BACHELORARBEIT

im Studiengang Bachelor Informatik

Software-Test von Web-Applikationen

Ausgeführt von: Bernhard Posselt
Personenkennzeichen: 1010257029

Begutachter: MSc Benedikt Salzbrunn

Wien, 26. Mai 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 aufzeigen und Anpassungen und Anwendung der vier Testformen des V-Modells: Unit Test, Integration Test, System Test und Acceptance Test.

Schlagwörter: Webapplikationen, automatisierte Tests, Webtest

Abstract

Creation and execution of automatic web application tests is different from tests of classic applications. A more complex infrastructure and modularisation require 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: web applications, automatic tests, webtest

Inhaltsverzeichnis

1 Einführung

Der Markt für elektronische Geräte hat in den letzten Jahren ein rasantes Wachstum erlebt, vor allem im Bereich der mobilen Geräte. Im Jahr 2011 wurden erstmals mehr mobile Geräte verkauft als PCs 1.1. Aufgrund des starken Wachstums dehnt sich der Markt für Software-Applikationen auch auf mobile Geräte aus, welche zum Großteil andere Betriebssysteme und Applikations-Frameworks verwenden als traditionelle Computer 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.

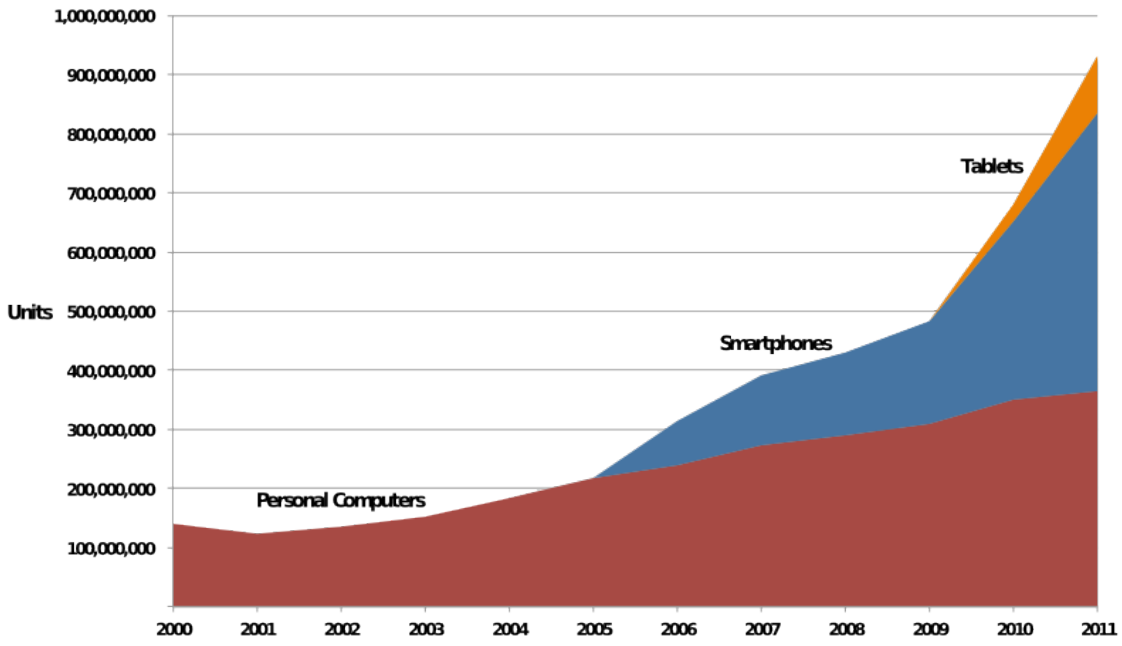

Abbildung 1.1: Entwicklung der PC und Mobilgerätverkäufe [3][S. 6]

Da viele dieser Mobilgeräte über einen Web-Browser verfügen, wird das Web als Applikations-Plattform immer attraktiver. Zudem erlauben Frameworks wie PhoneGap [4] dem/der EntwicklerIn eine plattformübergreifende Mobil-Applikation auf Web-Technologie-Basis zu erstellen, welche sich nahtlos in das System integrieren.

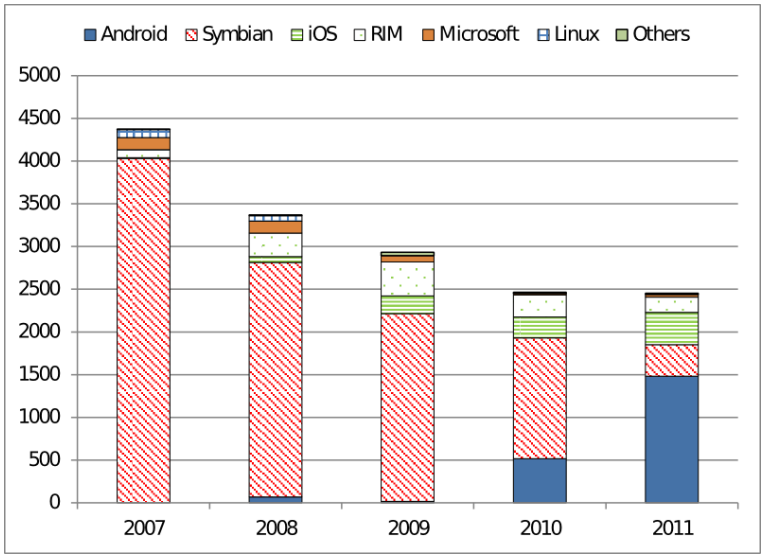

Abbildung 1.2: Marktanteil mobiler Betriebssysteme [5][S. 22]

1.1 Problemstellung

Aufgrund der Vielfalt an unterschiedlichen Web-Browsern erhöht sich jedoch auch der Testaufwand der Applikation. Die Browser implementieren die vom W3C vorgeschlagenen Standards ab und zu anders und in unterschiedlicher Geschwindigkeit. Nicht selten kommt es vor, dass gewisse Funktionen nur in einer eingeschränkten Auswahl von Web-Browsern voll funktionsfähig sind und zusätzliche Anpassungen erfordern. [6]

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. [7][S. 3]

All dies zwingt Web-EntwicklerInnen 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.

Diese Bachelorarbeit soll die zusätzlichen Herausforderungen beim Testen von Web-Applikationen aufzeigen. Des Weiteren soll sie einen Überblick über mögliche Anwendungen und Anpassungen der vier häufigsten Testarten - den Unit Tests, Integration Tests, System Tests und Acceptance Tests - im Bereich der Web-Applikationen geben.

Dazu wird anhand eines Projektes eine Test-Strategie erarbeitet und Beispiele für Tests gegeben.

1.3 Aufbau

Kapitel 2 behandelt den Sinn und die Grenzen des Software-Tests und wie er sich in den Entwicklungsprozess einbinden lässt. Außerdem wird anhand des V-Modells versucht, den Zusammenhang zwischen Software-Test und Projektablauf aufzuzeigen.

In Kapitel 3 wird auf die Unterschiede zwischen klassischen Applikationen und Web-Applikationen eingegangen. Dadurch sollen zusätzliche Probleme, die sich durch das Applikations-Modell ergeben, aufgezeigt werden.

Die Planung und der Ablauf des Testprozesses durch die Verwendung eines Testplanes Kapitel 4 beschrieben.

Kapitel 5 beschäftigt sich näher mit dem praktischen Einsatz von client- und serverseitigen Unit Tests. Es werden für beide Seiten Beispiele eines Unit Tests gezeigt und auf klassiche Probleme bei clientseitigen Unit Tests und deren Lösungen eingangen.

Das Thema des Integration Tests wird in Kaptitel 6 behandelt. Außerdem wird auf die verschiedenen Test-Ansätze eingegangen.

Kaptitel 7 behandelt den Systemtest. Es wird anhand von Beispielen gezeigt, wie die verschiedenen Server- und Client-Plattformen automatisch getestet werden können.

Kapitel 8 beschäftigt sich mit dem Thema des Acceptance Tests und wie dieser bei Web-Applikationen umgesetzt werden kann.

2 Warum testen

Keine Applikation ist fehlerfrei[8][S. 9]. Diese Fehler führen nicht nur zu unzufriedenen Kunden, 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.“[9][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 [10][S. 10].

Je früher diese Fehler entdeckt werden, desto kostengünstiger können diese beseitigt werden [9][S. 17]. Daher ist es wichtig, möglichst früh mit dem Testen zu beginnen und den Software-Entwicklungsprozess dementsprechend anzupassen [9][S. 16].

2.1 Verschiedene Testarten

Laut dem V-Modell 2.1 ist der Software-Test kein seperater Abschnitt im Projektplan. Vielmehr ist er ein Prozess, der die verschiedenen Entwicklungsabschnitte ergänzt [9][S. 23]. Das V-Model unterscheidet zwischen vier verschiedenen Testarten:

- Unit Test: Wird paralell zur Implementation erstellt
- Integration Test: Wird erstellt, um den Designprozess zu überprüfen
- System Test: Testet die konkreten Anforderungen an die Software
- Acceptance Test: Testet die Anforderungen des Kunden

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 Implementation 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. [12][S. 2]

Zudem ist eine Abschätzung riskanter Bereiche möglich, die durch eine erhöhten Testbedarf bestimmter Bereiche offensichtlich wird. [13][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. [14][S. 61]

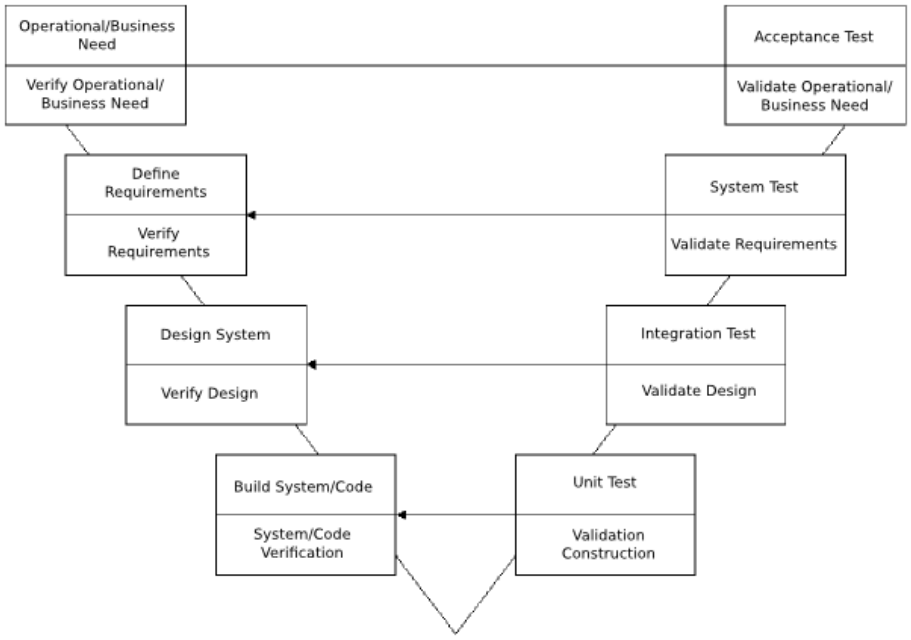

Abbildung 2.1: Schematische Darstellung des V-Modells [11][S. 3]

2.3 Automatisierten Tests

Da die vorhandenen 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. [12][S. 22-23]

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

Erstens 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. [10][S. 12]

Zweitens muss das Personal über eine dementsprechende Ausbildung verfügen, die Tests zu automatisieren. [15][S. 37]

Drittens eignet sich nicht jede Testart zur vollkommenen Automatisierung. Währen Unit-Tests und Integration-Tests sehr gut automatisiert werden können, ist es schon schwieriger System- und Acceptance Tests 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. [10][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. [16][S. 243].

Die implementierten Testfälle müssen auch regelmäßig gewartet und aktualisiert werden, um ihre Effektivität zu gewährleisten, denn diese nimmt auf Dauer ab. Es ensteht eine sogenannte „Testresistenz“[10][S. 12], die daraus resultiert, dass die bestehenden Tests nur bekannte Fehlerfälle abdecken und keine neuen, mögliche Fehlerfälle berücksichtigen. [10][S. 12-13]

3 Testplan

Mit dem Erstellen des Projektplanes sollte gleichzeitig auch ein Testplan erstellt werden [15][S. 24]. Dieser erlaubt, den Testprozess näher zu spezifizieren und somit den erforderlichen Aufwand besser abschätzbar zu machen [14][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. [15][S. 26]

3.1 Inhalt

Im Testplan werden unter anderem folgende Punkte behandelt [12][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 werden 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[16][S. 244] und der Testprozess zudem teuer ist [10][S. 24], ist es wichtig, Testfälle heraus zu arbeiten, zu priorisieren und effektiv zu verteilen. Auch Endkriterien sollten für die Tests veranschlagt und eine optimale Teststrategie 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 Teststrategien 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 Implementation immer konkreter werden und somit auch neue, mögliche Fehlerfälle enstehen können.[10][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 können.

Manche Tests haben eine sehr hohe Laufzeit. Um den Ablauf daher zu beschleunigen, wird ein Smoke-Tests 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. [10][S. 26]

3.3 Vorteile eines Testplans

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

- Fehlende Testfälle können schnell erkannt werden [14][S. 18]
- Redundante Testfälle werden identifziert, Testfälle können zusammegelegt werden um Zeit zu sparen [13][S. 34]
- Verschiedene Testbereiche können an mehrere Personen mit unterschiedlichen Fachkenntnissen verteilt werden um Personalengpässe zu vermeiden [14][S. 19]
- Ineffiziente Test-Tools und Test-Strategien können erkannt und verbessert werden [10][S. 25]
- Geschäftskritische Bereiche und solche mit einer erhöhten Fehleranfälligkeit werden sichtbarer [13][S. 34]. Dies ist vor allem wichtig, da Fehler ungleich verteilt sind und in bestimmten Bereichen häufiger vorkommen als in anderen [10][S. 12]
- Sinnlose Testfälle und Testbereiche können aussortiert werden
- Abhängigkeiten der Testfälle untereinander und der Implementation werden transparent und erlauben schnell auf Änderungen zu reagieren[12][S. 4]

4 Unterschiede zu klassischer Software

Eine Web-Applikationen 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 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. [13][Foreword]

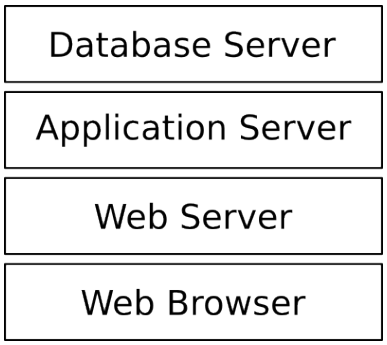

Abbildung 4.1: Typischer Aufbau einer Web-Applikation

4.2 Unterstützung mehrerer Plattformen

Web-Applikationen laufen auf einer Vielzahl verschiedener Plattformen, Server- und Clientseitig. Dies ermöglicht dem/der EntwicklerIn Zeit und Aufwand zu sparen, befreit ihn/sie jedoch nicht von der Aufgabe, sicher zu stellen, dass die Applikation auf möglichst vielen Plattformen korrekt ausgeführt wird.

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.

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 [13][Foreword]. Dies erhöht den Testaufwand, da bestimmte Funktionen nicht vorhanden sein bzw. anders funktionieren können [6].

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. [13][S. 18]

All diese Probleme werden durch eine zunehmende Verlagerung der Applikations-Logik von der Server- auf die Clientseite[13][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 andere Instanz immer noch existieren und in weiterer Folge zu Fehlern führen. [13][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“ [15][S. 18]

5 Projektumfeld

TBD

6 Unit Test

Unit Tests werden verwendet, um Klassen, Module oder einzelne Komponenten zu testen. Sie sind Whitebox Tests, sprich die Implementationsdetails sind bekannt[9][S. 26], und werden üblicherweise vom Programmierer selbst erstellt.

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, da sie keine Kommunikation zwischen den einzelnen Komponenten testen. Auch Tests für verschiedene Datenbanken können dadurch nicht abgedeckt werden. [14][S. 52][9][S. 28]

6.1 Serverseitige Unit Tests

Serverseitige Unit Tests unterscheiden sich nicht nur anhand des eingesetzten Test-Frameworks, sondern auch anhand der gewählten Programmiersprache.

In dieser Arbeit wird das Test-Framework PHPUnit eingesetzt. Ein konkreter Unit Test könnte so aussehen:

```
1  <?php
2  class Item {
3    public function isValid() {
4      return true;
5    }
6  }
7
8  // Implementation
9  class ItemBusinessLayer {
10
11   private $itemDatabaseLayer;
12
13   public function __construct ($itemDatabaseLayer) {
14     $this->itemDatabaseLayer = $itemDatabaseLayer;
15   }
16
17   public function create($item) {
```

```
18      if ($item->isValid()) {
19        $this->itemDatabaseLayer->save($item);
20      }
21    }
22
23  }
```

```
1   <?php
2   // Test
3   class ItemBusinessLayerTest extends PHPUnit_Framework_TestCase {
4
5     public function testCreateItem () {
6       // create a new mock of the item database layer
7       $itemDatabaseLayer = $this->getMockBuilder('ItemDatabaseLayer')
8         ->disableOriginalConstructor()
9         ->getMock();
10      $item = new Item();
11
12      $itemBusinessLayer = new ItemBusinessLayer($itemDatabaseLayer);
13
14      $itemDatabaseLayer->expects($this->once())
15        ->method('save')
16        ->with($this->equalTo($item));
17
18      $itemBusinessLayer->create($item);
19    }
20
21  }
```

6.2 Clientseitige Unit Tests

Im Gegensatz zum serverseitigen Code gibt es auf der Clientseite nur eine mögliche Programmiersprache, um Logik plattformübergreifend zu implementieren: *JavaScript*. Für JavaScript gibt es mehrere Test-Frameworks, unter anderem Jasmine [17], QUnit [18] und Mocha [19]. In dieser Arbeit wird für die Implementation der clienseitigen Unit Tests Jasmine eingesetzt.

6.2.1 Häufige Probleme bei clientseitigem Code

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 resultiert oft in halbgaren Architekturen und schlampigen Umsetzungen.

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. [20][S. 107-108]

jQuery ist sehr einfach zu verwenden [20][S. 110], verleitet jedoch den/die ProgrammierIn aufgrund der Funktionsweise dazu, Präsentation und Logik stark miteinander zu verweben. Dies liegt unter anderem daran, dass die meisten jQuery-Methoden einen Selektor als ersten Parameter erwartet. Dieser Selektor funktioniert ähnlich wie ein CSS Selector und selektiert bestimmte Element im DOM. [21]

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.

Verwendung von Globalen Objekten und Variablen

JavaScript macht es einfach, globale Variablen zu verwenden: Es unterstützt nicht nur implizite *Globals*[22][S. 11], sondern regelt den Zugriff auf das DOM über das globale *window* Objekt[22][S. 13].

Dadurch werden ProgrammiererInnen dazu verleitet, globale Variablen und Objekte zu benutzen. Das wiederum erschwert die die Testbarkeit, da eine implizite Abhängigkeit ensteht, die für den Tester nicht sofort offensichtlich ist. Auch wird es schwieriger, den Code in Isolation zu testen.

6.2.2 Lösungsansatz

Die vorher aufgezeigten Probleme können mit Hilfe folgende 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 das JavaScript Framework *AngularJS*[23] eingesetzt.

Trennung von Logik und Präsentation

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 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 so kann so isoliert getestet werden. Außerdem ist es nicht mehr möglich, durch bloßes Verschieben des HTML-Codes Fehler in der JavaScript Logik auszulösen.

Injecten von Globalen Objekten und Variablen

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* injected werden. Dadurch können die Abhängigkeiten einfach in den Tests durch Mocks ausgetauscht werden.

Unit Test

Für das oben aufgeführte Beispiel kann nun ein dazugehöriger Unit Test erstellt werden.

Für den AJAX Request wird das _*$httpBackend*_ Mock injected. 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.

```
1  var angular.module('MyApp', ['ngMock']); // inject test mocks into the
       container
2
3  describe('RequestController', function() {
4
5    var $controller,
6        $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
           instance
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 });
```

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. [14][S. 53-54]

Auf ein Integration Test Beispiel wird deshalb verzichtet.

Durch die vielen möglichen Kombinationen der einzelnen Module gibt es keine allgemein gültige Implementations-Strategie [9][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 Implementation 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. [14][S. 54-59]

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.[14][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. [14][S. 59]

8 System Test

TBD

9 Acceptance Test

TBD

10 Zusammenfassung

TBD

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] "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].

[7] 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.

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

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

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

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

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

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

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

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

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

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

[18] "QUnit API Documentation," Website, The jQuery Foundation, 2013, http://qunitjs.com/ [Zugang am 20.05.2013].

[19] T. Holowaychuk, "Mocha API Documentation," Website, 2013, http://visionmedia.github.io/mocha/ [Zugang am 20.05.2013].

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

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

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

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

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