Guten Tag

Bevor ich zuerst zufällig auf Joomla! und dann später über Codeception gestolpert bin, konnte ich mir nicht vorstellen, dass die Hindernisse, die mir beim Testen immer im Wege standen und somit das Leben schwer gemacht haben, tatsächlich von jemanden gelöst wurden.

Ich habe sehr viel Zeit mit dem Testen von Software und vorher noch mehr mit Problemen die aufgrund von fehlenden Tests entstanden sind, verbracht. Ich musste mir alle Informationen an verschieden Stellen zusammen suchen und selbst eine Menge nicht immer schöner Erfahrungen sammeln. Dann habe ich mich entschieden ein Buch zu schreiben welches ich gerne zum Lernen zur Verfügung gehabt hätte.

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 mitbringen

Sie müssen nicht sehr viele Voraussetzungen erfüllen, um dieses Buch zu bearbeiten. Eigentlich müssen Sie nur über heute üblichen Computer verfügen. Auf diesem sollte eine Entwicklungsumgebung und ein lokaler Webserver installiert sein. Mehr dazu finden Sie im Kapitel Praxisteil: Die Testumgebung einrichten.

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 Test für ihre Software schreiben wollten – es dann aber doch aus unterschiedlichen Gründen nie zu Ende gebracht haben. Codeception könnte ein Weg sein, Hindernisse aus dem Weg zu räumen.

Das sollten Sie bei Lesen beachten

Neue Begriffe oder wichtige Wörter sind im Text fett gedruckt.

Programmcode

Ein Buch im Bereich Programmierung enthält oft Programmcode der schwierig darstellbar ist. 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' => 'jooma"',
                'type' => 'authentication',
                'params' => new \JRegistry
        );
```

Kommandozeileneingaben

```
$ 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

> **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;

Wichte Merksätze

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 Buch habe ich bewusst Frameworks, hier 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 praktisch zu veranschaulichen tauchen wir kurz in die Themen 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 nichts Tolles. Das ist langweilig! Außerdem erscheint es auch nicht wichtig, Software zu testen. Schon im Studium war dieser Themenbereich ganz am Schluss eingeordnet und in meinem Fall bliebe dafür keine Zeit mehr. Prüfungsrelevant waren Testmethoden nicht. Demotiviert hat mich zusätzlich die Tatsache, das 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 gehen noch weiter gehen und sage: „Softwareentwicklung ohne Tests ist wie Klettern ohne Seil und Haken“. Probieren Sie es aus. Vielleicht springt der Funkte auch bei Ihnen über, wenn Sie das erste Mal hautnah erlebt haben, das ein Tests Ihnen eine mühsame Fehlersuche erspart hat. Mit einem Sicherheitsnetz von Tests können Sie mit weniger Stress hochwertige Software entwickeln.

(Todo spätere Dinge hier einarbeiten)

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.

Als nächstes stellt sich nun die Frage, wie intensiv und auf welche Art und Weise Test integriert werden sollten. Und dies ist die ideal Überleitung zum Thema Projektmanagement.

Projektmanagement

Das Magische Dreieck beschreibt den Zusammenhang zwischen den **Kosten**, der benötigten **Zeit** und der leistbaren **Qualität**. Ursprünglich wurde dieser Zusammenhang im Projektmanagement erkannt und beschrieben. Sie haben aber sicher 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 verschoben.

Nun kommt die Magie ins Spiel: Wir überwinden den Zusammenhangs aus Zeit, Kosten und Qualität! Auf lange Sicht kann der Zusammenhang aus Kosten, Zeit und Qualität tatsächlich überwunden werden. (Todo den Abschnitt schöner machen)

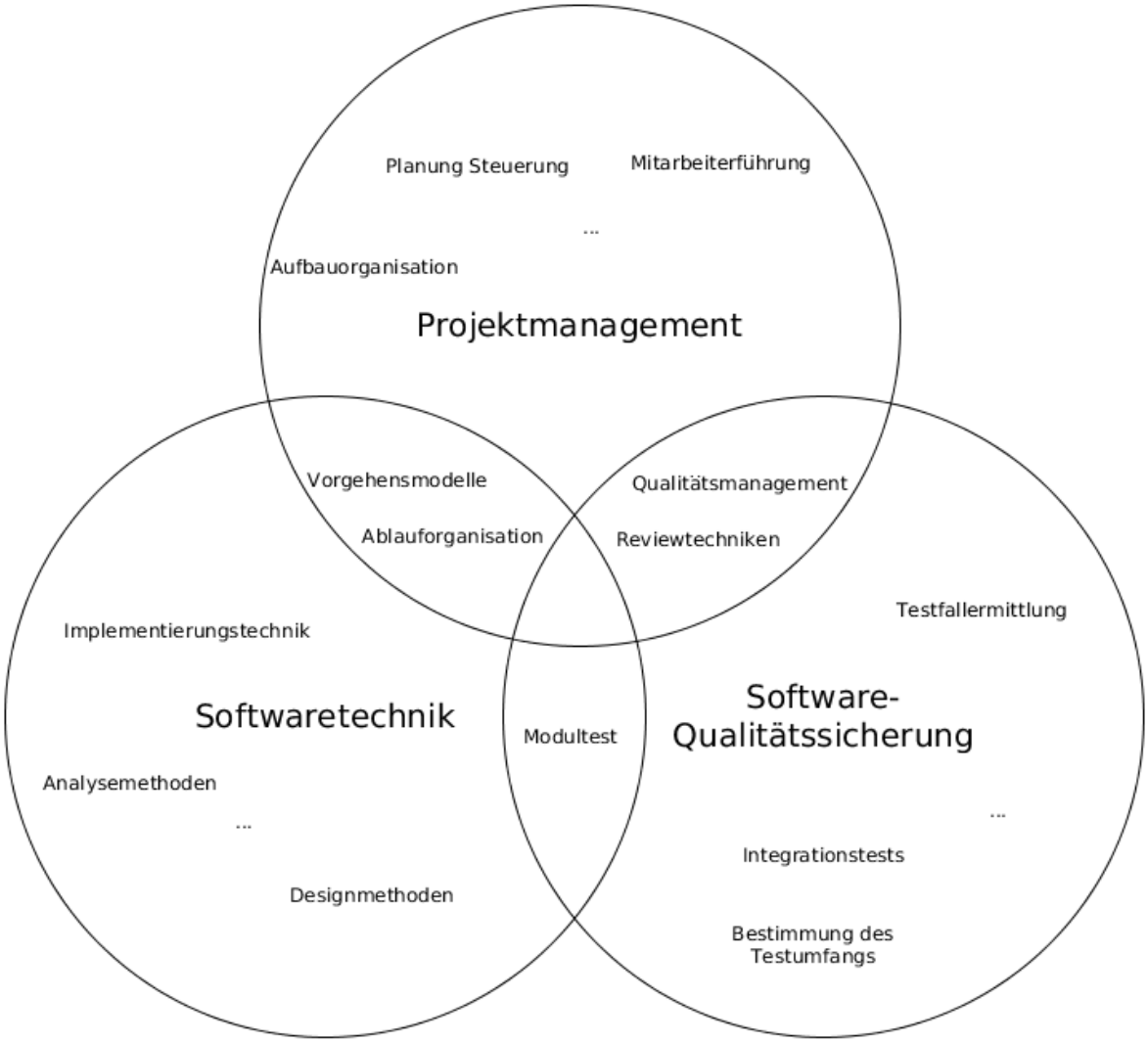
Vielleicht haben auch Sie schon in der Praxis selbst erlebt, dass eine Qualitätssenkung auf lange Sicht keine Kosteneinsparungen zur Folge hat. Die technische Schuld, die dadurch entsteht, führt oft sogar zu Kostenerhöhungen und zeitlichem Mehraufwand.

> 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)



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!

Das 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 ist fast immer Extreme zu vermeiden und einen Mittelweg zu finden.

(Todo erklären wie ich meine wie das geht – Gründe warum Softwareprojekte schlecht planbar sind (wissen von entwicklern) – 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 diese Buches liegt eher 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! Wie tun Sie dies aber nun am besten?

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 schon geschrieben habe)

Kontinuierliches Testes

(Todo einfügen von dem was ich schon geschrieben habe)

Stellen Sie sich folgendes Szenario vor. Der Release-Tag in einem großen Open Source Projekt ist gekommen. Alles das, was die Entwickler des Teams seit dem letzten Release entwickelt haben, wir nun das erste Mal zusammen eingesetzt. Wird alles funktionieren? Werden alle Tests erfolgreich sein, falls das Projekt Test integriert. Oder muss der Release-Tag doch wieder verschoben werden und es stehen nervenaufreibende Stunden der Fehlerbehebung an?

Das oben beschriebene Szenario mag wohl kein Entwickler gerne miterleben. Viel besser ist es doch, jederzeit zu wissen, in welchem Zustand sich das Softwareprojekt befindet? Weiterentwicklungen, die nicht zum bisherigen Bestand passen, sollten erst integriert werden, 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 am Tag des Release. Die Fehlerbehebung ist wesentlich leichter.

(Todo Jenkins Kaitel 9?)

Testgetriebene Entwicklung (TDD)

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. Test 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. Tests helfen Ihnen also dabei, das **Programm richtige** zu schreiben.

(Todo Link Theorie ist immmer schwer vorstellbar. Parktisches Beispiel Kapitel)

BDD

Neben den Funktionstests und Unit Tests erstellen Sie auch Akzeptanztests. Akzeptanztests prüfen ob die Spezifikation, also die Anforderung des Kunden, erfüllte ist. Akzeptanztests helfen Ihnen dabei, das **richtige Programm** zu schreiben.

(Todo passt das mit den Aktzeptanztests und Funktionstests hier? Vielleicht Verweis?)

Planen

Generieren

Sie wissen nun sehr viel Theoretisches zum Thema Softwaretests. In diesem Buch werde ich ihnen einige Werkzeuge erklären, die Sie beim genieren von Tests unterstützen. Wenn Sie diese Tools einsetzten, 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!

Starten

Zusammenfassung

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.

Einführung - Testen, Debugging, Optimierung?

Was ist Testen

In dieser Publikation geht es um das Testen von Software. Wenn Sie etwas Testen tun sie dies sicherlich oft mit der Absicht eine Fehlfunktion zu finden. So ist es auch beim Testen von Software. Es handelt sich um den Prozess, ein Programm mit der Intention auszuführen, Fehlfunktionen zu finden.

Dies ist ein sehr destruktive Sichtweise:

- Ein erfolgreicher *Testfall* ist der, bei dem eine bestimmte Eingabe ein falsches Verhalten erzeugt.
- Ein *nicht* erfolgreicher *Testfall* ist der, bei dem das Programm ein korrektes Verhalten zeigt.

Der gewollte Nebeneffekt des destruktiven Ansatzes ist neben einer *Bestätigung* der guten Qualität einer Software auch das *Erzeugen* von qualitativ guter Software.

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.

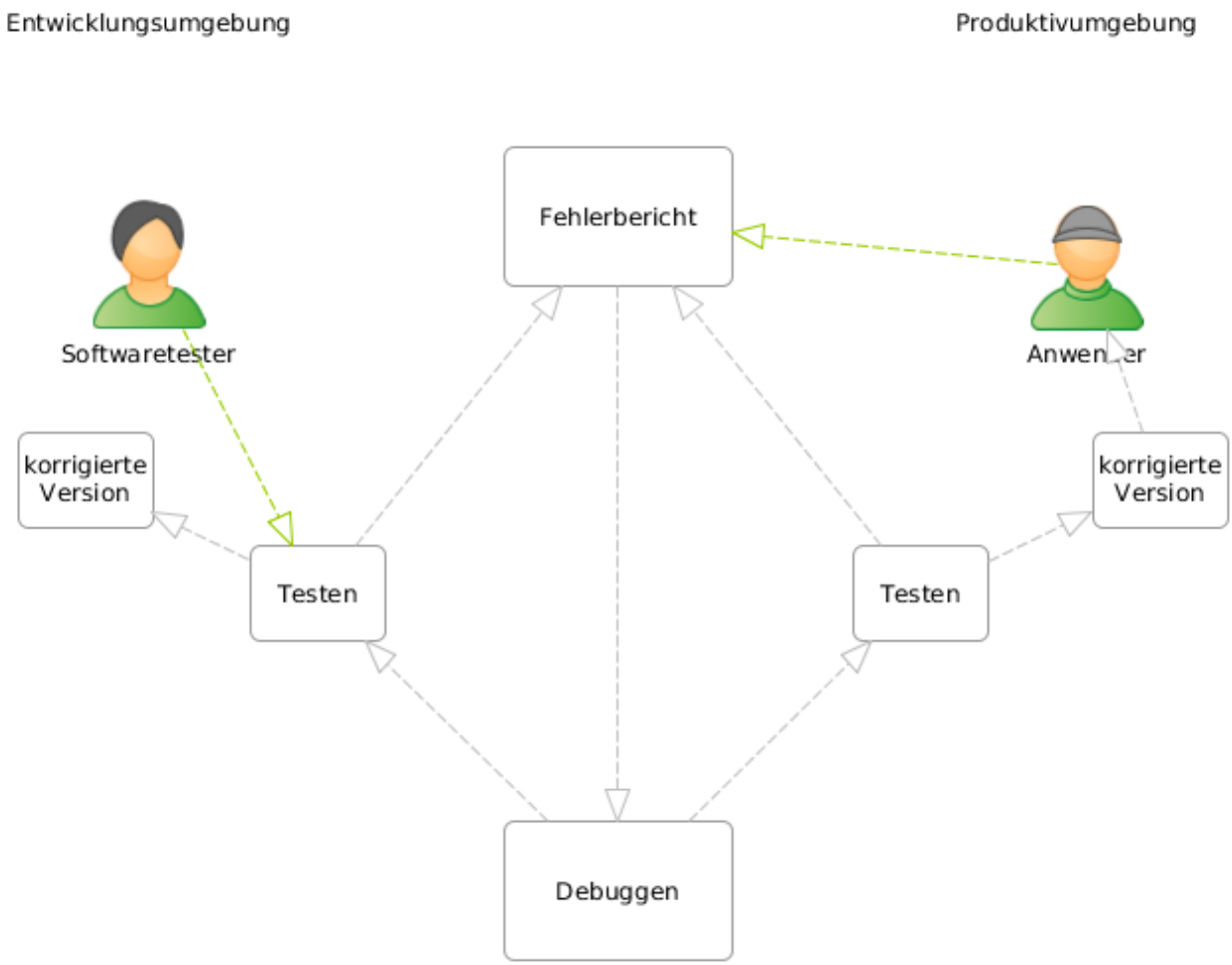


Abbildung 1: Abgrenzung Testen – Debuggen 999

Im Zusammenhang mit Testen und Debuggen hören Sie sicher auch oft den Begriff *Regressionstest*. Ein Regressionstest ist ein wiederholt durchgeführter Testfall. Durch die Wiederholung wird sicher gestellt, dass Modifikationen in bereits getesteten Teilen der Software keinen neuen Fehler oder keine neue Regressionen 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 unkorrekte Programmänderung B nicht mehr auf, weil der neuer Defekt B die Fehlfunktion A maskiert. Nachdem B gefixt ist tritt Fehlfunktion A wieder auf.

Testen und Optimieren

Bei der Optimierung geht es darum optimale Parameter eines komplexen Systems zu finden. Optimal bedeutet, dass eine Zielfunktion minimiert oder maximiert wird. Zielfunktion im Zusammenhang mit Software ist in der Regel die Geschwindigkeit.

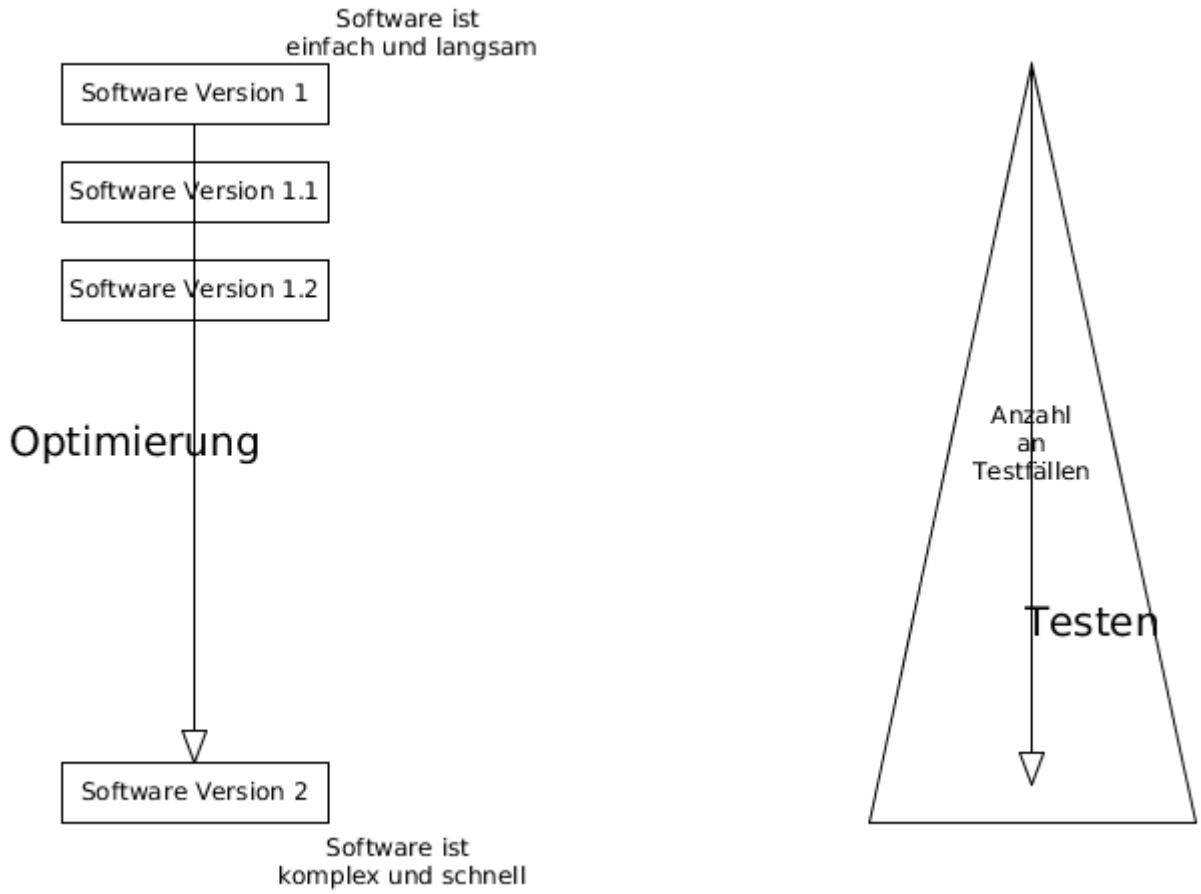


Abbildung 2: Testen und Optimieren 998

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.

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

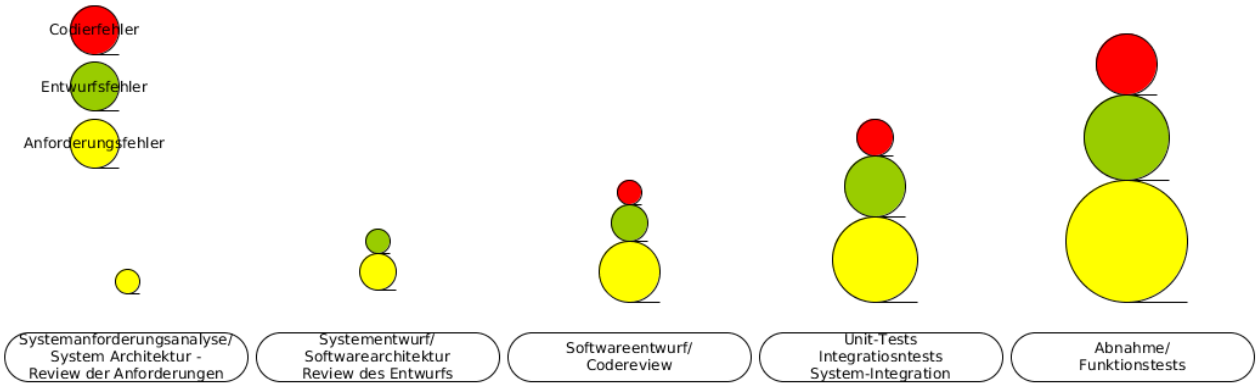


Abbildung 3: Relative Kosten für die Fehlerbehebung 997

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 wir. 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 ausführen. 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.

Wie generell sind Test-Methoden ?

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.

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 das 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)

- Fehlfunkion
  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

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 habe 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" (vorherige Folie: Alle beteiligten Variablen etc. im Kopf behalten)Verstehen von Nebenläufigkeiten
- mangelndes Problemverständnis algorithmische Lösung unzutreffend / unbekannt
- fehlende Information der Beteiligten
- Stress, Übermüdung, fehlende Motivation

Mentale Aspekte des Testens

Folgerungen aus der Psychologie des Testens

Spezifikation, Entwurf, Programmierung sind konstruktiver Prozesse. Das Testen ist ein destruktiver Prozess. Als Tester produziert man kein sichtbares Ergebnis. Im Gegenteil: Oft macht es den Eindruck als ob man alles kaputtmacht.

Der typischer Lernprozess eines Programmierers können Sie sich wie folgt vorstellen:

1. Optimismus: Mein Programm ist perfekt, Testen nicht nötig!
2. Widerspenstigkeit: Mein Programm ist falsch. Ich teste aber trotzdem nicht!
3. Resignation: Ich teste nur noch und gebe das Programm nie mehr frei.
4. Goldener Mittelweg: Ich weiß wann man mit dem Testen aufhören kann.

Ziel sollte ein Selbstloses Programmieren ein. Sie sollten die Sichtweise „ich teste mein Programm" in „ich teste das Produkt um es besser zu machen" austauschen. Jeder gefundene Fehler ist einer weniger.

Ein Programm sollte nicht durch den selbst Entwickler getestet werden. Nur so kann

- mit persönliche Distanz getestet werden.
- ein Missverständnis beim Interpretieren der Spezifikation gefunden werden.
- Ein Zielkonflikt wie zum Beispiel das Einhalten von Zeitvorgaben, vermieden werden.

Täuschungen bei der Inspektion von Programmcode

- Trick- oder Ablauftäuschung

  Ein Programmausdruck tut nicht das was man erwartet.

  if (c = 1) anstelle von if (c == 1)

- Erwartungstäuschung

  Ein Kommentar sagt nicht das aus, was das Programm tut.

  /* Den Wert n-mal aufaddieren */

  for (i==1; i<n; i++) …

- Überdeckungstäuschung

  Ein großer Eindruck überdeckt einen bedeutenden Eindruck. So finden nicht alle Testmethodon subtile Fehler.

- Hemmungstäuschung

  In einem vorherigen Kontext gelernte Zusammenhänge gelten nicht mehr.

  Morgenstern, Abendstern, Zwergelstern

- Wiederholungstäuschung

  Scheinbar gleiche Vorgänge haben unterschiedliche Ergebnisse. Kopierte und leicht veränderte Programmteile

Vermeidung oder Behebung von Fehlern?

Zur Erinnerung: Fehlerursachen sind unter anderem kommunikative Fehler, Komplexität, mangelndes Verständnis, falsche Information. Ein Ausschluss dieser Ursachen ist praktisch unmöglich. Softwaredefekte wird es immer geben. Diese Defekte müssen durch analytische Prozesse gefunden und behoben werden.

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

Softwaretests und Qualitätssicherung

Indem Sie Ihre Software testen bestätigen Sie nicht nur die Qualität Ihrer Software. Sie Erzeugen qualitativ gute Software! In diesem Abschnitt sehen wir uns den Zusammenhang zwischen Softwaretests und Qualitätssicherung genauer an.

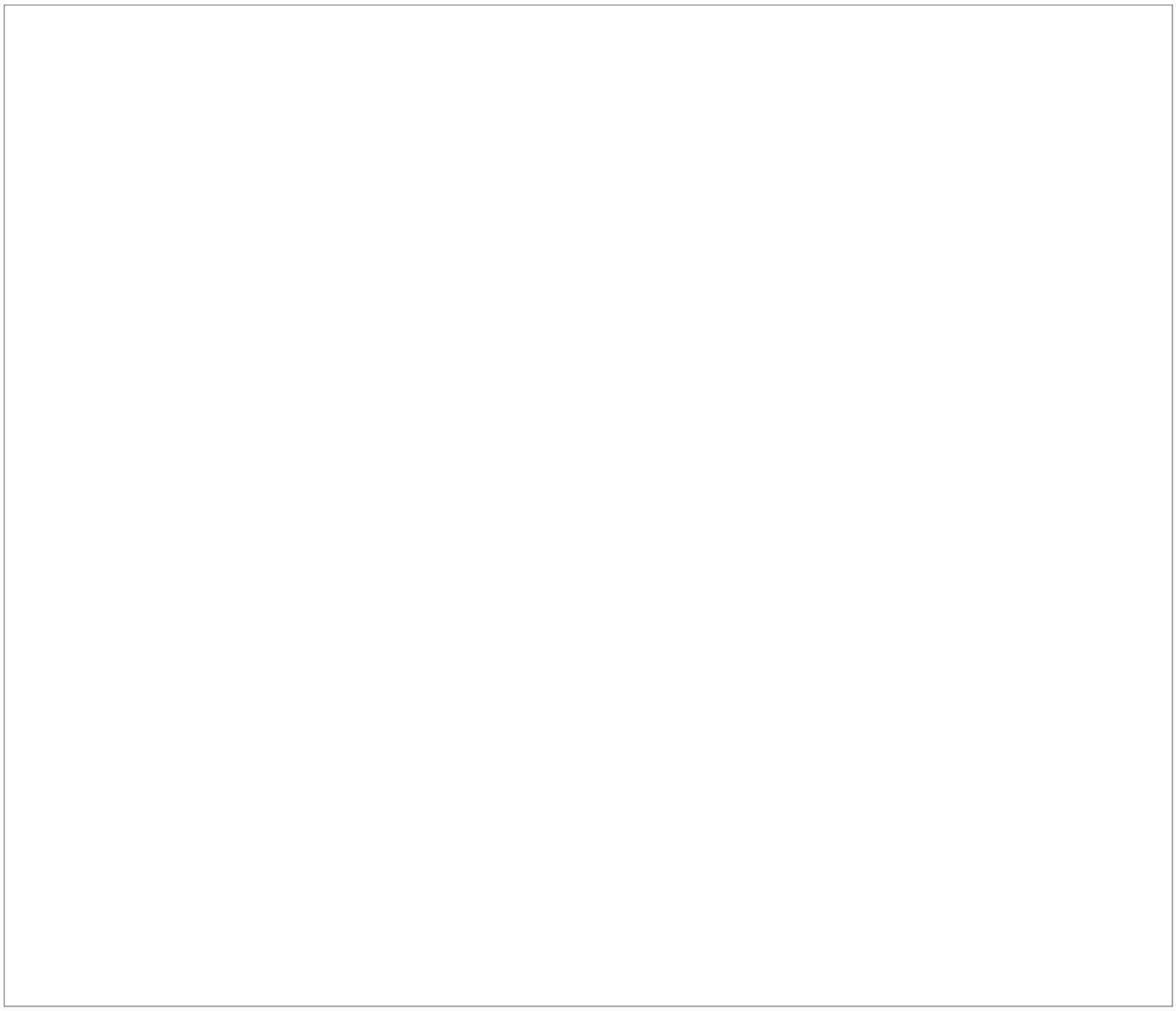


Abbildung 4: Management von Softwareprojekten - Abgrenzung Softwarequalitätssicherung 995

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

1.

Vorgehensweise

1. Zielbestimmung
   Qualitätsmerkmale festlegen und gewichten (Beispiel: Trade-off Effizienz / Änderbarkeit) Ausprägungen der Q-Merkmale angeben
2. Qualitätssteuerung
   QS-Maßnahmen planen, überwachen, auswerten
3. Qualitätsprüfung
   QS-Maßnahmen durchführen

Bild Einordnung

Grundlegende Teststrategien

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

Spezifikationsorientierter Test

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-Interna durchgeführt.

- ohne System
    - zufälliges Testen
    - raten von Fehlern
- systematisch
    - datenbereichsbezogene Tests
    - Funktionsbezogene Tests
    - Pfadausdrücke
    - Tests anhand von algebraischen Spezifikationen

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 werde nicht entdeckt / getestet

- hilft nicht beim Aufdecken von Nebenläufigkeiten
- hilft nicht bei Inkompatibilität zur Konfiguration

Implementationsorientierter Test

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

- Kontrollflussbezogene Tests
- Datenflussbezogen
- Ausdrucks-/Anweisungsbezogen
- Mutationsanalyse

Vorteile implementationsorientierter Tests sind

- fehlende Programmteile werden entdeckt
- Integritätsbedingungen können aufgestellt werden
- Portierungen werden unterstützt

Nachteile implementationsorientierter Tests sind

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

Erzeugen von Testfällen

Testfällen

Was sind Testfälle und wie erstellen Sie systematisch?

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.

$P: D \rightarrow R$

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, 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 ein Test!

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 ja keinen Fehler.

Leider ist die Menge T der idealen Tests aber 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 das ein Test bei dem die Menge der Testfälle gleich der möglichen Eingaben ist, also T = D gilt, ein erschöpfend idealer Test ist.

Todo Verweis erschöpfender Test

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

Todo Systematisches erstellen der Testfälle

Passwort leer bedeutet, dass ein Passwort, dass in der Datenbank zu einem Benutzer gespeichert worden ist, eingegeben wurde.
Todo Diskussion ob Information das Passwort oder Benutzer nicht stimmen ausgegeben werden soll ist hier nebenrangig.

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_USE R
0	0	0	1	JAuthentication:: STATUS_FAILURE Message:: JGLOBAL_AUTH_NO_USE R
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	JAuthentication:: STATUS_FAILURE Message:: JGLOBAL_AUTH_EMPTY_

				PASS_NOT_ALLOWED
1	0	0	0	JAuthentication:: STATUS_FAILURE Message:: JGLOBAL_AUTH_No_USER
0	0	0	0	JAuthentication:: STATUS_FAILURE Message:: JGLOBAL_AUTH_NO_USER

Testmethoden

Test-Driven-Development (Testgetriebene Programmierung) für möglichst hohe Testabdeckung

Im Gegensatz zu Unit Tests, wo tatsächliche Tests geschrieben werden, handelt es sich bei Test-Driven-Development, kurz TDD, um den Prozess des Schreiben und Ausführen der Tests. Folgt man dem Prozess, der aus sechs verschiedenen Stufen besteht, sorgt das für eine möglichst hohe Testabdeckung, sprich je höher die Testabdeckung, desto mehr Code wird automatisch getestet.

Dadurch wird nicht nur die Wahrscheinlichkeit für Bugs reduziert; Projekte mit einer hohen Testabdeckung sind auch deutlich leichter zu verwalten und ermöglichen eine einfachere Implementierung von neuen Funktionen. Test-Driven-Development lässt sich sowohl mit Unit Tests, als auch mit anderen Testmethoden anwenden und ist zudem nicht auf den Einsatz eines bestimmten Tools oder einer bestimmten Syntax angewiesen.

(ToDo genauer)

Best Practices im Behavior-Driven-Development (verhaltensgetriebene Softwareentwicklung)

Im Bereich des automatisierten Testens sorgt Behavior-Driven-Development (BDD) wohl für die größte Verwirrung. Dabei handelt es sich bei BDD um Best Practices für das Schreiben von Tests, die idealerweise gemeinsam mit TDD und Unit Tests zum Einsatz kommen. Im Prinzip zeigt Behavior-Driven-Development, wie man richtig testet – nämlich nicht die Implementierung, sondern das Verhalten des Codes.

(ToDo genauer und Specifyed bei unit tests)

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]

Entwicklungsumgebung und Arbeitsweise

Als Softwareentwickler haben Sie Ihre persönliche Entwicklungsumgebung in der Sie sich sicher und wohl fühlen. Falls dies nicht so ist, sollten Sie hieran arbeiten. Eine optimale Konfiguration Ihre Arbeitsumgebung hilft Ihnen gute Software zu erstellen.

Ich erläutere Ihnen hier die Installation des Computers auf dem ich die Beispiele im Buch erstellt haben. Ich bin der Meinung, dass dies für das Durcharbeiten der praktischen Teile unumgänglich ist.

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 Anpassungen vornehmen müssen.

Meine persönliche Arbeitsumgebung setzt sich aus folgenden Programmen zusammen

- Betriebssystem

XAMPP

Netbeans

Git

Download und Installation der Joomla! Entwicklerversion

Was ist Joomla!?

Joomla! Ist ein Content Management System (CMS) mit dem Sie nicht nur eine Website erstellen und pflegen können. Sie können mit Joomla! leistungsstarke Webanwendungen programmieren.

It is a simple and powerful web server application which requires a server with PHP and either MySQL, PostgreSQL or SQL Server to run. You can find full technical requirements here. Todo Link zu Joomla

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 Opensorce hier oder im nächsten Kapitel

Die Installation

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

(Todo installation)

Am Schluss die Installation testen, damit wir Voraussetzungen für Kapitel Unit Tests haben.

Die Joomla! Architektur verstehen

Wie jedes System besteht Joomla aus mehreren Elementen. Und wie jedes System ist es meiner Meinung nach mehr als die Summe der einzelnen Elemente! ToDo OpenSource?

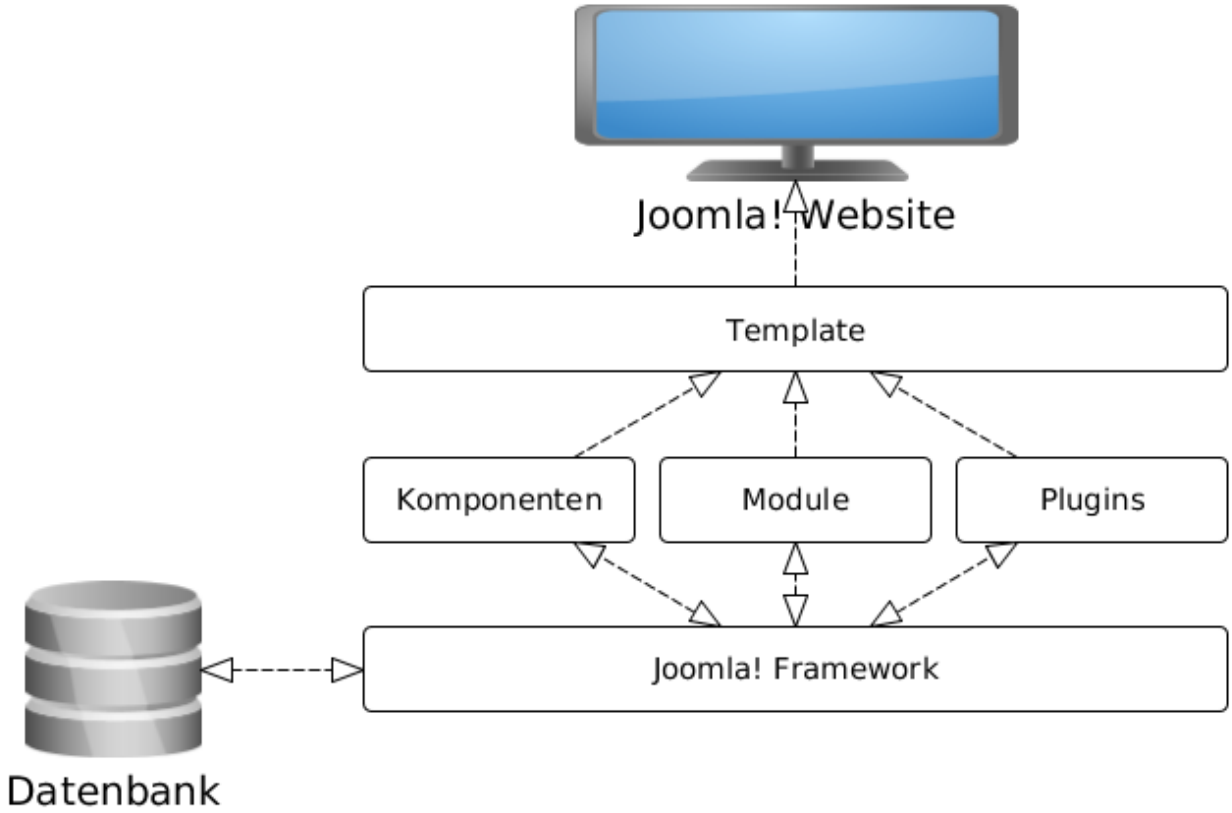


Abbildung 5: Joomla! Architektur 996

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 des Joomla! Frameworks. 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 installierst 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. Ein Plugin ist eine einfache aber sehr effektive Art das Joomla! Framework zu erweitern.

- Template

  Das Template bestimmt das Aussehen Ihrer Joomla! Website.

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 Pakte 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]

Im 3. Kapitel solles 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]

Unit Tests sind Tests, die Sie gleichzeitig mit der Programmierung des eigentlichen Programms erstellen können. Die Tests erfolgen, wie der Name schon vermuten lässt, isoliert von anderen Programmteilen. Unit Tests testen kleine Einheiten. Bei der Testgetriebenen Entwicklung führen Sie die Unit Tests nach jeder kleinen Programmänderung aus. Beim Erstellen eines Unit Tests kennen Sie die zwar Implementierung der zu prüfenden Unit. Unit Tests zählen aber trotzdem zu Black Box Tests. (Todo Verweis) Das Ziel von Unit Tests ist nicht, den Programmcode systematisch zu überprüfen und möglichst alle Programmcodeabschnitte zu teste. Dies tun White Box Tests. Ziel von Unit Tests ist es alle Testfälle abzudecken, egal ob dabei alle Programmcodeabschnitte durchlaufen werden oder nicht. (Verweis zu testfälle und Black und Whitebox tests)

Überblick verschaffen

Ich zeige Ihnen hier am Beispiel von Joomla! wie Sie Unit Test erstellen können. Konkret habe ich die Authentifizierung gewählt. Deshalb sehen wir uns zunächst einmal die Implementierung der Authentifizierung in Joomla! genauer an. Sie lesen dieses Buch sicherlich, weil Sie Test in ihre eigenen Software einbauen möchten. In diesem Fall kennen Sie den Programmcode genau. Lesen Sie dieses Kapitel trotzdem: Es geht auch auf die Vorteile der Testgetriebenen Entwicklung.

Die Authentifizierung in Joomla!

Joomla! prüft mithilfe des Authentication Plugins ob die von einem Benutzer eingegebenen Anmeldedaten korrekte sind. (Todo Verweis zu Aufbau von Joomla). Standardmäßig ist das Joomla! eigene Authentication Plugin aktiviert.

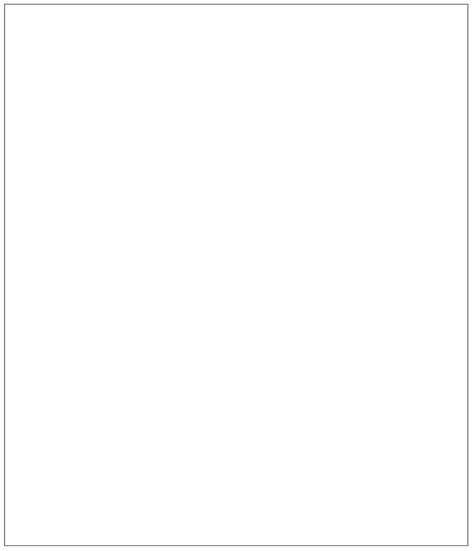


Abbildung 6:
Anmeldeaufforderung im Joomla-Standardtemplate Protostar 995

Dieses Plugin vergleicht die eingegebenen Anmeldedaten mit Werten die in der Joomla! Datenbanktabelle #__users gespeichert sind. (todo #__ erklären.). Darüber hinaus gibt es noch weitere Möglichkeiten zur Authentifizierung. Zum Beispiel können sie die Plugins zur Authentifizierung mittels Gmail oder LDAP aktivieren. Hier beschränken wir uns aber auf die Standardauthentifizierung und dabei auch nur auf einen kleinen Ausschnitt.

Abgrenzung der Begriffe Authentifizierung und Autorisierung

Authentifizierung und Autorisierung ist nicht das Gleiche! Bei der Authentisierung prüft ein System, ob ein Benutzer wirklich der ist, für den er sich vorgibt. Es ist also so etwas wie der Nachweisen der Identität. Eine erfolgreiche Authentifizierung kann ein Benutzer auf verschiedenen Wegen erreichen. Ein Weg ist der Nachweis der Kenntnis einer Information. Der Benutzer weiß etwas das kein anderer weiß, zum Beispiel ein Passwort.

War die Authentifizierung eines Benutzers erfolgreich, erfolgt die Autorisierung. Die Autorisierung weißt dem nun bekannten Benutzer die individuellen Rechte zu. Sie erlaubt einem Benutzer bestimmte Aktionen durchzuführen.

Authentifizierung und Autorisierung spielen in einem Content Management System eine wichtig Rolle bei der Rechteverteilung. Zum Beispiel kann es vorkommen, dass ein Benutzer sich erfolgreich authentifiziert aber trotzdem im System nicht tun kann. Nämlich dann, wenn ihm keine Rechte zugeordnet sind.

> Die Authentifizierung, die das Plugin das wir als Beispiel verwenden durchführt, ist somit nur ein Teil des Anmeldevorgangs.

Die Standardauthentifizierung in Joomla!: Das Authentication Plugins

Sehen wir uns als erstes einmal den relevanten Programmcode an. Sie finden diesen in der Datei WebsererRoot/gsoc16_browser-automated-tests/plugins/authentication/joomla/joomla.php.

Bei jedem Anmeldeversuch wird die Methode `onUserAuthenticate()` aufgerufen. Diese Funktion erwartet drei Eingabeparameter

- $credentials

  Der Parameter `$credentials` ist ein Array, der die Eingaben des Benutzers beinhaltet. Wenn der Benutzer im Feld Benutzernamen den Text *Reiner* und im Feld Passwort den Text *Zufall* eingegeben hat, dann ist `$credentials` folgendermaßen belegt:

  ```
  $array = [
  "username" => "Reiner",
  "password" => "Zufall",
  "secretkey" => ""
  ];
  ```

- $options

  Der Parameter `$options` einhält zusätzliche Optionen, die für die weitere Bearbeitung im Content Management System wichtig sind.

- &$response

  Der parameter `$response` ist ein Objekt des Typs `JAuthenticationResponse`. Ziel der Methode `onUserAuthenticate()` des Authentikation Plugin ist es, dieses Objekt mit dem richtigen Status zu belegen. Der Wert den die Methode mittels `return` ausgibt ist nicht wichtig. Ob die Authentifizierung erfolgreich war wird in der Variablen `$response` gespeichert.

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 Mehtode ihre Argumente modifizieren kann. Beispiel:

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

Die Ausgabe dieser Methode ist 1. Wenn der Parameter $num ohne das &-Zeichen in der Methodensignatur übergeben worden wäre, wäre der Wert 0 ausgegeben worden. Das Originalobjekt wäre nicht geändert worden.

```
...
public function onUserAuthenticate($credentials, $options, &$response) {
 $response->type = 'Joomla';
 if (empty($credentials['password'])) {
        $response->status        = Jauthentication::STATUS_FAILURE;
        $response->error_message =
                Jtext::_('JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED');
        return;
 }
….
```

Ein erstes Testbeispiel

Codeception führt Unit Tests mit PHPUnit aus. Sie können jeden schon fertig geschriebenen PHPUnit Test in die Codeception Testsuite integrieren und ausführen. Außerdem bietet Codeception eine Menge Zusatzfunktionen.

Die Vorlage für den ersten Test

Sie können eine Zusatzfunktion von Codeception nutzen, wenn Sie den ersten Test erstellen – nämlich den Testcode-Generatoren. Generieren Sie einen PHPUnit-Test, der die Klasse Codeception\Test\Unit erweitert, mithilfe des Befehl

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

Sie sehen nun im Verzeichnis

WebsererRoot/gsoc16_browser-automated-tests/tests/codeception/unit/suites/plugins/authentication/joomla/

die Datei

PlgAuthenticationJoomlaTest.php.

Ausführen können Sie den Test in dieser Datei über den Befehl codecept run unit.

```
/var/www/html/gsoc16_browser-automated-tests$ tests/codeception/vendor/bin/codecept run unit
/suites/plugins/authentication/joomla/PlgAuthenticationJoomlaTest.php
Codeception PHP Testing Framework v2.2.7
Powered by PHPUnit 5.7.5 by Sebastian Bergmann and contributors.
….
```

Sie können alle Unit Tests 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.

Der gerade neu generierte Test ist erfolgreich. Alles andere wäre auch verwunderlich. Bisher wird noch kein wirklicher Programmcode getestet. Die automatisch erstellte Datei enthält ausschließlich leere Methoden.

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

Die Testklassen, im Beispiel hier die Klasse PlgAuthenticationJoomlaTest, werden bewusst getrennt von den Produktionsklassen abgelegt. Um unsere Testklasse in Codeception einzubinden, leiten wir sie von dessen Framework-Klasse \Codeception\Test\Unit ab. 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: public function **test**...(). Der Methodenname muss mit dem Wort *test* beginnen.

> Exkurs
> Sie können mit Codeception auch einen klassischen PHPUnit Test, der die Klasse PHPUnit_Framework_TestCase erweitert, generieren. Verwenden Sie dazu einfach den Befehl
> tests/codeception/vendor/bin/codecept generate:**phpunit** unit /suites/plugins/authentication/joomla/PlgAuthenticationJoomlaTest.
> Diesen Test können Sie auf die gleiche Art und Weise wie einen Codeception Test ausführen. Die generierte Datei enthält anstelle der Methoden _before und _after() die Methoden tearUp() und tearDown().

```
<?php
namespace suites\plugins\authentication\joomla;
```

```
class PlgAuthenticationJoomlaTest3Test extends \PHPUnit_Framework_TestCase {
protected function setUp(){}
protected function tearDown(){}
public function testMe(){}
}
```

Der erste Test

Was wollen wir testen?

Todo Verweis zu Testfällen.

Fangen wir ganz vorne an. Die Frage ist nun: Was möchten wir genau testen? Wir wählen die Methode

public function onUserAuthenticate($credentials, $options, &$response)

in der Klasse

WebserverRoot/html/gsoc16_browser-automated-tests/plugins/authentication/joomla/joomla.php

Dazu sehen wir uns den wesentlichen Programmcode dieser Methode noch einmal an. Todo Verweis

```
...
public function onUserAuthenticate($credentials, $options, &$response){
 $response->type = 'Joomla';
 if (empty($credentials['password'])) {
        $response->status = JAuthentication::STATUS_FAILURE;
        $response->error_message =
                JText::_('JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED');
        return;
 }
….
```

Fangen wir mit dem Testfall an, bei dem der Benutzer weder einen Benutzernamen noch ein Passwort eingibt. Bei einem fehlerfreien Programmablauf sollte folgendes passieren:

1. Die Eigenschaft type des Objektes $response wird mit dem Wert „Joomla“ belegt.

2. Die Eigenschaft status des Objektes $response wird mit dem Wert JAuthentication::STATUS_FAILURE belegt.
3. Die Eigenschaft error_message des Objektes $response wird mit dem Wert JText::_('JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED') belegt.

Ausschließlich diese Punkte müssen wir testen wenn wir die korrekte Bearbeitung einer leeren Eingabe beim Anmeldeversuch eines Benutzers testen möchten. Ob andere Bedingungen korrekt gehandhabt werden, wird in den dafür zuständigen Klassen geprüft. Wir Testen hier nur diese Einheit unabhängig vom ganzen Programm!

Todo Exkurs JText

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

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

Programmcode von anderen testen

Oft kommt es vor, dass man nicht nur seinen eigenen Programmcode testet, sondern auch den, den andere Menschen erstellt haben. Insbesondere dann, wenn man Open Source Software nutzt oder vielleicht sogar in einem Open Source Projekt mitarbeitet.

Die Vorteile der Testgetriebenen Programmierung habe ich im Kapitel Testmethoden zusammengefasst. Diese Softwareentwicklungsmethode wird aber nicht immer eingesetzt und somit gibt es Software ohne automatische Tests.

Das Plugin zur Authentifizierung in Joomla!, welches wir hier als Beispiel nutzen, ist auch ohne zugehörigen Unit Test implementiert.

Was gibt es für Gründe nachträglich einen Test hinzuzufügen? Mir fallen alleine beim Ansehen des Plugin zur Authentifizierung in Joomla! folgende ein:

1. Beim Einsatz einer Open Source Software die Sie frei nutzen dürfen, könnten Sie als Dank an der Verbesserung der Software mitarbeiten indem Sie einen fehlenden Test ergänzen.
2. Beim Erstellen eines Tests wird die Produktivsoftware fast immer verbessert.

a. Zum Beispiel sollten Methoden kurz sein, kaum länger als 20 Zeilen. Alle Methoden die länger sind, lassen sich zu kleineren Funktionen refaktorieren. Die Methode onUserAuthenticate() im Authentifizierungs-Plugin besteht aus etwa 200 Zeilen. Dies macht es fast unmöglich Tests zu schreiben, ohne vorher die Struktur zu verbessern. Unter anderem wird in dieser Methode auf die Datenbank zugegriffen. Dieser Teil muss für den Test separiert werden, da beim Testen des Algorithmus der Methode immer mit eigenen Daten gearbeitet werden sollte.

b. In einer Methode können Werte mit der optionalen return-Anweisung zurückgegeben werden. Hierbei können Variablen jeden Typs zurückgegeben werden. Eine return-Anweisung beendet die Methode sofort und gibt die Kontrolle an die aufrufende Zeile zurück. Wenn in der return-Anweisung kein Wert übergeben wird, wird automatisch null zurückgegeben. Die Bezeichnung null nennt schon den Grund, warum es grundsätzlich keine gute Idee ist, diesen Wert als Rückgabewert in Methoden auszugeben. null ist schlichtweg vollkommen bedeutungslos! Würde die Methode onUserAuthenticate() anstelle von
return
die Anweisung
return JText::_('JGLOBAL_AUTH_EMPTY_PASS_NOT_ALLOWED')
nutzen, könnte ein Test anhand des Rückgabewertes genau feststellen an welcher Stelle die Methode abgebrochen wurde. Vielleicht ist der Rückgabewert auch an anderen Stellen im Programm nützlich. Schaden tut er Rückgabewert sicher nicht.

Die Implementierung des ersten Tests

In Kapitel Die Vorlage für den ersten Test auf Seite 35 hatten wir eine Testdatei generiert. Die Methoden in der Datei sind noch leer. Dies ändern wir nun!

Sie finden diese generierte Datei im Verzeichnis

WebserverRoot/gsoc16_browser-automated-tests/tests/codeception/unit/suites/plugins/authentication/joomla/

Öffnen Sie in diesem Verzeichnis die Datei

PlgAuthenticationJoomlaTest.php.

Für den ersten Test füllen wir die Methode testonUserAuthenticate_EmptyCredentials(). Jetzt wird es endlich konkret! Sehen Sie sich zunächst einmal die fertige Methode an:

```
<?php
namespace suites\plugins\authentication\joomla;
class PlgAuthenticationJoomlaTest extends \Codeception\Test\Unit
{
protected $class;
protected function _before() {}
protected function _after() {}
public function testonUserAuthenticate_EmptyCredentials() {
        require_once JINSTALL_PATH . '/libraries/joomla/event/dispatcher.php';
        $subject = \JeventDispatcher::getInstance();
        $config = array(
                'name' => 'jooma"',
                'type' => 'authentication',
                'params' => new \JRegistry
        );
        require_once JINSTALL_PATH . '/plugins/authentication/joomla/joomla.php';
        $this->class = new \PlgAuthenticationJoomla($subject, $config);
        $credentials = array(
                'username' => "",
                'password' => "",
                'secretkey' => ""
        );
        $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'));
}
}
```

Nun rolle ich den Programmcode von hinten auf: Ich haben für jede Bedingung eine Behauptung aufgestellt. Dies habe ich mithilfe der assertEquals-Methode getan. Hier musste ich nur die Parameter so setzten, dass die Bedingungen erfüllt sind. 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 PlgAuthenticationJoomla instanziiert und in der Variablen $class gespeichert. Dazu musste ich vorher die Objekte $subject und $config erzeugen. Diese Objekte benötigt die Klasse PlgAuthenticationJoomla bei der Instantiierung.
- Die Objekte $credentials, $options und $response habe danach erstellt, weil diese als Parameter in der Methode onUserAuthenticate benötigt werden. Beim Objekt credentials habe ich bewusst den Benutzernamen und das Passwort leer gelassen. Es soll ja die leere Eingabe getestet werden.
- Nun habe die die zu testende Methode aufgerufen. $this->class->onUserAuthenticate($credentials, $options, $response);.
- Zum Schluss habe dann die im Kapitel Was wollen wir testen?auf Seite 37 beschriebenen Bedingungen als Parameter in die Methode assertEquals() geschrieben. Diese Methode verifiziert die Gleichheit zweier Objekte.

Den ganzen Testprogrammcode habe ich in die Testfallmethode testonUserAuthenticate_EmptyCredentials() geschrieben. Das PHPUnit-Framework findet Methoden, deren Name mit test beginnt, automatisch und führt sie aus.

Geben Sie nun den Befehl

tests/codeception/vendor/bin/codecept run unit
/suites/plugins/authentication/joomla/PlgAuthenticationJoomlaTest.php

ein um den Test auszuführen.

```
/var/www/html/gsoc16_browser-automated-tests$ tests/codeception/vendor/bin/codecept run unit
/suites/plugins/authentication/joomla/PlgAuthenticationJoomlaTest.php
Codeception PHP Testing Framework v2.2.7
Powered by PHPUnit 5.7.5 by Sebastian Bergmann and contributors.
JPATH_TESTS /var/www/html/gsoc16_browser-automated-tests/tests/codeception/unit
JINSTALL_PATH /var/www/html/gsoc16_browser-automated-tests/tests/codeception/joomla-cms3
```

> JPATH_LIBRARIES /var/www/html/gsoc16_browser-automated-tests/tests/codeception/joomla-cms3/libraries
> JPATH_PLUGINS /var/www/html/gsoc16_browser-automated-tests/tests/codeception/joomla-cms3/plugins
> Unit Tests (1) ----------
> ✔ PlgAuthenticationJoomlaTest: Teston user authenticate_empty credentials (0.01s)
> ---
> Time: 163 ms, Memory: 10.00MB
> OK (1 test, 3 assertions)

Bei der Ausführung des Tests ist keine Fehlfunktion gefunden worden. Ich hoffe das war bei Ihnen auch so.

> Standardmäßig sagt Ihnen PHPUnit genau was fehlgeschlagen ist. Oft ist aber nicht direkt klar was dieser Fehler genau bedeutet. Zum Beispiel gibt PHPUnit aus, dass zwei Zeichenketten nicht gleich sind,
> **Failed asserting that two strings are equal.**
> wenn die Behauptung
> **$this->assertEquals($response->type, 'Joomla');**
> fehlschlägt.
> Sie können sich das Leben leichter machen, wenn Sie Nachrichten in den Testanweisungen als Parameter mitgeben. Zum Beispiel könnten Sie folgende Anweisung verwenden:
> **$this->assertEquals($response->type, 'Joomla', "Der Typ im Response Object ist nicht richtig gesetzt!");**
> Der Text "Der Typ im Response Object ist nicht richtig gesetzt!" wird im zweiten Fall bei einem fehlgeschlagenen Test neben der Nachricht von PHPUnit mit ausgegeben. Dieser kleine Mehraufwand kann auf lange Sicht viel Zeit sparen. In diesem Buch hier nutze ich diesen Parameter nicht. Die Codebeispiele sind ohne diesen Parameter kompakter.

Die Konfigurationsdatei

Falls etwas nicht klappt Konfiguration überprüfen. Todo auch yml und bootstrap

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 überprüfen. An das Ende des Namens wir lediglich ein Test abgehangen. 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 PHP Unit in der Klasse Assert unterschiedliche Prüfmethoden. Unser Klasse erben diese Prüfmethoden. Todo UML zur Vererbung.

Die Testmethoden der Klasse Assert

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

Die wichtigsten Assertations sind meiner Meinung nachträglich

- 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 mehrere 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 Annotations 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 Kommentarblocks 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 Exeptions arbeiten, sollten sie die Annotation @expectedExeption 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 aufführen!

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 Test auf Seite 37 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 Testfällen ab Seite 22 herausgearbeitet.

Es wäre langweilig, für jeden dieser Fällte 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 Mehtode.

```
<?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' => 'jooma"',
                '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 habe 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.

Fixturen

Eine Fixtur setzten 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, das 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 boostrap Datei vor allen Testdateien zusammen setzten. (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' => 'jooma"',
		'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 nuŕ 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 bestimmt Methoden automatisch ausgeführt.

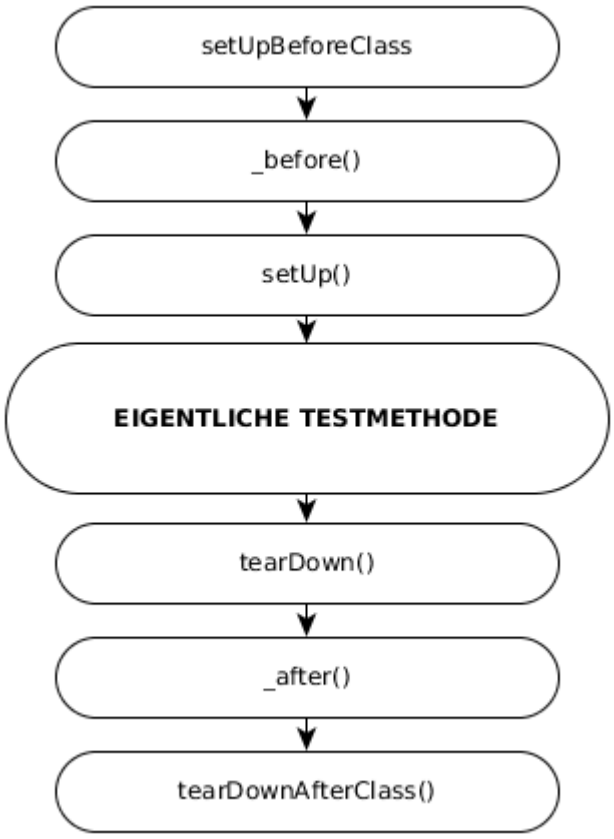


Abbildung 7: kk

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

1. tests\codeception\unit\plugins\authentication\joomla\PlgAuthenticationJoomlaTest**.setUpBeforeClass()**
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 effektive 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 häufigste die Mehtoden 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, dies 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 ansehe, schon einmal unser Testumgebung vor. Die Fixturen $config und $options werden wir in dem Test in unser 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 Fixturen 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' => 'jooma"',
                '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 Fixturen im vorausgehenden Kapitel haben wir schon mitwirkende Objekte, die selbst nicht direkt getestet werden, gesehen. Fixturen stehen für triviale *unechte* Implementierungen. Wie wir gesehen haben werden in der Regel vordefinierte Werte zurück gegeben. Fixturen sollen meist, komplexe Abläufe oder Berechnungen in der Anwendung zu ersetzten.

> Für Testduplikate gibt unterschiedliche Bezeichnungen. Martin Fowler unterscheidet die Namen Dummy, Fake, Stub und Mock Objekte. Und dann gibt es den Begriff Fixturen. 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: irgenwo 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 konnen 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' => 'jooma"',
                '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' => 'jooma"',
                '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 Joomal! 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 Fixturen 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 er 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 er 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 vorhanden 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' => 'jooma"',
                '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,
\JText::_('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 Programmcodeteilen. 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/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 weiter Aufruf würde den vorherigen überschrieben. 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 sie 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 Test 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 Objket 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 kannst du 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 sind mysteriös scheitern, wenn sie nicht sollten - versuchen, einen einzelnen Test durchzuführen. Dies wird sehen, 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 einführung 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 wieter 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 öffenet sich :)

Wir erstellen den ersten Akzeptanztest

Sie wissen nun wie Akzeptanztests, die Selenium Webdiver 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)

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

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.