FOM Hochschule für Oekonomie & Management Essen
Standort Duisburg

Berufsbegleitender Studiengang
Wirtschaftsinformatik, 7. Semester

Bachelor Thesis
zur Erlangung des Grades eines
Bachelor of Science (B. Sc.)

über das Thema

Testautomatisierung von PHP-Projekten dargestellt an einem Shopware Projekt

Betreuer: Dipl.-Kfm. Henning Mertes

Autor: Thomas Eiling
Matrikelnr.: 313489
Eichsfelder Weg 4b
46359 Heiden

Abgabe: 10. Dezember 2016

Inhaltsverzeichnis

Abkürzungsverzeichnis

ASCII ..... American Standard Code for Information Interchange

CPU ..... Central Processing Unit

CSV ..... Comma Separated Value

ECSV ..... Encapsulated Comma Separated Value

ERP ..... Enterprise-Resource-Planing

GUI ..... Graphical user interface

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Problemstellung

Ausgehend von einer Shopware ERP Schnittstelle, werden die gängigen Testautomatisierungen im PHP Umfeld in dieser Bachelor-Thesis

1.2 Zielsetzung

Kleiner Reminder für mich in Bezug auf die Dinge, die wir bei der Thesis beachten sollten und LaTeX-Vorlage für die Thesis.

1.3 Grenzen und Bedingungen der Arbeit

1.4 Aufbau der Arbeit

2 Theoretische Grundlagen

Siehe auch Wissenschaftliches Arbeiten[1]. Damit sollten alle wichtigen Informationen abgedeckt sein ;-)

2.1 Codequalität

Codequalität ist ein Teilaspekt der Softwarequalität und somit essentiell für die Softwareentwicklung. Folgende Parameter beschreiben, was die Qualität eines Programmcodes ausmacht:

1) Lesbarkeit

- Der Programmcode ist für einen Entwickler einfach zu lesen und kann diesen ohne großes einarbeiten in der der Dokumentation verstehen
- Die Benennung der Klassen-, Methoden- und Variablen haben ein einheitliches Schema und nutzen Idealerweise einen Standard wie zum Beispiel den PSR-2 Standard.
-

2) Testbarkeit

- Der geschriebene Quellcode sollte so aufgebaut sein, dass dieser automatisiert in kleinen Modulen/Units getestet werden kann sofern nicht schon Test- beziehungsweise Verhaltensgetrieben entwickelt wurde.

3) Variabilität

- Der Quellcode muss einfach erweiter- und anpassbar sein um die Möglichkeit für neue Funktionen in der Software nicht zu verwehren.

4) Flexibilität

- Abhängigkeiten im eigenen Programmcode-Fundament und den Implementierungen sollten gering gehalten werden.
- Es sollten keine festen Konstanten im Quellcode vorhanden sein, z.B. 19 % Mehrwertsteuer.
- Statisch programmierte Voraussetzungen über Datenstrukturen, Klassen oder Datengröße machen den Quellcode schwieriger adaptierbar.

[1] Vgl. Balzert et al. (2008), Seite 1.

5) Performance

- Die Verwendung der vorhandenen Systemressourcen durch das entwickelte Programm sollte unnötige Verwendungen der CPU oder dem Hauptspeicher vermeiden.

Da diese Parameter eine große Auswirkung auf die Qualität der entwickelten Software hat, müssen diese während des gesamten Prozesses kontrolliert werden. Idealerweise geschieht dies durch geeignete Tools wie z.B. !"!"!"!"!"!"!"!

Um die bestmögliche Codequalität zu garantieren, existieren verschiedene Ansätze den Softwareentwickler zu unterstützen. Schon bei der Analyse der Anforderungen und dem Design für eine Funktion, versuchen verhaltens- und testgetriebene Ansätze den Entwickler entscheidend zu unterstützen.

2.2 Testgetriebene Entwicklung

Wenn eine neue Funktionalität in einem Programm implementiert bzw. eine Funktionalität angepasst und erweitert werden soll, wie stellt man sicher, dass es im Nachhinein zu keinerlei Problemen kommt? Die Funktionalität per Hand zu testen ist aus mehreren Gründen nicht vorteilhaft.

Die testgetriebene Entwicklung versucht dieses Problem zu beheben. Um dieses Ziel zu erreichen, wird zu Erst mit einem Test die Funktionalität spezifiziert. Nach der Fertigstellung des Tests wird der Programmcode entwickelt. Das führt dazu, dass nun die komplette Funktionalität überprüft werden kann. Wenn unerwünschte Seiteneffekte entstehen, die auf Grund einer Codeänderung an einer anderen Stelle auftreten, können sie nun durch die Tests herausgefunden und dadurch behoben werden.

2.3 Verhaltensgetriebene Entwicklung

BDD wurde ursprünglich 2003 von Dan North als Weiterentwicklung von TDD bekannt gemacht. 6

Dan North führte dabei syntaktische Konventionen für Unit-Tests ein. Als „Unit-Tests“ bezeichnet man Überprüfungen ob Komponenten wie gewünscht funktionieren. Er entwickelte „JBehave“ als Ersatz für „JUnit“, das alle verwandten Wörter von „Test“ mit dem Wort „Verhalten“ ersetzt hat. „JUnit“ ist ein Framework zum Testen von Java- Programmen welches von „JBehave“ durch veränderte Namenskonventionen abgelöst wurde.

Wieso führte Dan North eine Vokabular Umstellung durch? Edward Sapir und Benjamin Whorf bildeten eine Hypothese die aussagt, dass die Sprache, die wir nutzen, unser Denken beeinflusst 7 . Wollen wir unsere Denkweise verändern, hilft es demzufolge nach, die Sprache zu verändern.

Die Testgetriebene Entwicklung führte dazu, dass viele Entwickler den Entwicklungszyklus nicht optimal verwendet haben. Deswegen kam Dan North auf die Idee, durch Namenskonventionen das Verhalten in den Mittelpunkt zu rücken. Die Basis von BDD sind flexible Methoden, die darauf abzielen, Teams mit wenig Erfahrung in agiler Softwareentwicklung, den Einstieg zugänglicher und effizienter zu gestalten.

2.4 Arten von automatisierten Tests

2.4.1 Unit-Tests

2.4.2 Integrationtests

2.4.3 GUI-Tests

2.4.4 Integrationstests

Integrationstest sind Black Box System Tests. Jeder Integrationstest repräsentiert ein zu erwartendes Ergebnis von dem System. Integrationstest werden vom Testmanager erstellt, bezogen auf neue Features in diesem Release. Eine Feature kann mehrere Integrationstests haben. Ein Feature ist nicht als vollständig angesehen bis er seine Integrationstests bestanden hat.

2.4.5 Regressionstests

In der Entwicklung sollte nach jeder Änderung ein Regressionstest ausgeführt werden. Bei Regressionstest müssen Sie darauf achten, dass 1) Sie immer die gleichen Tests ausführen, wenn Sie einen bestimmen Code-Abschnitt testen und 2) das betreffende Tests die Akzeptanzkriterien der jeweiligen Anforderung abgleicht. Wenn nun später aber Jenkins in den Testprozess integriert würde, könnte man sich diese Arbeit sparen. Mit Jenkins würde es dann möglich sein, die Regressionstest regelmäßig auszuführen, etwas nach jeder Codeänderung oder einmal pro Nacht (Nightly Build). Falls Probleme auftrenten sollte, würde das Team per Email, HipChat oder ähnlichen Kommunikationswege informiert.

2.5 Automatisches Deployment von Software

3 Analyse

3.1 Ist-Analyse

Es gibt eine Vielzahl von verschiedenen Prozessmodellen die dazu beitragen, eine strukturierte und steuerbare Softwareentwicklung durchzuführen. Je nachdem welches Prozessmodell verwendet wird, sollte man die entsprechenden Testprozesse dem Vorgehensmodell zuordnen. Aus diesem Grund soll hier als erstes der bisherige Entwicklungsprozess der Shopware ERP-Schnittstelle Etos des vorherigen Agentur-Dienstleisters vorgestellt werden. Diese Schnittstelle greift auf die Ïmport- und Exportschnittstelle Internetshop"der Warenwirtschaft Apollon zu.

Diese Schnittstelle bittet für den Import in das ERP-System, dass ECSV-Format an. Zum Export Richtung Shopsystem wird ein CSV-Format offeriert. Beide Austauschformate nutzen als Zeichenkodierung das ASCII Format.

Auf Basis dieser Spezifikationen hat die vorherige Agentur eine Import- und Export Schnittstelle für das Shopsystem Shopware in der Version 5.0 entwickelt. Dies beinhaltete das Einspielen der vorhandenen Artikel inkl. Lagerbestände, Preise und Kategorien. Des Weiteren ermöglicht dies eine Übermittlung der Kunden und Bestellungen zum Warenwirtschaftssystem. Der Dienstleister ist hierbei nach einem klassischen Wasserfallmodel ohne Testautomatisierung vorgegangen.

Dabei ist die umsetzende Firma in 5 verschiedenen Phasen vorgegangen. In dem ersten Abschnitt, der Anforderungsanalyse und -spezifikation, wurden vom Projektleiter die Erwartungen und notwendigen Eigenschaften einer Schnittstelle vom Kunden aufgenommen, verarbeitet und in ein Pflichtenheft niedergeschrieben.

In der anschließenden Phase, Systemdesign und -spezifikation, wurde von den Softwareentwicklern die zu erstellende Softwarearchitektur konzipiert und niedergeschrieben. Dabei tauschten sich die Entwickler oft mit dem Projektleiter aus um zu überprüfen, dass alle Punkte aus dem Pflichtenheft berücksichtigt sind. Parallel dazu tauschte sich der Projektleiter mit dem Kunden aus um auf mögliche Änderungswünsche zu reagieren.

Im Anschluss dieses Abschnittest wurde die Programmierung von den Softwareentwicklern durchgeführt. Die umgesetzten Module wurden auf Basis des Quellcodes überprüft, sogenannte Reviews. Erste manuelle Tests der Schnittstelle fanden hier bereits statt. Als Resultat dieser Phase entstand die eigentliche Software für den Kunden.

Darauf folgend wurde die Software in einer Testumgebung eingespielt und in Betrieb genommen. Hierbei fanden dann die manuellen Integrations- und Systemtest statt. Dies

bedeutet das der Kunde in dem ERP-System verschiedene Artikel verändert hat, und diese Änderungen den Entwicklern mitteilte. Diese wiederum prüften, ob die Schnittstelle die gewünschten Veränderungen auch umgesetzt hat. Sobald bei dieser Testsituation ein Fehler aufgetreten ist, sind die Entwickler wieder in die vorherige Phase zurück gekehrt und haben diesen Fehler Analysiert, behoben und getestet. Anschließend haben die Entwickler die angepasste Version wieder in die Testumgebung eingespielt und die Phase erneut angestoßen.

Erst nach Vollendung der Integrations- und Systemtests, hat die Schnittstelle eine Freigabe vom Kunden für das Live-System erhalten. Das Einspielen der neuen Softwareversion ins Produktiv-System übernahmen die Softwareentwickler. Darauf hin führte der Kunde und die Softwareentwickler weitere System- und Integrationstests durch.

Diese Prozesse werden bei jeglichen Veränderungen des Warenwirtschafts- oder Shopsystems erneut angestoßen.

3.2 Schwachstellen-Analyse

3.2.1 Allgemein

Wir haben einen kurzen Entwicklungszyklus dem ein Planungszyklus voran geht. Nach der Entwicklung folgt eine zwei stufige Testphase. Dies sollte eigentlich ausreichend sein um eine annähernd fehlerfreie Software entwickeln zu können, oder? Bevor man diese Frage beantwortet, zeigen ein paar typische Szenen, was während der manuellen Testphasen im vorherigen Entwicklungsprozess passiert ist.

3.2.2 Schlechte Ressourcennutzung

Die meisten Testfälle von der ERP-Schnittstelle wie im Kapital Y!!§!§!§!§!§ geschildert, benötigen mindestens 1 Softwareentwickler und einen Sachbearbeiter auf Seiten des Kunden. Des Weiteren ist manuelles Testen eine sehr aufwendige, monotone und fehlerbehaftete Tätigkeit. Zum Beispiel: Um Features von der ERP-Schnittstelle zu testen, muss jemand in der Warenwirtschaft Änderungen vornehmen, diese übermitteln und dann anschließend prüfen ob diese auch korrekt übermittelt sind. Beim testen des Kunden- und Bestellexportes muss der Softwareentwickler einen Kunden anlegen, Artikel in den Warenkorb legen, Rechnungs- und Lieferadresse auswählen und schlussendlich die Bestellung mit einer Zahlungsart abschließen. Anschließend müssen die Importierten Daten in der Warenwirtschaft

3.2.3 Ineffizient

Bevor der Integrationstest starten kann, muss die Testumgebung vom Online-Shop und Warenwirtschaftssystem aktualisiert und mit dem jeweils letzten Stand der Software bestückt werden. Dies erfordert auf Dienstleister, wie auch Kundenseite einen größeren Vorbereitungsaufwand um mit dem eigentlichen testen anfangen zu können. Des Weiteren wird die Vorbereitung durch das Tagesgeschäft des Dienstleisters und Kunden beeinträchtigt. Z.B. kommt ein Notfall-Support für den Dienstleister rein weil der Kunde keine Bestellungen mehr ins ERP-System einspielen kann, oder beim Kunden funktionieren andere Elementare Anwendungen nicht, die vom Ansprechpartner des Kunden gelöst werden müssen. Man spricht in diesem Zusammenhang auch vom sogenannten „Sägeblatt-Effekt"

3.2.4 Sehr fehleranfällig

Durch die große Anzahl von einzelnen manuellen Testschritten ist der Prozess sehr anfällig. Es genügt das nur ein einzelner Abschnitt falsch oder nicht vollständig ausgeführt wird, dass der ganze Testprozess fehl schlägt und somit von vorne starten muss. Des Weiteren erfordertet diese Art von Testprozess eine exakte und perfekte Zusammenarbeit zwischen dem Dienstleister und dem Kunden. Es ist ausgeschlossen, dass die beteiligten Personen über einen so langen Zeitraum fehlerfrei arbeiten können.

3.2.5 Gesunkenes Vertrauen

Wegen mangelhafter Testabdeckung und fehlenden Integrationstests tauchten immer wieder nach der Veröffentlichung einer neuen Schnittstellenversion unangenehme Überraschungen auf. Z.B. fanden beim Import in das Shop-System nur noch Artikel ohne Varianten Berücksichtigung. Nach dem darauffolgenden Release und Fix des beschriebenen Problems, ließen sich zwar Artikel mit Varianten importieren, aber Artikel ohne Varianten fehlten gänzlich.

Die Anhaltenden Probleme das z.B. Funktion A defekt ist und die Reparatur dadurch Funktion B beeinträchtigt hat das Vertrauen des Kunden in die ERP-Schnittstelle nachhaltig beeinflusst und jedes Release für den Kunden zu einer Zerreißprobe werden lassen.

3.2.6 Review

Bei Reviews sollten Kollegen konkrete Feedbacks geben und codierte Stellen aufdecken, die gegen diese Richtlinien verstoßen, so dass keine unnötigen Fehler in den Sourcecode

einfließen. Aber der vorherige Dienstleister hatte mit starker Entwickler Fluktuation zu kämpfen.

3.2.7 Keine Regressionstests

3.2.8 Schlussfolgerung

Die Fehlerquellen wie menschliches Versagen, Schwankungen und mangelnde Konsistenz lassen keinen Zweifel daran aufkommen, dass manuelle Prozesse nur eine geringe Chance haben, die schnellen und reproduzierbaren Ergebnisse zu liefern. Ganz zu schweigen davon, dass bei einer großen Code-Basis das manuelle Testen in aller Regel den für diese Iteration bemessenen Zeitrahmen bei weitem sprengt.

Außerdem werden Integrationstests sehr viel seltener ausgeführt als eigentlich empfehlenswert. Somit steigt das Risiko, dass Fehler erst ziemlich auftauchen, enorm.

3.3 SWOT-Analyse

3.3.1 Durchführung

Anhand der vorliegenden Ist-Analyse wurde für eine detailliertere Entscheidungsgrundlage eine SWOT-Analyse durchgeführt. In dieser Analyse wird die Thematik der Automatisierung von Tests beleuchtet um eine Entscheidungsgrundlage für das zu erstellende Soll-Konzept zu schaffen.

Zuerst wurden die Stärken und Schwächen der Testautmatisierung analysiert und festgelegt. Anschließend wurden die Chancen und Risiken des Prozesses erhoben.

3.3.2 Stärken

- Verlässlichkeit:

  Einmal erstellte Tests werden bei jeder Änderung wieder durchgeführt und Garantieren das die Software sich so verhält wie es der Test es überprüft. Ist dies nicht der Fall wird der betroffene Entwickler darüber informiert. Dadurch, dass die Tests immer ausgeführt werden, erhält der Entwickler deutlich früher eine Rückmeldung ob seine Veränderung des Quellcodes ggf. Seiteneffekte aufweist.

- Vollständigkeit (Testabdeckung):

  Wenn der Quellcode zu 100 % mit automatischen Tests abgedeckt ist, ermöglicht dies eine Neustrukturierung, sogenanntes Refaktoring, ohne das Funktionen vom Entwickler unbemerkt aus der Software verschwinden oder nicht wie gewohnt weiter funktionieren. Durch eine Vollständige Testabdeckung wird das Vertrauen in die Software beim Endanwender deutlich gesteigert da eine einmal bekannte Funktionalität des Programms sich nicht unbewusst verändert.

- Wiederholbarkeit

  Durch das automatisiertes Testing können nach jeder Änderung die vorher definierten Tests angestoßen werden. Durch diesen Automatismus sind keine Personal Ressourcen zur Überprüfung der Software notwendig. Zusätzlich sinkt die Fehleranfälligkeit der Tests enorm, da die erstellten Tests - bei jeder Ausführung immer wieder das genau gleiche Testmuster anwenden. Dies ist bei manuellen Tests nicht immer gegeben.

- Reproduzierbarkeit

  Einmal definierte Testmuster lassen sich immer wieder ausführen (siehe Wiederholbarkeit). Diese Eigenschaft ermöglicht es, dem Programmierer aufgetretene Fehler, zu reproduzieren. Hier ist kein investigativer Rechercheaufwand des Entwicklers beim Tester notwendig, da er durch den definierten Test, den exakten Systemkontext kennt.

- Reporting

  Eine zusätzliche Stärke der automatisierten Tests ist die Reportingmöglichkeit. Bevor Änderungen in den Hauptzweig der Versionsverwaltung dürfen, können diese durch automatisierten Tests auf Funktion und Qualität überprüft werden. Bei einem negativen Testergebnis ist die Übernahme der Anpassungen nicht möglich un der betroffene Entwickler erhält ein detaillierte Informationen zu den jeweils fehlgeschlagenen Tests.

- Auswertung

  Durch die ständige Ausführung der Tests können auch Statistiken erstellt werden. Wie hoch ist die Code-Coverage (Abdeckung des Quellcodes durch automatisierte Tests) und welche Qualitätsmetriken haben sich über die Zeit wie verändert. Dies sind für das Projektmanagement objektive Kennzahlen, die dem Kunden interessieren und eine Auskunft über den Status und Qualität der zu entwickelnden Software gibt.

3.3.3 Schwächen

- Hohe Einstiegshürde

  Aller Anfang ist schwer. Der initiale Aufwand um Automatisierte Tests, sogenannte Unit-Tests zu schreiben, ist sehr hoch HIER MICH SELBST ZITIEREN :D. Die Aufwände für die Einarbeitung amortisieren sich erst Mittel- und Langfristig.

- Hoher Planungsaufwand

  Um bei einem initialen neuem Software-Projekt automatisierte Tests zu verwenden ist ein hoher Planungsaufwand notwendig. Wo sollen die Tests ausgeführt werden? Benötigen wir eine Systemlandschaft für die automatisierten Tests? Können wir in diesem Projekt überhaupt alles Testen?

- Aufdecken unerwarteter Fehler

  Manuelle Tests decken deutlich mehr Fehler als das Automated Testing auf, weil Tester immer auch intuitiv agieren und von den geplanten Wegen durch die Anwendung abweichen können. Dazu kommt: Je erfahrener der Tester, desto besser ist meist auch seine persönliche „Testheuristik“ und damit die Erfolgsquote beim Auffinden von Fehlern.

  Durch die festgelegten Testszenarien ist kein destruktives Testen, wie es Menschliche Tester können, möglich. Dies bedeutet das Fehler die nicht durch Tests abgedeckt sind, beim Entwickler auch nicht erscheinen. Zum Beisiel wäre ein Warenkorbprozess innerhalb eines Shopsystem der zu 100 % mit Tests abgedeckt und somit von den Metriken her ideal umgesetzt ist aber Wertlos wenn die Tests immer nur mit 19 % Mehrwertsteuer durchgeführt werden. Sobald in dem Shop Bücher mit 7 % zum verkauf stehen, besteht hier ein großes Fehlerpotenzial, dass die Artikel ggf. mit den falschen Mehrwertsteuern berechnet oder Mischwarenkörbe (Artikel mit 7 % und 19 % MwSt im Warenkorb) vollständig falsch kalkuliert sind.

3.3.4 Chancen

- gesteigertes Vertrauen in die Software bei den Kunden

  Durch die Zuverlässigkeit der einzelnen Funktionen innerhalb der Software wird das Vertrauen vom Kunden in ihr gestärkt. Eventuelle Ängste, dass nach jedem Update alles anders funktioniert als vorher, können damit entkräftet werden.

- Bessere Ressourcennutzung Die automatisierten Tests ermöglichen dem Entwicklungsteam die freigewordenen Ressourcen, die vorher bei den manuellen Tests gebunden waren, für die Pflege und Weiterentwicklung der Software zu verwenden. Dies ermöglicht dem Kunden, dass vorhandenen Budget effektiver einzusetzen.

3.3.5 Gefahren

- Skepsis der Kunden auf Wirtschaftlichkeit Durch den am Anfang spürbaren Mehraufwand für Automatisierte Tests, besteht die Möglichkeit das der Kunde an der Wirtschaftlichkeit dieses Prozesses zweifelt. Der Messbare Erfolg ist erst Mittel- und Langfristig beweisbar.
- Integration Testprozess Eine halbherzige Integration des Automatisierungs-Prozesses von den Software-Entwicklern stellt eine Gefahr da. Ohne Korrektur dessen, kann dem Kunden dann auch Mittel- und Langfristig kein Vorteil der Testautomatisierung bescheinigt werden und er verliert das Vertrauen in die Software und dem beauftragtem Unternehmen.

3.4 Soll-Konzept

3.4.1 Automatisierter Testprozess

Durch die Schwachstellen-Analyse können wir feststellen, dass es vom entscheidender Wichtigkeit ist, einen automatisierten Testprozess in dem Entwicklungsprozess von der Etos Schnittstelle einzubinden. Die dadurch eingebrachten Vorteile sind erheblich. Fast alle zuvor erwähnten Probleme können durch Automatisierung einfach gelöst werden. Automatisiertes Testen bringt mehr Flexibilität und Reproduzierbarkeit, ermöglicht Regressionstest in jeder Iteration und erleichtert Testern maßgeblich die Arbeit. Somit ist eine Steigerung der Effizienz und der Qualität in der Softwareentwicklung gewährleistet.

Basierend auf der im vorherigen Kapitel erstellten Schwachstellen- und SWOT-Analyse ist eine Umsetzungsstrategie entstanden, die nachhaltig die Probleme des Kunden löst.

3.4.2 Umsetzungsstrategien

Die zu erstellenden Testroutinen sollen anhand einer sogenannten Testpyramide (ABBILDUNG HIER EINFÜGEN VERLINKEN) umgesetzt werden. Dabei sollen die Testarten,

die im Kapitel 2.4 beschrieben sind, Anwendung finden. Zielsetzung ist die Verhinderung von automatisierten Test die nach Monaten oder sogar Jahren nach der eigentlichen Programmierung erst Berücksichtigung finden.

Das Fundament der Pyramide bilden die Unit-Tests. Sie sind die Grundlage jeder soliden Vorgehensweise zur Testautmatisierung. Die Ausführung der jeweiligen Tests benötigen in der Regel nur ein paar Millisekunden und verwenden daher extrem wenig Computer-Ressourcen. Durch diese Art von Tests wird gewährleistet, dass einzelne Module Ihre getestete Funktionalität immer beibehält, ansonsten schlägt der Test fehl.

Dies alleine garantiert noch keine Fehlerfreie Software. Die Komponenten können einzelnen einwandfrei funktionieren aber durch die Kompositionsart der jeweiligen Module nicht die gewünschten Eigenschaften aufweisen. Z.B. können die einzelnen Komponenten einwandfrei funktionieren (Schiebeschloss und Schiebetür). Wenn man diese aber falsch miteinander verbindet, ist es trotz verriegeltem Schloss möglich die Tür zu öffnen. Um so ein Verhalten in der Softwareentwicklung zu verhindern sind die Integrationstests verantwortlich.

Die Integration-Tests-Ebene der Testautomatisierungspyramide ist dafür verantwortlich, dass das Systemverhalten zu prüfen ist, unabhängig von der Benutzeroberfläche. Wenn Testfälle auf dieser Ebene erledigt werden können, sollte dies nicht in der Oberfläche der Anwendung durchgeführt werden, weil GUI-Test aufwändig zu schreiben, aufwendig durchzuführen und labil sind.

- Einführung Automatisierter Tests - Testpyramide - Einführung von Qualitätsüberprüfungstools - bessere Ressourcen Nutzung

4 Umsetzung von Testautomatisierung

4.1 Voraussetzungen

4.2 Prozessänderungen

4.3 Testwerkzeuge

4.3.1 PHP-Unit

4.3.2 PHPSpec

4.3.3 Behat/Mink

4.4 Durchführung

4.5 Controlling

5 Fazit

Wünsche Euch allen viel Erfolg für das 7. Semester und bei der Erstellung der Thesis. Über Anregungen und Verbesserung an dieser Vorlage würde ich mich sehr freuen.

Literaturverzeichnis

[1] Balzert, Helmut; Bendisch, Roman; Kern, Uwe; Schäfer, Christian; Schröder, Marion; Zeppenfeld, Klaus: Wissenschaftliches Arbeiten: Wissenschaft, Quellen, Artefakte, Organisation, Präsentation, W3L-Verl., Herdecke [u.a.] 2008, ISBN: 978-3937137599.

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde/Prüfungsstelle vorgelegen hat. Ich erkläre mich damit einverstanden/nicht einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hoch geladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

_____ _____

(Ort, Datum) (Eigenhändige Unterschrift)