BACHELORARBEIT

im Studiengang Bachelor Informatik

Software-Test von Web-Applikationen

Ausgeführt von: Bernhard Posselt
Personenkennzeichen: 1010257029

Begutachter: Benedikt Salzbrunn, MSc

Wien, 2. Juni 2013

FACHHOCHSCHULE
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht."

Ort, Datum

Unterschrift

Kurzfassung

Die Durchführung und Erstellung von automatisierten Tests für Web-Applikationen unterscheidet sich von klassischen Applikationen: Aufgrund der komplexeren Infrastruktur und Modularisierung werden zusätzliche Testfälle und Strategien benötigt, um Web-Applikationen ausreichend abzudecken und eine fortwährende Qualität zu gewährleisten. Diese Arbeit soll Möglichkeiten für den Test von Web-Applikationen anhand eines Projektes und Anpassungen und Anwendung der vier Testformen des V-Modells aufzeigen: Unit Test, Integration Test, System Test und Acceptance Test.

Schlagwörter: Softwaretest, Web-Applikation

Abstract

Creation and execution of automatic web application tests is different from tests of classic applications. A more complex infrastructure and modularisation requires additional testcases and strategies to guarantee a good enough test coverage which in return ensures constant quality. This thesis highlights various possibilities to test web applications based on a real world project and shows how to use and adjust the V-Model's four test methods: Unit Test, Integration Test, System Test and Acceptance Test.

Keywords: software test, web applications

Inhaltsverzeichnis

1 Einführung

Immer mehr unserer Alltagsgeräte ermöglichen einen Zugang zum Internet. Besonders der Markt für mobile elektronische Geräte hat in den letzten Jahren ein rasantes Wachstum erlebt: Im Jahr 2011 wurden erstmals mehr Smartphones und Tablets als PCs verkauft(Abbildung 1.1).

Aufgrund des starken Wachstums dehnt sich der Markt für Software-Applikationen auch auf Smartphones und Tablets aus, welche zum Großteil andere Betriebssysteme und Applikations-Frameworks verwenden als traditionelle Computer (Abbildung 1.2). Viele dieser Plattformen erfordern das Erlernen von unterschiedlichen Programmiersprachen, Frameworks und Betriebssystemen [1][2]. Will ein/eine Software-EntwicklerIn eine Applikation plattformübergreifend anbieten, erfordert dies daher einen höheren Zeit- und Kostenaufwand.

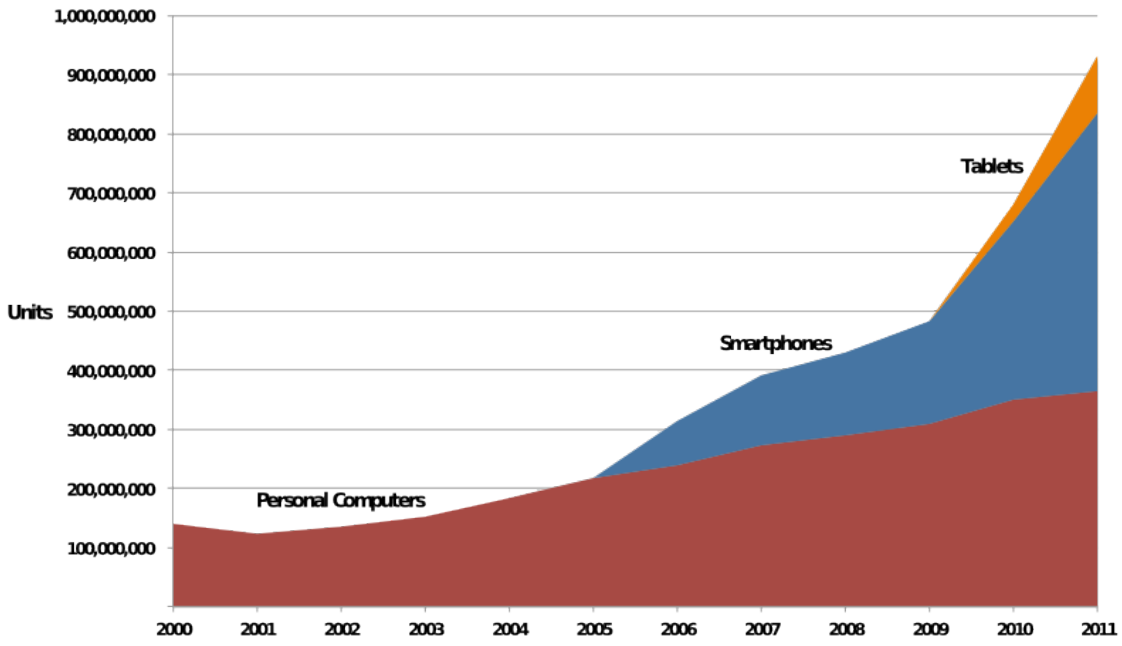

Abbildung 1.1: Entwicklung der PC und Mobilgerätverkäufe [3][S. 6]

Dieses Problem der Fragmentierung und des damit verbundenen immer höheren Portierungsaufwandes kann mit einer Web-Applikation gelöst werden. Soll sich diese auch noch gut in das jeweilige System integerieren, können Frameworks wie z.B. PhoneGap [4] verwendet werden. Diese Frameworks bieten über JavaScript einen Zugriff auf native Funktionen der mobilen Betriebssysteme, womit der Portierungsaufwand deutlich reduziert werden kann.

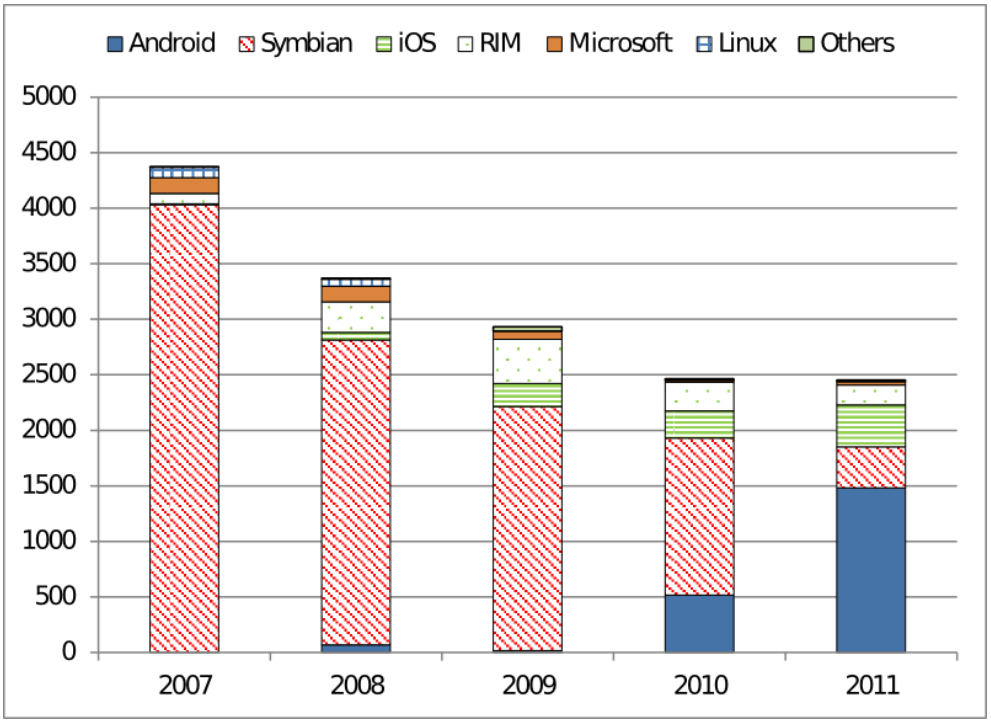

Abbildung 1.2: Marktanteil mobiler Betriebssysteme [5][S. 22]

1.1 Problemstellung

Eine Web-Applikation wird innerhalb eines Web-Browsers ausgeführt. Da mehrere unterschiedliche Browser existieren, erhöht sich damit der Testaufwand der Web-Applikation. Zudem implementieren die Browser die vom W3C vorgeschlagenen Standards nicht immer gleich und in unterschiedlicher Geschwindigkeit: Ein Beispiel dafür ist der Internet Explorer 8, der über einen eigenen Quirks Modus verfügt, um älteres, nicht standardkonformes Markup zu rendern, welches für ältere Internet Explorer Versionen geschrieben wurde [6][S. 41]. Daher kann es vorkommen, dass gewisse Funktionen nur in einer eingeschränkten Auswahl von Web-Browsern voll funktionsfähig sind und zusätzliche Anpassungen erforderlich sind. [7]

Ein weiteres Problem ist das verhaltene Upgradeverhalten der NutzerInnen. Im Falle des Internet Explorers 7 dauerte es beispielsweise länger als 19 Monate, bis rund die Hälfte der NutzerInnen auf die neue Version umgestiegen sind. [8][S. 3]

All dies zwingt Web-EntwicklerInnen dazu, eine Vielzahl an unterschiedlichen Browsern und Browser-Versionen zu unterstützen.

1.2 Lösungsansatz

Um eine konstante Qualität der Web-Applikation auf allen Plattformen zu gewährleisten, muss das bisherige Testssystem angepasst werden, um der höheren Komplexität von Web-Applikationen gerecht zu werden.

Dazu werden die jeweiligen Testarten des V-Modells - Unit Test, Integration Test, System Test und Acceptance Test - anhand eines Projektes an eine Web-Applikation angepasst. Sowohl client-

als auch serverseitige Tests werden erstellt, um die verschiedenen Plattformen erfolgreich testen zu können. Dies wird mit dementsprechenden Code-Beispielen illustriert.

1.3 Aufbau

Der erste Teil der Arbeit (Kapitel 2, 3 und 4) behandelt den Sinn und die Grenzen des Software-Tests und wie das Testen sich in den Entwicklungsprozess einbinden lässt. Es wird auf Unterschiede zwischen Software-Test von klassischen und Web-Applikationen wird eingegangen. Außerdem wird anhand des V-Modells versucht, den Zusammenhang zwischen Software-Test und Projektablauf aufzuzeigen. Schlussendlich wird der Ablauf des Testprozesses mit Hilfe eines Testplanes erläutert.

Der zweite Teil (Kapitel 5, 6, 7, 8) beschäftigt sich mit den praktischen Anpassungen der verschiedenen Testarten des V-Modells. Kapitel 5 beschäftigt sich näher mit dem praktischen Einsatz von client- und serverseitigen Unit Tests. Es werden für beide Seiten Beispiele eines Unit Tests erstellt und auf klassiche Probleme sowohl bei client-, als auch serverseitigen Unit Tests und deren Lösungen eingangen. Das Thema des Integration Tests wird in Kaptitel 6 behandelt. Außerdem wird auf die verschiedenen Test-Ansätze eingegangen. Darauf folgt der System Test in Kapitel 7. Es wird anhand von Beispielen gezeigt, wie die verschiedenen Server- und Client-Plattformen automatisch getestet werden können. Schlussendlich wird noch der Acceptance Test in Kapitel 8 behandelt.

2 Warum testen

Keine Applikation ist fehlerfrei[9][S. 9]. Diese Fehler führen nicht nur zu unzufriedenen Kunden/-Kundinnen, sondern auch zu hohen Kosten: „Im Jahr 2000 wurde in den USA ein Schaden durch Softwarefehler in der Auto- und Flugzeugindustrie von 1,8 Milliarden US-Dollar errechnet. Dies entspricht ca. 16 % des Softwareumsatzes.“[10][S. 15]

Die Fehler-Rate wird auf ungefähr 3 pro 1.000 Zeilen geschätzt, was bei einer aufwendigeren Applikation mit 100 Millionen Zeilen Quellcode eine durchschnittliche Anzahl von 300.000 Fehlern ergibt [11][S. 10].

Je früher diese Fehler entdeckt werden, desto kostengünstiger können diese beseitigt werden [10][S. 17]. Daher ist es wichtig, möglichst früh mit dem Testen zu beginnen und den Software-Entwicklungsprozess dementsprechend anzupassen [10][S. 16].

2.1 Verschiedene Testarten

Laut dem V-Modell (Abbildung 2.1) ist der Software-Test kein seperater Abschnitt im Projektplan. Vielmehr ist er ein Prozess, der die verschiedenen Entwicklungsabschnitte ergänzt [10][S. 23]. Das V-Model unterscheidet zwischen vier verschiedenen Testarten:

- Unit Test: Wird paralell zur Implementierung erstellt; Testet Komponenten und Klassen
- Integration Test: Wird erstellt, um den Designprozess zu überprüfen. Testet die Kommunikation zwischen verschiedenen Klassen und Komponenten
- System Test: Testet die konkreten Anforderungen an die Software und die Systemschnittstellen
- Acceptance Test: Testet die Anforderungen des/der Kunden/Kundin

Tests können jedoch nicht nur dazu verwendet werden, um Fehler in der Applikation zu finden, sondern auch um einen Überblick über den derzeitigen Stand der Implementierung zu gewinnen: Sie geben dem/der EntwicklerIn und ProjektmanagerIn ein direktes Feedback über bereits korrekt implementierte Teile der Software-Spezifikation. Auch Milestones können durch Tests definiert werden[13][S. 2]. Zudem ist eine Abschätzung riskanter Bereiche möglich, die durch eine erhöhten Testbedarf bestimmter Bereiche offensichtlich wird. [14][S. 34]

2.2 Manuelle Tests

Manuelle Tests eignen sich vor allem im Bereich des System Tests und Acceptance Tests, da sich diese Bereiche oft schwer komplett automatisch testen lassen. Darunter fallen z.B. Kontrollen der Übersetzungen und der Dokumentation, Usability Tests, externe Beta Tests, Security Tests und explorative Tests. [15][S. 61]

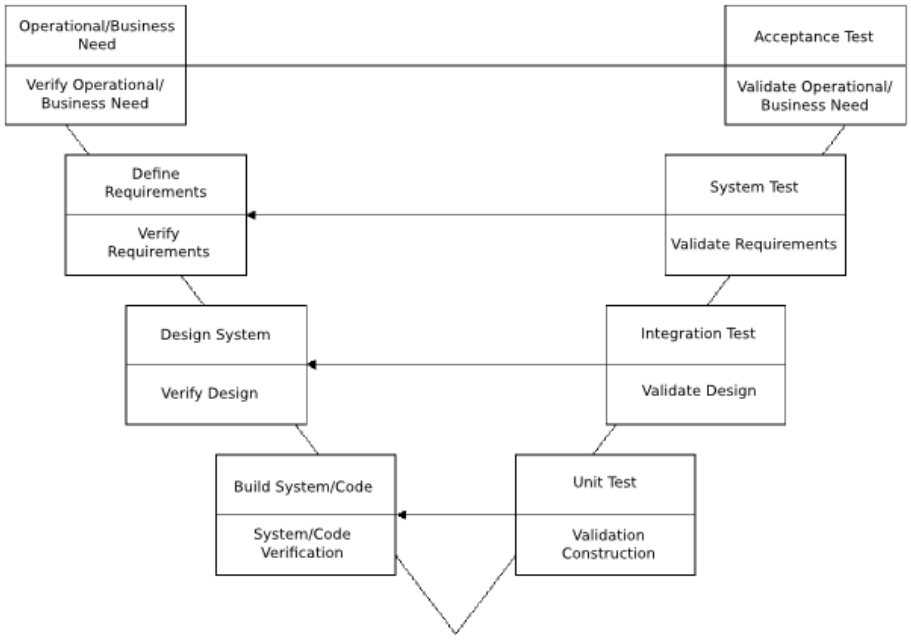

Abbildung 2.1: Schematische Darstellung des V-Modells [12][S. 3]

2.3 Automatisierte Tests

Da Tests durch die Einbindung in den Entwicklungsprozess öfters durchgeführt werden müssen, kann durch das repetetive Durchführen immergleicher Prozesse beim/bei der TesterIn schnell eine gewisse Eintönigkeit und Genervtheit entstehen. Das wiederum kann unter anderem dazu führen, dass TesterInnen im Laufe der Zeit bestimmte Tests nicht korrekt oder ineffizient ausführen.

Die Lösung des Problems ist die teilweise Automatisierung des Testprozesses. Diese erlaubt bei zukünftigen Testdurchläufen nicht nur eine starke Reduzierung des Zeitaufwandes pro Durchlauf, sondern ermöglicht es auch, die Tests rund um die Uhr durchzuführen. Oft werden am Abend alle zeitintensiven Tests gestartet, damit die EntwicklerInnen am nächsten Morgen einen guten Überblick darüber haben, was nicht oder nicht mehr funktioniert. [13][S. 22-23]

Es gibt mehrere Faktoren, die bei der Entscheidung der Automatisierung berücksichtigt werden müssen:

- Es muss klar sein, dass der Aufwand der Testerstellung einen wirtschaftlichen Nutzen hat. Bei einer sehr kleinen und unkritischen Applikation kann es sich z.B. nicht lohnen, Tests zu automatisieren. Dies liegt unter anderem daran, dass Tests erst über einen längeren Zeitraum ihre volle Stärke ausspielen. [11][S. 12]
- Das Personal muss für die Erstellung der automatischen Tests ausgebildet sein. [16][S. 37]
- Es eignet sich nicht jede Testart zur vollkommenen Automatisierung. Währen Unit-Tests und Integration Tests sehr gut automatisiert werden können, sind System und Acceptance Tests schon schwieriger zu automatisieren.

2.4 Grenzen von Software-Tests

Software-Tests können nie eine komplett fehlerfreie Software garantieren. Es kann höchstens eine Fehlerabwesenheit für jene Fälle garantiert werden, welche mit Tests abgedeckt wurden [11][S. 12]. Dies resultiert unter anderem daraus, dass die Anforderungen an die Software oft nicht komplett spezifiziert oder zu ungenau formuliert sind. Zusätzlich steigt die Komplexität der Applikation im Laufe der Entwicklung stark an. Ein weiterer Grund stellt die oft schier unbegrenzten Möglichkeiten an unterschiedlichen Eingaben dar. [17][S. 243].

Die implementierten Testfälle müssen zudem regelmäßig gewartet und aktualisiert werden, um ihre Effektivität zu gewährleisten, denn diese nimmt auf Dauer ab. Es ensteht eine sogenannte „Testresistenz“[11][S. 12], die daraus resultiert, dass die bestehenden Tests nur bekannte Fehlerfälle abdecken und keine neuen, möglichen Fehlerfälle berücksichtigen. [11][S. 12-13]

3 Testplan

Mit dem Erstellen des Projektplans wird gleichzeitig auch ein Testplan erstellt werden [16][S. 24]. Dieser erlaubt, den Testprozess näher zu spezifizieren und somit den erforderlichen Aufwand besser abschätzbar zu machen [15][S. 18]. Sollte es notwendig sein, kann damit der Testprozess auch an ein externes Team ausgelagert werden. Wird der Testplan zudem in den Abnahmenkriterien verankert, reduziert sich das Projektrisiko durch klar definierte Kriterien. [16][S. 26]

3.1 Inhalt

Im Testplan werden unter anderem folgende Punkte behandelt [13][S. 3]:

- Was wird getestet?
- Welche Bereiche besitzen eine höhere Priorität als andere?
- Wo wird getestet?
- Mit welchen Konfigurationen, Soft- und Hardware Plattformen respektive Versionen wird getestet?
- Wie wird getestet?
- Mit welchen Tools wird getestet?
- Wer testet?
- Wie lange wird getestet?
- Bis wann muss ein Test erfolgreich absolviert werden?
- Was wird nicht getestet?

3.2 Ablauf

Da die Entwicklung der Applikation meistens unter einem hohen Zeitdruck abläuft[17][S. 244] und der Testprozess zudem teuer ist [11][S. 24], ist es wichtig, Testfälle heraus zu arbeiten, zu priorisieren und an geeignete TesterInnen zu verteilen. Auch Endkriterien werden für die Tests veranschlagt und eine optimale Test-Strategie gewählt werden.

Danach folgt die Überprüfung des Testplans. Hier wird überprüft, ob die einzelnen Testfälle genau genug beschrieben worden sind, um daraus die jeweiligen Testfälle erstellen zu können. Auch die Effiktivität der Test-Strategien wird überprüft. Diese Prüfung findet nicht nur einmal statt, sondern wiederholt sich mehrfach im Testprozess. Dies hängt damit zusammen, dass die Anforderungen an die Software im Laufe der Implementierung immer konkreter werden und somit auch neue, mögliche Fehlerfälle erkannt werden können.[11][S. 25]

Ist der Testplan soweit fertig, kann ein Zeitplan für die einzelnen Testfälle erstellt werden. In weiterer Folge werden die einzelnen Tests an die TesterInnen vergeben. Diese können nun mit der Erstellung der konkreten Testfälle beginnen.

Manche Tests haben eine sehr hohe Laufzeit. Um den Ablauf daher zu beschleunigen, wird ein Smoke-Test erstellt. Der Smoke-Test testet grobe Szenarien der Applikation. Schlagen diese fehl, sind weitere Testdurchläufe nicht notwendig und der Testdurchlauf kann abgebrochen werden.

Nach dem Ausführen der Tests wird ein Bericht erstellt. Je nach Ausgang der Tests muss der Testplan angepasst werden. Verliefen alle Tests positiv, kann das die jeweilige Phase abgeschlossen werden. [11][S. 26]

3.3 Vorteile eines Testplans

Aufgrund der Dokumentation der Testfälle, ergeben sich zusätzlich folgende Vorteile:

- Fehlende Testfälle können schnell erkannt werden [15][S. 18]
- Redundante Testfälle werden identifziert; Testfälle können zusammegelegt werden um Zeit zu sparen [14][S. 34]
- Verschiedene Testbereiche können an mehrere Personen mit unterschiedlichen Fachkenntnissen verteilt werden, um Personalengpässe zu vermeiden [15][S. 19]
- Ineffiziente Test-Tools und Test-Strategien können erkannt und verbessert werden [11][S. 25]
- Geschäftskritische Bereiche und solche mit einer erhöhten Fehleranfälligkeit werden sichtbarer [14][S. 34]. Dies ist vor allem darum wichtig, weil Fehler ungleich verteilt sind und in bestimmten Bereichen häufiger vorkommen als in Anderen [11][S. 12]
- Sinnlose Testfälle und Testbereiche können aussortiert werden
- Abhängigkeiten der Testfälle untereinander und der Implementierung werden transparent und erlauben schnell auf Änderungen zu reagieren[13][S. 4]

4 Zusätzliche Herausforderungen beim Testen von Web-Applikationen

Eine Web-Applikation unterscheidet sich an mehreren Stellen von einer klassichen Applikation, was das Testen und das Auffinden von Fehlern zusätzlich erschwert. Die Hauptunterschiede sind:

- Modularer Aufbau
- Plattformunabhängigkeit
- Session Modell

4.1 Modularer Aufbau

Im Gegensatz zu klassischen Desktop- oder Mobil-Applikationen bestehen Web-Applikationen wegen ihrer Client-Server-Architektur immer aus mehreren Modulen (Abbildung 4.1). Diese Module sind meist über ein Netzwerk miteinander verbunden.

Durch diesen modularen Aufbau ist es besonders schwer einen Fehler zu lokalisieren. Ein Fehler kann z.B. durch einen Fehler im Quellcode des Applikations-Servers oder durch ein Netzwerkproblem entstanden sein. [14][Foreword]

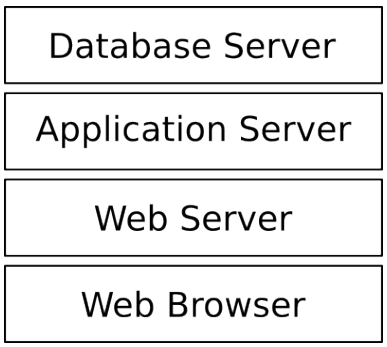

Abbildung 4.1: Typischer Aufbau einer Web-Applikation

4.2 Plattformunabhängigkeit

Web-Applikationen laufen auf einer Vielzahl verschiedener Plattformen, server- und clientseitig. Während auf der Serverseite die jeweilig zu verwendenden Plattformen noch vom/von der BetreiberIn festlegbar sind, sprich welches Betriebssystem und welche Datenbank eingesetzt wird, ist dies auf der Clientseite schon nicht mehr möglich. Daher werden der/die EntwicklerInnen nicht von der Aufgabe befreit, sicher zu stellen, dass die Applikation auf möglichst vielen Web-Browsern korrekt ausgeführt wird.

Die BesucherInnen der Webseite verwenden unterschiedliche Web-Browser auf verschiedenen Betriebssystemen in unterschiedlichen Versionen. Auch können unterschiedliche Plugins und Fonts in unterschiedlichen Versionen installiert sein [14][Foreword]. Dies erhöht den Testaufwand, da bestimmte Funktionen nicht vorhanden sein bzw. anders funktionieren können [7].

Eine weitere Herausforderung von clientseitigem Code stellt die asynchrone und eventbasierte Programmierung dar. Dies erschwehrt nicht nur die Erstellung von Testfällen sondern erhöht auch die möglichen Kombinationen, in der die Events eintreten können. Die Möglichkeit, dass durch eine Aktion, z.B. ein Mausklick auf einen Link, mehrere Events ausgelöst werden können erhöht die Komplexität noch weiter. [14][S. 18]

All diese Probleme werden durch eine zunehmende Verlagerung der Applikations-Logik von der Server- auf die Clientseite[14][S. 13] noch weiter verstärkt.

4.3 Session Modell

Eine weitere Herausforderung stellt das Session Modell von Web-Applikationen dar: viele Web-Applikationen verwenden nur eine Session pro NutzerIn, erlauben aber mehrere gleichzeitige Logins. Werden mehrere Instanzen der Applikation gestartet - z.B. loggt sich der/die NutzerIn auf dem Mobiltelefon und dem Laptop auf der Webseite ein - kann dies zu Synchronisationsproblemen zwischen den einzelnen Instanzen führen: Wird in einer Instanz ein Eintrag gelöscht, kann dieser durch eine fehlerhafte Synchronisation in einer anderen Instanz immer noch existieren und in weiterer Folge zu Fehlern führen. [14][S. 20]

Diese Vielfalt an verschiedenen möglichen Konfigurationen und Herausforderungen erfordert eine neue Herangehensweise an das Thema Software-Test: Die bestehenden Techniken sind „zwar auch notwendig, aber nicht ausreichend, um die Qualität der Applikation sicherzustellen“ [16][S. 18]

5 Projektumfeld

Um die zusätzlichen Herausforderungen beim Testen von Web-Applikationen zu bewältigen, sind zusätzliche Werkzeuge und Testmethoden von Nöten, die vor allem beim Testen der verschiedenen Plattformen und der asynchronen Logik behilflich sind. Dazu wurde im Rahmen eines Praktikums bei onwCloud Inc. eine App erstellt, welche dann mit Testfällen abgedeckt wurde. Bei der App handelt es sich um eine RSS und Atom News Reader App, welche auf der von der Firma entwickelten Plattform, ownCloud[18], aufsetzt.

ownCloud ist eine Open Source Web-Applikations-Plattform, welche oft genutzte Cloud Dienste wie z.B. Kalendar, Kontakte, Dateiupload und Synchronisation zur Verfügung stellt. Sie ist in PHP und JavaScript geschrieben, und kann von jedem/jeder NutzerIn auf seinem/ihren privaten Server installiert und verwendet werden. Im Unterschied zu anderen großen Cloud-Service-Anbietern wie Google und Microsoft liegt die Kontrolle der Daten damit beim/bei der NutzerIn.

5.1 Setup auf der Serverseite

Auf der Serverseite wird PHP in der Version 5.3 eingesetzt. Als Inversion of Control Container kommt Pimple[19] zum Einsatz, als Test-Framwork wird PHPUnit[20] in der Version 3.8.

Für die Erstellung und Einrichtung der virtuellen Maschinen werden Vagrant[21] und Chef[22] verwendet. Vagrant erlaubt das einfache Erstellen und starten von Virtuellen Maschinen. Chef erlaubt durch das Verwenden sogenannter *Cookbooks* und *Recipes* eine einfache Installation und Konfiguration von häufig verwendeten Software-Paketen durch Ruby Skripte.

5.2 Setup auf der Clientseite

Die clientseitige Logik ist mit JavaScript und dem JavaScript Framework AngularJS[23] in Version 1.0 umgesetzt. AngularJS ist ein MV* Framework und erlaubt durch eine Templatesprache eine Trennung von Präsentation und Logik. Da AngularJS bereits einen Inversion of Control Container mitliefert, wird dieser verwendet.

Die JavaScript Tests werden mithilfe von Karma[24](früher Testacular) 0.8 ausgeführt und basieren auf dem JavaScript Unit Test Framework Jasmine 1.3.1 [25]. Die Acceptance und System Tests werden mit Cucumber 1.3.2 [26] erstellt. Cucumber bietet die Möglichkeit, bereits erstellte User-Stories zu parsen und mittels Selenium im Browser auszuführen.

6 Unit Test

Unit Tests werden verwendet, um Klassen, Module oder einzelne Komponenten zu testen. Sie sind Whitebox Tests, sprich die Implementationsdetails sind bekannt[10][S. 26], und werden üblicherweise vom ProgrammiererInnen selbst erstellt.

Da Unit Tests eine Komponente isoliert von anderen Komponenten testet, werden Abhängigkeiten oft mit Mocks (auch Stubs genannt) ersetzt. Diese Mocks implementieren die Methoden und Attribute, von denen die geteste Komponente abhängt und erlauben es, eigene Testwerte zurückzugeben. Viele Test-Framworks wie PHPUnit oder Jasmine liefern eigene Mock-Frameworks mit, mit welchen die Erstellung vereinfacht wird. Ein Mock in PHPUnit sieht in etwa so aus:

```
1  <?php
2  // inside PHPUnit test method
3  $itemDatabaseLayerMock = $this->getMockBuilder('ItemDatabaseLayer')
4        ->disableOriginalConstructor()
5        ->getMock();
6
7  $itemDatabaseLayerMock->expects($this->once())  // expect it to be called
8        ->method('myMethod')  // mock this method
9        ->with($this->equalTo('Parameter 1'))  // expect paramters
10       ->will($this->returnValue(3));  // if method is called, return this
             value
```

6.1 Häufige Probleme

Das Web ist noch recht jung und entwickelt sich rasant. Wurde es früher nur für Webseiten eingesetzt, sollen nun auf einmal komplette Applikationen entstehen. Dies überfordert viele Entwickler und hat oft einen negativen Einfluss auf die Architektur und Umsetzung der Applikation.

Da Unit Tests isolierte Komponenten Tests sind, kann durch es durch eine schlechte Architektur sehr schwierig sein, eine komplette Isolation bestimmter Bereiche zu erreichen. Vor allem folgende Punkte gestalten das Erstellung der Tests als äußerst kompliziert:

- Fehlende Trennung von Logik und Präsentation
- Verwendung globaler Objekte und Variablen

6.1.1 Fehlende Trennung von Logik und Präsentation

Die mit Abstand am beliebtesten clientseitige JavaScript Library ist jQuery. Im Jahr 2011 verwendeten laut Alexa 45% der Top 100.000 Webseiten eine JavaScript Library. Von diesen 45% entfielen 63% auf jQuery. [27][S. 107-108]

jQuery ist sehr einfach zu verwenden [27][S. 110], verleitet jedoch den/die ProgrammiererIn aufgrund ihrer Funktionsweise dazu, Präsentation und Logik stark miteinander zu verweben. Dies

liegt unter anderem daran, dass die meisten jQuery-Methoden einen Selektor als ersten Parameter erwarten. Dieser Selektor funktioniert ähnlich wie ein CSS Selector und selektiert bestimmte Element im DOM. [28]

Folgender Code soll dieses Problem verdeutlichen: Der angeführte Code führt einen AJAX Request aus und gibt abhängig von der zurückgegebenen Datenstruktur einen unterschiedlichen Text aus:

```
<div id="field">
  <div class="info">
    Ausgabe: <span></span>
  </div>
</div>
```

```
(function ($, undefined){
  $(document).ready(function() {

    $.get('http://myurl.com/request.php', function(data) {
      if (data.isValid) {
        $('#field .info span').text('The request was successful');
      } else {
        $('#field .info span').text('The request was not successful');
      }
    });

  });
})(jQuery);
```

Durch das Verwenden des Selektors *#field .info span* ist der JavaScript Code nun abhänig von der HTML Struktur der Webseite. Verändert ein Designer die Struktur des HTML-Codes, beispielsweise um die Ausgabe an einen anderen Ort zu verschieben, besteht die Gefahr, dass der Selektor nicht mehr die gewollten Elemente selektiert und damit der JavaScript-Code nicht mehr funktioniert. Auch serverseitig kann durch eine Verwebung von Präsentation und Logik schwer zu testender Code entstehen.

6.1.2 Verwendung von Globalen Objekten und Variablen

JavaScript macht es einfach, globale Variablen zu verwenden: Es unterstützt nicht nur implizite *Globals*[29][S. 11], sondern regelt den Zugriff auf das DOM über das globale *window* Objekt[29][S. 13]. Dadurch werden ProgrammiererInnen dazu verleitet, globale Variablen und Objekte zu benutzen. Das wiederum erschwert die Testbarkeit, da eine implizite Abhängigkeit ensteht, die für den/die TesterIn nicht sofort offensichtlich ist. Auch wird es schwieriger, den Code in Isolation zu testen.

Jedoch nicht nur JavaScript sondern auch PHP setzt auf globale Variablen: Durch den Einsatz von *Super-Globals* (z.B. $_POST oder $_SERVER) ist es schwer, eine komplette Isolation von globalen Variablen zu erreichen.

6.2 Lösungsansatz

Die vorher aufgezeigten Probleme können mit Hilfe folgender Techniken gelöst werden:

- Trennung der Logik und Präsentation durch Templates
- Vermeidung von globalen Objekten und Variablen
- Verwendung der Software-Patterns Dependency Injection und Inversion of Control, um Abhängigkeiten von Objekten und Funktionen offensichtlich und konfigurierbar zu machen

Um dies zu erreichen wird auf der Clientseite das JavaScript Framework *AngularJS*[23], auf der Serverseite der Inversion of Control Container Pimple[19] und ownCloud Templates eingesetzt.

6.3 Serverseitige Unit Tests

Durch die Verwendung von Pimple lässt sich nun die Konstruktion der Klassen konfigurieren, indem die Abhängigkeiten im Container definiert werden. Auch Super-Globals können hier konfiguriert werden, damit der Code unabhängig von globalen Variablen wird.

Ein Beispiel für die Container-Konfiguration könnte so aussehen:

```
1  <?php
2
3  class Container extends Pimple {
4
5    public function __construct () {
6      // also make Super-Globals injectable
7      $this['Request'] = array(
8        'POST' => $_POST,
9        'GET' => $_GET,
10       'SERVER' => $_SERVER
11       // etc
12     );
13
14     $this['ItemBusinessLayer'] = function ($c) {
15       return new ItemBusinessLayer($c['ItemDatabaseLayer']);
16     }
17
18     // create only one instance ever
19     $this['ItemDatabaseLayer'] = $this->share(function ($c) {
20       return new ItemDatabaseLayer($c['Request']);
21     });
22   }
23
24 }
```

Dazu lassen sich nun einfach Unit Tests erstellen, da die Klasse komplett vom globalen Zustand isoliert wurde. Ein konkreter Unit Test könnte so aussehen:

```
1  <?php
2  class Item {
3    public function isValid() {
4      return true;
5    }
6  }
```

```

// Implementation
class ItemBusinessLayer {

  private $itemDatabaseLayer;

  public function __construct ($itemDatabaseLayer) {
    $this->itemDatabaseLayer = $itemDatabaseLayer;
  }

  public function create($item) {
    if ($item->isValid()) {
      $this->itemDatabaseLayer->save($item);
    }
  }

}
```

```
<?php
// Test
class ItemBusinessLayerTest extends PHPUnit_Framework_TestCase {

  public function testCreateItem () {
    // create a new mock of the item database layer
    $itemDatabaseLayer = $this->getMockBuilder('ItemDatabaseLayer')
      ->disableOriginalConstructor()
      ->getMock();
    $item = new Item();

    $itemBusinessLayer = new ItemBusinessLayer($itemDatabaseLayer);

    $itemDatabaseLayer->expects($this->once())
      ->method('save')
      ->with($this->equalTo($item));

    $itemBusinessLayer->create($item);
  }

}
```

6.4 Clientseitige Unit Tests

Auf der Clientseite wird als JavaScript Framework AngularJS eingesetzt. AngularJS erlaubt das Erstellen eigener HTML-Attribute und HTML-Elemente. Diese werden *Directives* genannt und werden für die Darstellungslogik verwendet.

Mit AngularJS würde das vorige jQuery Beispiel in 6.1 in etwa so aussehen:

```
<div id="field" ng-app="MyApp">
  <div class="info" ng-controller="RequestController">
    Ausgabe: <span>{{ text }}</span>
  </div>
</div>
```

```
(function ($, angular, undefined) {

  // instantiate a new container named MyApp
  angular.module('MyApp', []).

    // create a new controller which can be used in the view
    controller('RequestController', ['$http', '$scope', function($http,
        $scope) {

      $http.get('http://myurl.com/request.php').success(function (data) {
        if (data.isValid) {
          $scope.text = 'The request was successful';
        } else {
          $scope.text = 'The request was not successful';
        }
      });

    }]);

})(jQuery, angular);
```

Das *Scope* dient als Schnittstelle zwischen der Logik und der Präsentation und erlaubt einen bidirektionalen Datenaustausch. Beide Layer sind nun sauber voneinander getrennt: Der JavaScript Code referenziert keine DOM-Elemente mehr und kann so isoliert getestet werden. Außerdem ist es nicht mehr möglich, durch bloßes Verschieben des HTML-Codes Fehler in der JavaScript Logik auszulösen.

AngularJS bietet zudem einen Inversion of Control Container um dynamisch Abhängigkeiten unter den Objekten und Funktionen aufzulösen. Häufig benutzte globale Objekte werden schon fertig konfiguriert mitgeliefert: das *window* Objekt etwa kann durch den Service *$window* verwendet werden. Dadurch können die Abhängigkeiten einfach in den Tests durch Mocks ausgetauscht werden.

Für das oben angeführte Beispiel kann nun ein dazugehöriger Unit Test erstellt werden: Für den AJAX Request wird das *_$httpBackend_* Mock verwendet. Dies erlaubt den asynchronen Request in einen synchronen umzuwandeln. Auch ein neues Scope wird erstellt, um das korrekte Binding zwischen Präsentation und Logik testen zu können.

```
var angular.module('MyApp', ['ngMock']); // inject angular provided mocks

describe('RequestController', function() {

  var $controller,
      $scope,
```

```
7       $http;
8
9     beforeEach(module('MyApp')); // tell inject to use the MyApp container
10
11    beforeEach(inject(function (_$httpBackend_, $controller, $rootScope) {
12      $http = _$httpBackend_; // $http mock provided by angular
13      $scope = $rootScope.$new(); // create a new scope
14      $controller = $controller; // used to instantiate controllers and allows
15                                 // to replace parameters with mocks
16    }));
17
18
19    it('should set the success message if the result is valid', function () {
20      $http.expectGET('http://myurl.com/request.php').respond(200, {
21        isValid: true
22      });
23
24      $controller('RequestController', { // instantiate a new controller
25        $scope: scope
26      });
27
28      $http.flush(); // resolve open AJAX requests
29
30      expect($scope.text).toBe('The request was successful');
31    });
32
33
34  });
```

6.5 Vor- und Nachteile

Zu den Vorteilen von Unit Tests zählen:

- Gute Paralellisierbarkeit
- Sehr schnell ausführbar
- Früh einsetzbar
- Erlaubt die Aufteilung in kleine Teilprobleme
- Zeigt schwer zu verwendende Interfaces auf

Viele dieser Vorteile werden durch den Einsatz von Mocks erreicht, die die konkreten Implementationen ersetzen. Nur Unit Tests alleine reichen jedoch nicht aus, um das Projekt komplett abzudecken, denn Unit Tests[15][S. 52][10][S. 28]:

- testen keine Kommunikation zwichen den Modulen
- testen keine Systemschnittstellen wie z.B. Datenbanken

7 Integration Test

Integration Tests werden verwendet, um die Kommunikation zwischen einzelnen Klassen, Modulen und Komponenten und den Programm-Fluss zu testen. Sie funktionieren ähnlich wie Unit Tests, verwenden jedoch je nach Implementations-Strategie wenig bis keine Mocks und testen weniger Implementationsdetails[15][S. 53-54]. Auf ein Integration Test Quellcode-Beispiel wird deshalb verzichtet.

Durch die vielen möglichen Kombinationen der einzelnen Module gibt es keine allgemein gültige Implementations-Strategie [10][S. 29]. Die zwei häufigsten verwendeten Methoden zur Erstellung der Testfälle sind:

- Nicht Inkrementelle Testfallerstellung
- Inkrementelle Testfallerstellung

7.1 Nicht Inkrementelle Testfallerstellung

Wird die nicht inkrementelle Testfallerstellung genutzt, werden die nicht vorhandenen Module mit Mocks ersetzt. Diese werden am Schluss dann durch die aktuelle Implementierung ausgetauscht.

Dies mag vor allem am Anfang sehr aufwendig erscheinen, erlaubt jedoch schon früh die Kommunikation zwischen den einzelnen Bereichen zu testen. Auf diese Weise kann sehr schnell ein Überblick über den Implementationsstand der Software erlangt werden. Diese Methode eignet sich daher besonders für Agile Software-Entwicklung. [15][S. 54-59]

Typische Strategien der nicht inkrementellen Testfallerstellung beinhalten[15][S. 54]:

- Big-Bang Testing: Möglichst viele fertige Module werden in einen Test integriert
- Transaction Testing: Alle Module, die zu einer Transaktion gehören, werden in einem Test vereint

7.2 Inkrementelle Testfallerstellung

Bei der inkrementellen Testfallerstellung werden wenig bis gar keine Mocks erstellt. Stattdessen wird immer versucht, auf den bisherig erstellten Modulen aufzubauen. Dadurch werden die zuerst erstellten Module am Besten getestet.[15][S. 54-59]

Dies eignet sich vor allem dann, wenn es bestimmte Module mit einer hohen Komplexität und Fehlerdichte gibt und das Erstellen von Mocks sehr schwer und aufwendig ist. [15][S. 59]

Typische Vertreter der inkrementellen Testfallerstellung sind[15][S. 54-55]:

- Top-Down: Die Module auf dem höchsten Level werden zuerst erstellt
- Bottom-Up: Die Module auf dem niedrigsten Level werden zuerst erstellt

- Hardest First: Die schwierigsten oder kritischsten Module werden zuerst erstellt
- Thread Testing: Es wird versucht, einen zusammenhängenden Strang an Funktionen zu erstellen, welcher dann komplett getestet werden kann
- Availability Testing: Nur Module, welche schon durch Unit Tests getestet worden sind, werden integriert

7.3 Vor- und Nachteile

Integration Tests haben folgende Vorteile:

- Testen die Kommunikation und das Zusammenspiel der Komponenten
- Testen wichtige Szenarien der einzelnen Module

Zu den Nachteilen gehören:

- Langsamer als Unit Tests, da ganze Sub-Systeme getestet werden
- Kein Test der Systemschnittstellen

8 System Test

System Tests werden verwendet, um das Zusammenspiel zwischen dem System und der Applikation zu testen. Sie beinhalten sowohl funktionale (z.B. Installations Tests) als auch nicht funktionale Tests (z.B. Security, Usability oder Performance Tests)[10][S. 30]. Diese Tests „werden meistens nicht mehr von den Entwicklern selbst durchgeführt, sondern von unabhängigen Tester bzw. Test-Teams.“[11][S. 17]

Da die Tests auf einem dem/der Kunden/Kundin ähnlichen System ausgeführt werden, ist es gut, den/die Kunden/Kundin in den Test zu involvieren. Dies gestaltet sich oft als schwierig, da der/die Kunde/Kundin meist nicht die nötige Expertise mitbringt, um die exakten Anforderungen zu spezifizieren und Testfälle zu erstellen. Als Lösungsmöglichkeit bietet sich hier die Zusammenarbeit mit einem qualifizierten Team des/der Kunden/Kundin [15][S. 60]

8.1 Lösungsansatz

Um die Web-Applikationen sowohl server- als auch clientseitig testen zu können, muss eine entsprechende Möglichkeit bestehen, die verschiedenen System zu simulieren. Dazu werden verschiedenen virtuelle Maschinen mit den zu testenden Systemen erstellt. Dann werden die automatisierbaren Tests auf den jeweiligen Systemen ausgeführt, nicht automatisierbare Tests wie etwa die Korrektur der Übersetzungen wird an Test-Teams ausgelagert.

8.2 Serverseitige System Tests

Es ist schwierig, die Tests in server- und clientseitige Tests zu spalten, da das ganze System getestet wird. Clientseitige Tests testen somit auch serverseitige Funktionalitäten. Bestimmte Tests wie Performance Tests müssen z.B. in Kombination durchgeführt werden.

Die serverseitigen Tests reduzieren sich deshalb auf das Erstellen und Einrichten der jeweiligen VMs für den serverseitigen Code und auf das Testen von[15][S. 60-66]:

- Security: Statische Code Analysen können mit Tools durchgeführt werden; Security Audits von Fachkräften werden durchgeführt
- Konfiguration: Mit einem Skript werden verschiedene Konfigurationen getestet
- Kompatibiltät der Applikation mit dem Server Betriebssystem und den serverseitigen Applikationen: Mit Vagrant werden virtuelle Maschinen der unterstützten serverseitigen Betriebssysteme erstellt; die zu testenden Web-Server und Datenbanken werden mit Chef Cookbooks eingerichtet
- Recovery Funktionen: Falls verfügbar, wird dies mit einem Script getestet
- Performance und Speicherverbrauch: Mit einem Skript werden verschiedene, gleichzeitige Anfragen an den Server erstellt; der Arbeitsspeicher wird auf das spezifizierte Minimum reduziert

- Installation: Wird durch den clientseitigen Installationsdialog server- und clientseitig getestet
- Zuverlässigkeit: Kommt darauf an, ob die Anforderungen testbar formuliert sind, z.B. mit durchschnittlicher Fehlerrate pro Stunde

8.3 Clientseitige System Tests

Die clientseitigen System Tests werden auf unterschiedlichen Web-Browsern ausgeführt. Auch zusätzliche VMs für Windows und Mac OS X sind notwendig um verschiedene Versionen des Internet Explorers respektive Safari zu testen. Die restlichen Tests können unter Linux VMs durchgeführt werden.

8.3.1 JavaScript Tests

Der JavaScript Testrunner Karma erlaubt die Konfiguration verschiedener Web-Browser, unter welchen die JavaScript Unit Tests ausgeführt werden können. Dadurch kann sichergestellt werden, dass der JavaScript Code auf allen Systemen wie gewünscht funktioniert.

8.3.2 User-Interface Tests

Für die User-Interface Tests wird Cucumber verwendet. Die gewünschten User-Stories werden in Gherkin (der User-Story Grammatik) spezifiziert. Danach lassen sich die Methoden erstellen, die die User-Stories parsen und in den verschiedenen Browsern durchführen.

Folgendes Beispiel loggt sich ein und überprüft, ob bei einem Klick auf einen Button das entsprechende Formular sichtbar wird:

```
# encoding: utf-8
Feature: create_new
  In order to start using the news rss reader
  As a user
  I want to be able to add feeds and folders

  Background:
    Given I am logged in
    And I am in the "news" app

  Scenario: show add website dialogue
    When I click on the add new button
    Then I should see a form to add feeds and folders
```

Die User-Story lässt sich nun mit folgenden Schritten parsen und testen:

```
Given (/^I am logged in$/) do

  # be sure to use the right browser session
  Capybara.session_name = 'test'

  # logout - just to be sure
  visit '/index.php?logout=true'
  visit '/'
  fill_in 'user', with: 'test'
  fill_in 'password', with: "test"
  click_button 'submit'

```

```
13    # if visibility is not set to false it will fail
14    page.should have_selector('a#logout', :visible => false)
15  end
16
17  Given (/^I am in the "([^"]+)" app$/) do |app|
18    visit "/index.php/apps/#{app}/"
19    page.should have_selector('a#logout', :visible => false)
20  end
21
22  When (/^I click on the add new button$/) do
23    click_link 'Add Website'
24  end
25
26  Then (/^I should see a form to add feeds and folders$/) do
27    selector = "//*[contains(@class,\"add-new-popup\")]"
28    page.should have_selector(:xpath, selector)
29  end
```

8.4 Vor- und Nachteile

Durch die Simulation der verschiedenen Umgebungen ergeben sich folgende Vorteile [15][S. 60-66]:

- Sicherheit, dass die Applikation auf unterschiedlichen Plattformen funktioniert
- Auch nicht funktionale, aber manchmal projektkritische Anforderungen wie Performance testbar

Da für die verschiedenen Plattformen jeweils Testumgebungen gestartet werden müssen, entstehen dadurch vor allem folgende Nachteile:

- Lange Startzeit der Testumgebungen
- Lange Laufzeit der Tests
- Dadurch langsame Entwicklung der Tests
- Fehler durch großen Bereich schwer lokalisierbar
- Sehr schwierig die exakte Testumgebung zu reproduzieren
- Schwer automatisierbar, vor allem bei nicht funktionalen Tests wie Usability Tests

9 Acceptance Test

Beim Acceptance Test wird die Applikation auf die Anforderungen des/der Kunden/Kundin geprüft [15][S. 66]. Der Test ist erforderlich, um eine erfolgreiche Abnahme durch den/die Kunden/Kundin zu erreichen. Da es vor allem im Interesse des/der Kunden/Kundin ist, eine wie im Vertrag vereinbarte funktionierende Software zu erhalten, ist er/sie meistens der/die HaupttesterIn.

9.1 Bestandteile

Zu den verschiedenen Tests zählen [10][S. 33-34]:

- „Test auf vertragliche Akzeptanz“
- „Test der Benutzbarkeit“
- „Akzeptanz durch den Systembetreiber“
- „Feldrest (Alpha- und Betatest)“

Um das Risiko einer verweigerten Abnahme im späten Stadium zu vermindern, können die Acceptance Tests bereits während der Entwicklung mit dem/der Kunden/Kundin zusammen durchgeführt werden, was sich vor allem in einem agilen Software-Projekt anbietet. [11][S. 17]

9.2 Lösungsansatz

Dadurch, dass der/die Kunde/Kundin in diesem Fall der/die HaupttesterIn der Software ist, gestaltet es sich schwierig, diesen Prozess zu automatisieren. Es empfiehlt sich, regelmäßige Meetings mit dem/der Kunden/Kundin abzuhalten, damit das Projekt früh genug an eventuell eintretbare Probleme angepasst werden kann. Außerdem bietet es sich an, die während des System Test entstandenen User-Stories zu integrieren. Wurden diese vom oder mit Hilfe des/der Kunden/Kundin erstellt, kann der/die HerstellerIn damit argumentieren, dass die verlangten Features komplett umgesetzt wurden.

10 Zusammenfassung

Das Web bietet sich immer mehr als Applikations-Plattform an, erfordert jedoch auch eine Anpassung der Testmethoden, um qualitativ hochwertige Applikationen zu erstellen. In dieser Arbeit wurde anhand eines Projektes Beispiele zu den jeweiligen Testarten des V-Modells erabeitet, welche die zusätzlichen Herausforderungen adressieren:

- Der Modularität, Fragmentierung und dem Session Modell kann mit einer Anpassung der System Tests begegnet werden, indem mittels virtuellen Maschinen unterschiedliche Server- und Clientplattformen erstellt werden. Auf diesen werden dann die Tests gestartet.
- JavaScript Test-Frameworks erlauben, asynchrone Methoden zu testen; zudem können durch den Einsatz von Mocks asynchrone Methoden synchron gemacht werden.
- Sowohl mit server- als auch mit clientseitigen Frameworks lässt sich die Trennung von Logik und Präsentation umsetzen, um das Testen zu vereinfachen.

Zusätzlich ist es wichtig, die richtigen Testmethoden abzuwägen, da sie unterschiedliche Vor- und Nachteile mit sich bringen: Je höher das Level der Tests (Integration und System Test), desto länger dauert es, die Testfälle zu erstellen und desto schwieriger wird es, die Fehler zu finden. Je niedriger das Level (Unit Test), desto näher an der aktuellen Implementierung ist der Test-Code und desto öfter muss dieser bei Änderungen angepasst werden.

Schlussendlich muss immer abgewägt werden, wieviel Zeit und Geld in Tests investiert werden, und vor allem, welche Tests umgesetzt werden, um das Projekt nicht unwirtschaftlich werden zu lassen.

Literaturverzeichnis

[1] "Android Developer Website," Website, Google, 2013, http://developer.android.com/ [Zugang am 20.05.2013].

[2] "iOS Dev Center," Website, Apple, 2013, https://developer.apple.com/devcenter/ios/index.action [Zugang am 20.05.2013].

[3] H. Blodget, "The future of mobile," 2011.

[4] "PhoneGap Website," Website, Adobe, 2013, http://phonegap.com/ [Zugang am 20.05.2013].

[5] E. T. u. K. M. Koeder, Marco Josef, "Mobile ecosystems in global transition: Driving factors of becoming a mobile ecosystem enabled. A comparison between the US and Japan," 2012.

[6] M. Crowley, *Pro Internet Explorer 8 & 9 Development*. Apress, 2010.

[7] "Can I use... Compatibility tables for support of HTML5, CSS3, SVG and more in desktop and mobile browsers," Website, 2013, http://caniuse.com/ [Zugang am 20.05.2013].

[8] G. O. u. M. M. Stefan Frei, Thomas Duebendorfer, "Understanding the web browser threat: Examination of vulnerable online web browser populations and the ïnsecurity iceberg‖," *ETH Zurich Tech Report Nr. 288*, 2008.

[9] M. Schmidt, "An Empirical Investigation of Human-Based and Tool-Supported Testing Strategies," Master's thesis, Technische Universität Wien, 2011.

[10] F. Richter, "Betriebliche Einführung von automatischen Softwaretests unter Berücksichtigung ökonomischer und technischer Rahmenbedingungen," Master's thesis, Technische Universität Wien, 2007.

[11] M. Oberreiter, "Evaluierung, Konzeption und Härtung eines Testprozesses für automatisierte Regressionstests in einem mittelgroßen Entwicklungsszenario," Master's thesis, Technische Universität Wien, 2011.

[12] S. Nageswaran, "Test effort estimation using use case points," *Quality Week 2001, San Francisco, California*, 2001.

[13] A. Waser, "Test Automation A Case Study," Master's thesis, Technische Universität Wien, 2002.

[14] H. Q. Nguyen, *Testing Applications on the Web*. Wiley Computer Publishing, 2001.

[15] C. Dobritzhofer, "Towards Test Methodologies for Developing Large Systems," Master's thesis, Technische Universität Wien, 1994.

[16] S. Avci, "Evaluieren von automatisierten Tests bei Web-Applikationen," Master's thesis, Technische Universität Wien, 2010.

[17] D. W. Hoffmann, *Software-Qualität*. Springer, 2008.

[18] "ownCloud Website," Website, ownCloud Inc., 2013, http://owncloud.org/ [Zugang am 20.05.2013].

[19] "Pimple Documentation," Website, SensioLabs, 2013, http://pimple.sensiolabs.org/ [Zugang am 20.05.2013].

[20] "PHPUnit Documentation," Website, Sebastian Bergmann, 2013, http://phpunit.de/manual/current/en/index.html [Zugang am 20.05.2013].

[21] "Vagrant Documentation," Website, HashiCorp, 2013, http://www.vagrantup.com/ [Zugang am 20.05.2013].

[22] "Chef Documentation," Website, Opscode, 2013, http://www.opscode.com/chef/ [Zugang am 20.05.2013].

[23] "AngularJS," Website, Google, 2013, http://angularjs.org/ [Zugang am 20.05.2013].

[24] "Karma Documentation," Website, Google, 2013, http://karma-runner.github.io/0.8/index.html [Zugang am 20.05.2013].

[25] "Jasmine API Documentation," Website, Pivotal Labs, 2013, http://pivotal.github.io/jasmine/ [Zugang am 20.05.2013].

[26] "Cucumber Website," Website, Aslak Hellesøy, 2013, http://cukes.info/ [Zugang am 20.05.2013].

[27] M. B. u. F. K. Rein Smedinga, *SC@RUG 2011 proceedings*. Bibliotheek der R.U., 2011.

[28] "jQuery API Documentation," Website, The jQuery Foundation, 2013, http://api.jquery.com/category/selectors/ [Zugang am 20.05.2013].

[29] S. Stefanov, *JavaScript Patterns*. O'Reilly, 2010.

Abbildungsverzeichnis

Abkürzungsverzeichnis

www	World Wide Web
CSS	Cascading Style Sheets
AJAX	Asynchron JavaScript And XML
HTML	HyperText Markup Language
DOM	Document Object Model
W3C	World Wide Web Consortium
MV*	Model View - View Model
VM	Virtual Machine
PHP	PHP Hypertext Preprocessor
PC	Personal Computer