => "admin", 'secretkey' => "")]

]]);

$this->specify("User enters an invalid user name", function($credentials) {

    $this->class->onUserAuthenticate($credentials, Fixtures::get('options'), $this->response);

    $this->assertEquals($this->response->status,
\JAuthentication::STATUS_FAILURE);

    $this->assertEquals($this->response->type, 'Joomla');
```



```

        $this->assertEquals($this->response->error_message,
        \JText::_('JGLOBAL_AUTH_NO_USER'));

        }, ['examples' => [

            [array('username' => "", 'password' => "admin", 'secretkey' => "")],

            [array('username' => "", 'password' => "PasswordNotInDatabase", 'secretkey' =>
            "")],

            [array('username' => "UserNotInDatabase", 'password' => "admin", 'secretkey' =>
            "")],

            [array('username' => "UserNotInDatabase", 'password' =>
            "PasswordNotInDatabase", 'secretkey' => "")],

        ]]);

        $this->specify("User enters a valid user name but a wrong password", function($credentials)
        {

            $this->class->onUserAuthenticate($credentials, Fixtures::get('options'), $this-
            >response);

            $this->assertEquals($this->response->status,
            \JAuthentication::STATUS_FAILURE);

            $this->assertEquals($this->response->type, 'Joomla');

```

```

        $this->assertEquals($this->response->error_message,
        \Text::_('JGLOBAL_AUTH_INVALID_PASS'));

        }, ['examples' => [

                [array('username' => "admin", 'password' => "PasswordNotInDatabase",
'secretkey' => "")]

            ]]);
    }
}

```

```

Unit Tests (1) -----
✓ PlgAuthenticationJoomlaTest: Authentication plugin (0.39s)
-----

Time: 3.55 seconds, Memory: 16.00MB
OK (1 test, 27 assertions)

```

Verify

<https://github.com/Codeception/Verify>

`$this->assertEquals` mit `verify` ersetzen macht lesbarer und kürzer.

```

        $this->specify("User enters a valid user name but a wrong password", function($credentials)
        {

                $this->class->onUserAuthenticate($credentials, Fixtures::get('options'), $this-
>response);

```

```
verify($this->response->status, \JAuthentication::STATUS_FAILURE);

verify($this->response->type, 'Joomla');

verify($this->response->error_message,
\JText::_('JGLOBAL_AUTH_INVALID_PASS'));

}, ['examples' => [

    [array('username' => "admin", 'password' => "PasswordNotInDatabase",
'secretkey' => "")]

    ]]);
```

In diesem Kapiteln ...

... haben wir Mocks und Stubs und ... Todo dies in jedes Kapitel und auch über all ein vorwort.

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

Funktionstest

In diesem Kapitel werden wir uns Funktionstest genauer ansehen. Im letzten Kapitel haben wir das Authentifizierens-Plugin von Joomla! als einzelne Einheit getestet. Der Test war unabhängig von anderen Programmcodeanteilen. In diesem Kapitel werden wir das Zusammenspiel von mehreren Einheiten testen. Todo REST

Funktionstest in Joomla

Momentan setzt Joomla! keine Funktionstest ein. Bei den Unittests (todo Unittest überall zusammen geschrieben?) hatte ich schon erwähnt, dass diese teilweise keine

reinen Unittests sind. Hier wird in manchen Tests modulübergreifend eine Funktion getestet. (todo alte unittests in joomla im vorherigen Kapitel erklären).

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

Wie Sie wissen besteht unsere Website bisher nur aus einer einzigen Unterseite mit integriertem Anmeldeformular.

Fertige Tests beispielhaft ansehen

Todo

Wir erstellen den ersten Funktionstest

Generieren

```
/var/www/html/gsoc16_browser-automated-tests$ tests/codeception/vendor/bin/codecept generate:cept functional FrontendAnmeldung
```

```
Test was created in /var/www/html/gsoc16_browser-automated-  
tests/tests/codeception/functional/FrontendAnmeldungCept.php
```

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

Ich habe hier den Text „perform actions and see result“ in „Ich will sicherstellen, dass das Loginformular der Website korrekt funktioniert“ geändert.

```
<?php  
$I = new FunctionalTester($scenario);  
$I->wantTo('Ich will sicherstellen, dass das Loginformular der Website korrekt funktioniert');
```

Funktionstests in Codeception sind einfach zu verstehen.

Die Zeile `$I = new FunctionalTester($scenario);` instantiiert den Akteur. Also den fiktiven Tester. Dieser heißt hier `FunctionalTester`. Sie können dem Tester auch einen anderen Namen geben. Den Namen legen Sie in der Konfigurationsdatei fest. Die Konfigurationsdatei finden Sie im Verzeichnis

`Webserverroot/tests/codeception/_support/` (todo webserverroot) Sie heißt `FunctionalTester`. Die erste Zeile in der Datei enthält den Namen der Klasse, die den

Tester bildet. Standardmäßig steht hier `class_name: FunctionalTester`. Die Klasse wird erstellt, wenn Sie den Befehl `codeception build` ausführen.

```
tests/codeception/vendor/bin/codecept build
```

```
/var/www/html/gsoc16_browser-automated-
```

```
tests/codeception/_support/_generated/FunctionalTesterActions.php
```

wird erstellt.

Die zweite Zeile ist optional. Hier wird das Ziel des Tests beschrieben. Dieser Methodenaufruf hilft Projektmitarbeitern, die keine Programmierer sind, den Testablauf und was dabei fehlgeschlagen ist, zu verstehen.

Die Methode `wantto()` sollte nur einmal aufgerufen werden. Ein weiterer Aufruf würde den vorherigen überschreiben. Führen Sie den Test nun aus:

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

```
Codeception PHP Testing Framework v2.2.7
```

```
Powered by PHPUnit 5.7.5 by Sebastian Bergmann and contributors.
```

```
Functional Tests (1)
```

```
-----  
✓ FrontendAnmeldungCept: Ich will sicherstellen, dass das loginformular der website korrekt  
funktioniert (0.00s)
```

```
-----  
Time: 161 ms, Memory: 10.00MB
```

Test erweitern

Nun geben wir den Startpunkt an. Wir wollen das Login Formular testen und das finden wir auf der Startseite unserer Website.

```
$I->seeLink('Forgot your username?');
```

```
$I->click('Forgot your username?');
```

```
$I->see('Please enter the email address associated with your User account. Your username will be  
emailed to the email address on file.');
```

```
$I->expect('erwarte');
```

```
$I->expectTo('erwarte');
```

```
$I->amGoingTo('erwarte');
```

```
$I->$I->seeElement('#modlgn-passwd');
```

```

$I->fillField('#modlgn-username', 'admin');
$I->fillField('#modlgn-passwd', 'admin' );
$I->click('button.login-button');
/*
$I->submitForm('#login-form', array('user' => array(
    'username' => 'admin',
    'password' => 'admin'
)));
*/
$I->see('Hi Super User');

```

Todo Fehler in See complete response in '/var/www/html/gsoc16_browser-automated-tests/tests/codeception/_output/' directory]

Todo Xpath tutorial

Todo Firefox instpektor.

Wiederverwendbare Tests

Wenn Sie das letzte Kapitel durchgearbeitet haben können Sie sich nun sicher gut vorstellen, dass das Erstellen von Tests nicht die spannendste Tätigkeit ist.

Insbesondere dann, wenn es um Formulare geht, die viele Felder enthalten und sich vielleicht sogar über mehrere Seiten erstrecken. Und weil wir uns das Leben so einfach wie machen möchten, werden wir Tests so zu schreiben, dass wir sie an anderen Stellen wieder verwenden können. Wie machen wir das am besten?

Das Login-Formular ist ein gutes Beispiel. Höchstwahrscheinlich werden Sie es vor fast jedem Test ausfüllen müssen. Das Formular befindet sich auf der Startseite Ihrer Website.

Akteure

/var/www/html/gsoc16_browser-automated-tests/tests/codeception/_support/FunctionalTester.php

```

<?php
class FunctionalTester extends \Codeception\Actor
{
    use _generated\FunctionalTesterActions;
}
/**

```

```

    * Define custom actions here
    */

public function login()
{
    $I = $this;
    $->fillField('#modlgn-username', 'admin');
    $->fillField('#modlgn-passwd', 'admin' );
    $I->click('button.login-button');
}
}

$I->seeLink('Forgot your username?');
$I->click('Forgot your username?');
$I->see('Please enter the email address associated with your User account. Your username will be
emailed to the email address on file. ');
$I->expect('erwarte');
$I->expectTo('erwarte');
$I->amGoingTo('erwarte');
$I->$I->seeElement('#modlgn-passwd');
/*$I->fillField('#modlgn-username', 'admin');
$I->fillField('#modlgn-passwd', 'admin' );
$I->click('button.login-button');*/
$I->login();
/*
$I->submitForm('#login-form', array('user' => array(
    'username' => 'admin',
    'password' => 'admin'
)));
*/
$I->
/*
$I->submitForm('#login-form', array('user' => array(
    'username' => 'admin',
    'password' => 'admin'
)));
*/
$I->see('Hi Super User');

```

Step Objekte

Step Objekte kommen ins Spiel, wenn Sie möchten für eine bestimmte Gruppe spezielle Tests schreiben. Zum Beispiel macht es Sinn Tests im Frontend und im Backend zu separieren. Mit Codeception tun wir dies in eigenen Klassen aus denen wir Step Objekte erstellen.

Mit dem Generator erstellen.

```
/var/www/html/gsoc16_browser-automated-tests$ tests/codeception/vendor/bin/codecept
generate:stepobject functional Frontend
Add action to StepObject class (ENTER to exit): login
Add action to StepObject class (ENTER to exit):
StepObject was created in /var/www/html/gsoc16_browser-automated-
tests/tests/codeception/_support/Step/Functional/Frontend.php
```

Und das Step Objekt sieht dann so aus.

```
Todo sie können auch Unterordner anlegen. Beispiel kommt später in
acceptance.
tests/codeception/vendor/bin/codecept generate:stepobject functional Frontend/Frontendlogin
erzeugt.
<?php
namespace Step\Functional\Frontend;
class Frontendlogin extends \FunctionalTester
{
    public function login()
    {
        $I = $this;
    }
}
```


Page Objekte

Wenn Sie einen Funktionstest schreiben, und auch bei den Akzeptanztests die ich Ihnen im nächsten Kapitel vorstellen werden, gibt es nicht nur gemeinsame Aktionen. Sinnvoll ist es auch Elemente des HTML-Dokumentes wiederverwenden. Für diese Fälle müssen wir das Page Objekt Muster implementieren. Das Page Objekt repräsentiert eine Webseite als Klasse und die DOM-Elemente auf dieser Seite als ihre Eigenschaften und einige grundlegende Interaktionen als ihre Methoden. Page Objekte sind sehr wichtig, wenn Sie eine flexible Architektur Ihrer Tests entwickeln. Codieren Sie nicht komplexe CSS- oder XPath-Locators in Ihren Tests, sondern verschieben Sie sie in PageObject-Klassen. So müssen Sie nur an einer Stelle einen Eintrag ändern, wenn Sie ein Element auf der Website ändern. (Todo schöner formulieren)

der Befehl `tests/codeception/vendor/bin/codecept generate:pageobject functional Frontend` erstellt ein Page Objekt im Verzeichnis `/var/www/html/gsoc16_browser-automated-tests/tests/codeception/_support/Page/Functional/Frontend.php`

Das Page Objekt sieht so aus:

```
<?php
namespace Page\Functional;
class Frontend
{
    public static $URL = 'http://localhost/joomla-cms';
    public static $moduleUsername = '#modlgn-username';
    public static $modulePassword = '#modlgn-passwd';
    public static $moduleLoginButton = 'button.login-button';
    public static function route($param)
    {
        return static::$URL.$param;
    }
    protected $functionalTester;
    public function __construct(\FunctionalTester $I)
    {
        $this->functionalTester = $I;
    }
}
```

Und so können Sie diese Page dann verwenden

```
<?php
use Page\Functional\Frontend as Frontend;
class FunctionalTester extends \Codeception\Actor
{
    use _generated\FunctionalTesterActions;
    public function login()
    {
        $I = $this;
        $I->fillField(Frontend::$moduleUsername, 'admin');
        $I->fillField(Frontend::$modulePassword, 'admin' );
        $I->click(Frontend::$moduleLoginButton);
    }
}
```

Was Sie bei Funktionstests beachten müssen

Todo pitfalls umschreiben

Akzeptanztests sind meist viel langsamer als funktionelle Tests. Aber funktionale Tests sind weniger stabil, da Codeception und die Anwendung in ein und derselben Umgebung laufen. Wenn Ihre Anwendung nicht für langlebige Prozesse ausgelegt ist, zum Beispiel den Exit-Operator oder globale Variablen, dann funktionieren wahrscheinlich funktionelle Tests nicht für Sie.

Header, Cookies, Sitzungen

Eines der häufigsten Probleme mit Funktionstests ist die Verwendung von PHP-Funktionen, die sich mit Headern, Sitzungen und Cookies. Wie Sie vielleicht schon wissen, löst die Header-Funktion einen Fehler aus, wenn er ausgeführt wird, nachdem PHP bereits etwas ausgegeben hat. In funktionalen Tests führen wir die Anwendung mehrere Male, so erhalten wir viele irrelevante Fehler im Ergebnis.

Speichermanagement

In funktionalen Tests, im Gegensatz zu laufen die Anwendung der traditionellen Art und Weise, die PHP-Anwendung nicht aufhören, nachdem sie die Verarbeitung einer

Anfrage abgeschlossen ist. Da alle Anfragen in dem einen Speicherbehälter ausgeführt werden, sind sie nicht isoliert. Also, wenn Sie sehen, dass Ihre Tests mysteriös scheitern, wenn sie nicht sollten - versuchen, einen einzelnen Test durchzuführen. Dies wird zeigen, wenn die Tests fehlgeschlagen waren, weil sie nicht während des Laufs isoliert wurden. Halten Sie Ihren Speicher sauber, vermeiden Sie Speicherlecks und reinigen Sie globale und statische Variablen.

REST-Schnittstelle

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

Todo einföhrung in REST

tests/codeception/vendor/bin/codecept generate:cept functional Benutzerschnittstelle

Test was created in /var/www/html/gsoc16_browser-automated-tests/tests/codeception/functional/BenutzerschnittstelleCept.php

```
<?php
$I = new FunctionalTester($scenario);
$I->wantTo('perform actions and see result');
$I->sendGET('http://localhost/joomla-cms/index.php?option=com_users&view=remind&Itemid=101');
$I->see('Please enter the email address associated with your User account. Your username will be emailed to the email address on file.');
```

Todo http://docs.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern

todo Session and helper

Sie sehen, es gibt viele Möglichkeiten, wiederverwendbare und gut lesbare Tests zu erstellen.

Das 6. Kapitel hat Funktions-Tests zum Thema. Ich grenze Black-Box-Tests von White-Box-Tests ab.

Acceptancetests

Nun haben wir den letzten Teil dieses Titels erreicht. Akzeptanztests. Dies ist Testmethode, die in Joomla! hauptsächlich mit Codeception verwendet wird.

Wie Sie in den vorhergehenden Kapiteln gesehen haben sind Tests in Codeception ähnlich aufgebaut. Hier im Kapitel wird es nichts grundsätzlich neues geben.

Wichtig ist sich hier noch einmal klar zu machen, warum wir testen. Mit Unit Tests wollen wir sicherstellen, dass jede kleine Einheit Eingaben zu korrekten Ausgaben verarbeitet. Funktionstests sollen sicherstellen, dass die diese Einheiten technisch korrekt zusammen arbeiten. Mit Akzeptanztests wollen wir überprüfen ob die Spezifikationen, die ganz zu Beginn des Projektes aufgestellt wurden, erfüllt sind.

Selenium WebDriver – Eine Einführung

Wenn Sie meine Version (todo passt das noch, was schlage ich vor zu installieren?) installiert haben, sehen Sie im Verzeichnis schon einige Akzeptanztest, die im Joomla! Projekt erstellt wurden.

Todo Wir nutzen Joomla Browser PHP Browser ist schneller

Todo link zu installationsanleitung von Selenium

Konfiguration

Im Joomla! Projekt haben wir Webdriver in der Klasse JoomlaBrowser erweitert. JoomlaBrowser bietet weitere Funktionalitäten.

Webdriver

PHP Browser mit Webdriver ersetzen

- url
Ist der Hostname der beim Test verwendet wird.
- Restart: true
Dieser Parameter veranlasst Webdriver dazu

JoomlaBrowser

Todo Weitere Parameter

Akzeptanztests in Joomla!

Fertige Tests beispielhaft ansehen

Todo test starten Browser öffnet sich :)

Wir erstellen den ersten Akzeptanztest

Sie wissen nun wie Akzeptanztests, die Selenium Webdriver nutzen kann, strukturiert sind, können Sie selbst Tests erstellen.

Gherkin

Den Test generieren

Unterschiedliche Browsern und Robo

Die Grenzen von Selenium

Sie sind nun sicherlich begeistert von Selenium. Tests können realitätsnah durchgeführt werden. Der Ablauf ist identisch mit den Benutzereingaben, die ein Mensch tätigen würde. So kann Zeit gespart werden. Außerdem muss kein Mensch wiederholt ein und dasselbe Formular ausfüllen. Selenium Webdriver macht dies anstandslos. Er beschwert sich nicht.

Leider gibt es aber ein paar Dinge, die auch nicht mit Selenium Webdriver sichergestellt werden können. Zum Beispiel alles was mit Design zu tun hat. Kriterien dafür, dass etwas passend und schön aussieht kann man einer Maschine nicht mitgeben. Ob eine Website auf allen Geräten gut lesbar und übersichtlich ist, also ob sie responsive ist, lässt sich auch nicht sicher testen.

Spezielle Effekte (todo Hover)

EXKURS: Welche Qualitätsmerkmale sind testbar

Die **Funktionalität** und die **Zuverlässigkeit** können Sie systematisch und objektiv testen. Die **Benutzbarkeit** und die **Effizienz** ist hingegen teilweise testbar sofern passende Metriken vorliegen. Bei der **Veränderbarkeit** und **Übertragbarkeit** stoßen Softwaretests allerdings an Ihre Grenzen. Diese Merkmale sind nicht testbar.

Funktionalität

Angemessenheit - Vorhandensein und Eignung von Funktionen für

Aufgabenstellung

Richtigkeit - Liefern der richtigen Ergebnisse und Wirkungen

Interoperabilität

Zusammenarbeit mit anderen Systemen

Ordnungsmäßigkeit

Einhaltung anwendungsspezifischer Normen und Gesetze

Sicherheit - Unberechtigten Zugriff auf Programme und Daten verhindern

Zuverlässigkeit

Reife

Häufigkeit des Versagens durch Fehlerzustände

Fehlertoleranz

Beibehaltung des spezifizierten Leistungsniveaus bei

Nicht-Einhaltung spezifizierter Schnittstellen

Software-Fehlern

Wiederherstellbarkeit - Wiederherstellung des Leistungsniveaus nach Versagen;

Zurückgewinnung der betroffenen Daten

Kapitel 7. geht es um Annahmetest oder Acceptance Test. Es wird erläutert wie diese erstellt und mithilfe des Frameworks Selenium (<http://www.seleniumhq.org/>) automatisch im Browser ablaufen.

Analyse

Unit Testing gives you the what.

Test-Driven-Development gives you the when.

Behavior Driven-Development gives you the how.

[Jani Hartikainen]

Das letzte Kapitel schließt mit der Analyse von Tests ab. Wie können diese verbessert werden? Wie misst man die Testabdeckung und gibt diese als Report aus. Ist es sogar möglich den Programmcode mithilfe von Tests zu verbessern? Automation

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

Bucheinband

Viele Programmierer haben ein ungutes Gefühl bestehenden Code zu erweitern oder zu verändern. Nach getaner Arbeit ist es oft so, dass irgendwo im Programm ein Problem auftritt, dass mensch nicht beachtet hat. (Todo Verweis zu menschlichen Fehler Warum dies menschlich ist, habe ich im Kapitel beschrieben. Darauf gehe ich in Kapitel ein.) Wer dann einmal in die Testgetriebene Entwicklung hinein schnuppert macht oft die gute Erfahrung, dass er mit dieser Arbeitsweise nach getaner Arbeit entspannt nach Hause gehen kann.

Gerade in der heutigen schnelllebigen Zeit, in der man Angst vor einem Update hat, weil es oft Probleme danach gibt ist das Thema aktuell wie nie.

Beispiele unter Ubuntu 16.04 und LAMP. Mitmachbuch.

Auch wer nicht entwickelt aber Websites erstellt kann diese mit Akzeptanztest testen. (todo auch im buch bei Akzeptanztests erwähnen)

Testen: Abgrenzung zu anderen Verfahren

Verifikation

Die Verifizierung oder Verifikation ist der theoretische Nachweis, dass ein vermuteter oder behaupteter Sachverhalt wahr ist, also der formaler Korrektheitsbeweis. In der Theorie ist dies die ideale Methode. Ein automatisches Beweisen ist leider nicht möglich und die manuelle Beweisführung ist stupide und fehleranfällig. Nachteilig gegenüber Testen ist, dass nur die formale Korrektheit überprüft wird. Beim Testen wird zusätzlich die Robustheit, die Spezifikation und die Programmumgebung mit einbezogen.

Simulation

Die Simulation ist eine Vorgehensweise zur Analyse von Systemen, die für die theoretische oder formelmäßige Behandlung zu komplex sind. In der Softwareentwicklung wird hierzu ein Modell der Software ausgeführt. Dies ist zu einem frühen Zeitpunkt möglich. Außerdem kann unabhängig von der realen Umgebung getestet werden. Problematisch ist es sicherzustellen, dass die Simulation auch tatsächlich mit der Realität übereinstimmt und der Simulator korrekt abläuft.

Was ist eine Fehlfunktion

Software-Bugs entzaubern

Als Programmierer haben Sie vielleicht auch manchmal das Gefühl, dass Fehler von außen in das Programm eindringen und dass diese zufällig – quasi aus heiterem Himmel – entstehen. Das ist aber eine Fehlwahrnehmung. Stattdessen sind Fehler meist von Anfang an im Programm oder durch spätere Programmcodeänderungen erzeugt worden. Bei Fehlern handelt es sich um menschliche Fehlleistungen des Programmierers.

Bug ganz konkret

Der Begriff Bug ist sehr allgemein. Konkretere Begriffe sind

- Defekt
Nicht korrekter Programmcode (ein Bug im Programmcode)

- Infektion
Nicht korrekter Zustand (ein Bug im Zustand)
- Fehlfunktion
Nicht korrektes Verhalten (ein Bug im Verhalten/Ausgabe)

Ein Entwurfsfehler ist kein Bug im eigentlichen Sinne. Entwurfsfehler entstehen bereits in der Entwurfsphase und sollten in dieser Phase behoben werden. Bleiben sie dabei unentdeckt können sie hohe Folgekosten oder Schäden verursachen.

Andere Namen für Bugs aus psychologischer Sicht betrachtet: Anstelle des Begriffs Bug wird dem Kunden gegenüber gerne der Begriff *Issue* verwendet. Issue klingt dem Kunden gegenüber verniedlichend. Im Gegensatz dazu klingen die Begriffe *Error* oder *Fault* für den verantwortlichen Programmierer belastend.

Was ist überhaupt eine Fehlfunktion?

Eine *Fehlfunktion* ist die Nichterfüllung einer festgelegten Forderung.

Bei der Nichterfüllung einer beabsichtigten oder angemessenen Forderung handelt sich um einen *Mangel*.

Meinungsverschiedenheiten bei der Benutzerfreundlichkeit oder unkonkrete Forderungen wie „Programm zu langsam“ oder „Schachprogramm spielt nicht gut“ gehören zur Grauzone.

Wodurch entsteht falsche Programmierung?

Kommunikationsprobleme

- Unvollständiger Entwurf
- Ungenauer Entwurf
- falsch interpretierter Entwurf

Entwurf verstanden, aber falsch programmiert

Einstufung von Fehlern nach ihrer Schwere

1. Kritisch - Produktionsausfall; Aufgabe nicht mehr erfüllbar
2. Hoch - Produktion / Leistung herabgesetzt
3. Mittel - Verhinderung der vollen Ausnutzung der Möglichkeiten des Programms
4. Niedrig - kosmetische Probleme; Leistung bleibt erhalten
5. Unproblematisch - nicht geforderte Softwareverbesserung

Möglichkeiten um Fehler zu finden

Bei der Planung haben Sie folgende Möglichkeiten eine Fehler zu finden:

- Review (gedankliches Durchspielen)
auch wieder Kommunikationsprobleme
- Simulation - teuer, keine umfassenden Systematiken verfügbar

Im Stadium der Programmierung

- Verifikation Korrektheit beweisen
- idealer Test für n Defekte reichen n Testfälle
- erschöpfender Test alle Eingabemöglichkeiten durchprobiere

Ein Programm sollte nicht durch den Entwickler selbst getestet werden. Nur so kann mit persönliche Distanz getestet und ein Missverständnis beim Interpretieren der Spezifikation gefunden werden. Ein Zielkonflikt wie zum Beispiel das Einhalten von Zeitvorgaben, vermieden werden.