Erweiterung eines Software-Testansatzes durch soziale Aspekte

Bachelorarbeit

vorgelegt am: 28. Juni 2017

Name:	Kevin Glöckl
Matrikelnummer:	67601
Fachbereich:	Lehrstuhl für Software Engineering
Studienbereich:	Internet Computing
Studiengang:	Informatik
Studienjahrgang:	2017
Erstgutachter:	Prof. Dr. Sven Apel
Betreuer:	Sergiy Kolesnikov

Inhaltsverzeichnis

1 Einführung

Als Entwicklerteam steht man, besonders am Anfang einer Softwarelösung, vor schwierigen Entscheidungen, wie ein Softwareprodukt entwickelt werden soll. Seien es Überlegungen, wie die Architektur auszusehen hat oder wie die darunterliegende Datenbank gefüllt werden soll. Dann müssten auch noch Tests geplant werden, die zur Sicherung der Stabilität und Performanz durchgeführt werden. Ein umfangreiches Unterfangen, welches für ein reines Entwicklerteam meist zu viel ist und deswegen in mehrere Arbeitsbereiche aufgeteilt werden. Dabei werden die weniger wichtigen Bereiche wie das Testen nicht genug beachtet und nach großen Entwicklungsschritten durchgeführt. Aber das Durchführen von Tests während oder nach kleineren Entwicklungen ist wichtig für die Qualität des gelieferten Produktes und kann als direktes Feedback für das Entwicklerteam betrachtet werden. Aufgrund der meist längeren Entwicklungenzeiten und den Zeitstress Software pünktlich zu liefern, rücken die Tests an den Rand eines Entwicklungsprozesses und dadurch fallen diese wiederum spärlicher aus als geplant. Fehler im System können dadurch übersehen werden oder fallen erst in Produktivbetrieb auf.
Ein weiterer Punkt ist auch die Vielschichtigkeit der Tests, die für viele verschiedene Bereiche eingesetzt werden. Seien dies Performanztests zur Sicherung der Robustheit des Systems, oder ein Test zur generellen Überprüfung aller Funktionen. Dabei sehen sich die Entwickler angesichts dieser Vielzahl an Tests überfordert und können die Zeit dafür nicht aufwenden diese Fülle an Tests zu implementieren, was ein eigenes Testteam rechtfertigen würde.
Heutiger Entwicklungsstandard der Industire beinhaltet, dass ein Entwicklerteam ihre neu entwickelte Funktionalität testet, um Qualität sicherzustellen, aber ein genereller Test über die Funktionalität im gesamten System geschieht erst am Ende des Entwicklungsprozesses, was zu diversen Problemen führt. [1] Aufgrund des späten Testens der einzelnen Komponenten im gesamtem System werden Fehler erst am Ende der Entwicklung aufgeworfen, welche teuer werden diese im Nachhinein zu beheben. Zum Anderen liegt die Komplexität des Testens nicht in der Verantwortung einer einzelnen Abteilung.
Dieses späte Testen von Software wird in dieser Arbeit genauer betrachtet. Dazu

wird das „späte“ Testen, was im Wasserfall häufig der Fall ist, beschrieben und mit dem agilen Testansatz, als die neuere Art des Testens, verglichen. Es wird erklärt, wo die Vorteile des aufbauenenden (=iterativen) Ansatzes liegen. [1] Im Vergleich zum späten Testen, löst der agile Ansatz die Probleme durch kleinere Prototypentwicklungen, die als Basis jeglicher Weiterentwicklungen benutzt werden und ermöglicht eine umfangreichere Sicht auf die Schwachpunkte eines Systems. Dieser, für diesen Testansatz typische iterative Qualitätscheck, beugt dem plötzlichen Aufkommen von gröberen Implementierungsfehlern vor, da der neu entwickelte Code zeitnah auf unvorhersehende Fehler getestet wird. Desweiteren können die Entwickler gemeinsam mit den Testteam die Entwicklungen durchgehen und Testfälle produzieren. Dabei wird durch die iterativen Entwicklungszyklen dem Kunden ein Ergebnis präsentiert und mögliche Fehlentwicklungen vermieden. Aufgrund dieser Konversation der Entwickler, Tester und Nutzer wird die Last der Verantwortung, die mit der Qualitätssicherung einhergeht, aufgeteilt. Das iterative Testen baut darauf, dass sich ein Tester mit dem Entwicklerteam und den Kunden auseinandersetzt und versucht basierend auf die Ideen aller Parteien, ein Grundgerüst für einen Testfall zu bauen. Aber auch die agile Testentwicklung hat einen Schwachpunkt, da dieser Ansatz immer dem gleichen Schema folgt. Dieses Schema gibt die ungefähre Richtung an, wie das agile Testen im Gesamten auszusehen hat und woran sich ein Testteam halten soll, um effektiv zu arbeiten. Dadurch erscheint das iterative Testen zwar dem System angemessen und agil, benötigt aber aufgrund des Schemas mehr Zeit einen Testfall zu entwickeln, der genau auf die Schwachpunkte des Systems hinweist. Dazu stellt James Bach eine schnellere Lösung vor, die den Testfall noch besser abbildet und auf die Bedürfnisse des zu testenden Objektes einhergeht. [2] Seiner Ansicht nach sorgen die Antworten auf die folgende Fragen für den nötigen Testfall:

- Was sind die Schwachstellen?
- Welche Eingaben können getätigt werden, die die Schwachstelle aufdeckt und zu einem Fehler führt?
- Wie schlimm wären die Konsequenzen und Wer oder Was würde von dem Fehler betroffen sein?

In dieser Arbeit vorgestellten Fragen gehören bereits zum ersten Teil des vorgestellten Testansatzes, der das Ziel verfolgt, zu erklären, was der beste Weg ist, zu einem neuen Testfall zu gelangen. Diese Fragen helfen dem Tester, auf die Kommunikation mit dem Entwickler zu achten, um dadurch schneller die wesentlichen Punkte zu verstehen. Durch das Verstehen der implementierten Funktion wird es einfacher diese schnellst möglich zu testen. Dadurch werden von vornherein, im Gespräch mit den Entwicklern, mögliche Fehler ausgekundschaftet und durch Tests aufgezeigt. Dieser

theoretische Ansatz stützt sich auf den iterativen Weg, der in Kapitel Zwei erklärt wird, und mit dem oben genannten Fragegerüst von James Bach verbessert wird.
Der zweite Teil ist die praktische Anwendung des neuen Testansatzes an einem produktiv genutzten Content Management System, kurz CMS, welches in der Industrie aktiv benutzt wird und für die Evaluation herangezogen wurde. In der Vergangenheit wurde das System intern benutzt und sollte bald auch externen Nutzer zur Verfügung stehen. Dementsprechend muss die Leistung getestet werden - dies geschieht unter anderem durch Performanztests. Die Performanztests werden am Ende des iterativen Schemas im Kapitel Zwei erklärt und bieten die Möglichkeit, einzelne Bereiche des Systems zu testen. Aufgrund der oben erwähnten Fragen ergab sich die Möglichkeit, Überlegungen hinsichtlich des wie das System getestet werden muss, zu überarbeiten und den Performanztest vom agilen Ansatz so zu erweitern, dass ein Problem des Systems nachgewiesen werden konnte.
Die Erweiterung des Performanztestes liegt darin menschliche Nutzer zu simulieren und deren Suchverhalten nachzuahmen. Als Grundlage für die Simulation wird ein Algorithmus auf Basis der Statistik der Google Search Engine benutzt. Deswegen konnte der simulierte Nutzer das System so realitätsnah wie möglich testen und es konnte somit Engpässe oder Fehler in der Ausführung ermittelt werden.

Die Ziele dieser Arbeit werden in folgenden Punkten aufgezählt:

- Zu Einem werden in dieser Arbeit die gängigsten und viel benutzten Testarten aufgezählt und miteinander verglichen. Dabei gezielt auf die Schwachstellen jeder Art eingegangen und ein neuer Ansatz beschrieben, der sich für Performanztests besser eignet.
- Zum Anderen wird ein neuer theoretischer Testansatz praktiziert und, der daraus resultierende Performanztest, der durch den Einsatz der Google Search Statistik einen Nutzer simuliert, benutzt. Dabei werden diese erweiterten Performanztests an einem industriellen benutzten System einer Firma ausgeführt.

2 Übersicht klassischer Test-Ansätze und deren Nachteile

Einer der klassischen Schematas wie Software entwickelt wird ist das Wasserfall-Modell, welches zu den neueren Modellen der Software Entwicklung herangezogen wird.[3] Dazu wird in diesem Kapitel zuerst das Wasserfall-Modell erläutert, wobei speziell das, für Wasserfall typische, Testen des Systems betrachtet wird und später mit dem agilen Test Ansatz verglichen.

2.1 Spätes Testen - Wasserfall-Modell

Software Entwicklung im Wasserfall-Modell geschieht strikt nach einem festgesetzten Muster wie in 2.1.1 betrachtet werden kann. In der Anforderungsanalyse werden in Kooperation mit dem Kunden die Ziele und Funktionalitäten des Systems entwickelt. Die daraus abgeleiteten Anforderungen an das System dienen als Systemspezifikation, in Form eines Pflichten-Lastenheftes. Die so festgelegte Systemspezifikation ist die Grundlage für den zweiten Kasten, die Festlegung der Systemarchitektur und des Systemdesigns. In der Codierung und Modultestphase findet die eigentliche Softwareentwicklung statt. Das aus der vorherigen Phase stammende Designdokument

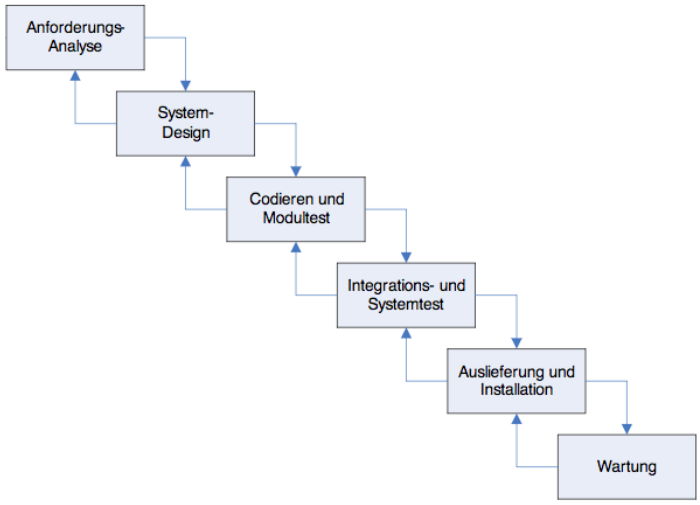

Abbildung 2.1.1: Wasserfall-Modell

dient als Vorgabe für die Implementierung, dabei werden die implementierten Module von den Entwicklern selbst getestet. Nach der Implementierungsphase folgt die Systemintegration, in der die individuellen Module zusammengefügt, integriert und als ganzes System getestet werden. Dieser als Systemtest bezeichnete Schritt bildet den Abschluss der eigentlichen Entwicklung. Das System wird nun dem Kunden übergeben und befindet sich im Einsatz. Dies fällt in die Phase der Benutzung und Wartung, welches unter normalen Umständen die längste Phase innerhalb des Softwareentwicklungsprozesses ist. Die Wartung des Systems beinhaltet dabei das Korrigieren der Fehler, die bei der Benutzung des Systems gefundenen wurden. Dazu werden neue Entwicklungen gestartet, falls sich neue Anforderungen ergeben.[3]
In Analyse und Design werden die Ziele und Funktionalitäten des Systems mit dem Kunden besprochen und niedergeschrieben. Das fertige Dokument beinhaltet sämtliche Informationen, die mit dem System in Verbindung gebracht werden können, so auch Beschreibungen zum Durchführen der Tests. Diese Beschreibungen existieren im Dokument, aus denen werden vor der eigentlichen Implementierung Tests entwickelt, die nach Abschluss der gesamten Entwicklung durchgeführt werden.[3] Für diese ausgeführten Tests soll in der Anforderungsanalyse bereits genügend Zeit eingeplant werden, da es zu Verzögerungen der Implementierungsphase kommen könnte, die die Testphase verkürzen könnte.[1] In der Testphase zeigen sich Fehler in der Umsetzung, woraufhin üblicherweise Bugfixing[1] betrieben wird, wodurch es häufig zu code- and fix-cycle kommt.[1][4] Dabei benötigt das Ausbessern der gefundenen Fehler weitere Ressourcen. Diese Ressourcen können unter anderem weitere Entwickler sein, die sich in den Code einlesen müssen, um ihn verstehen zu können. Diese extra Ressourcen, sind üblicherweise nicht einkalkuliert, was unter Umständen teuer werden kann. Dazu kommen häufig Lizenzen der Software, die von den Entwicklern benutzt werden oder weitere Anschaffungen im Hardwarebereich. Schließlich kosten diese Ressourcen Geld, wodurch die späte Überarbeitung des Fehlers entsprechend teurer ist, als es eine frühzeitige Überarbeitung gewesen ist.[4]
Wie in der Einleitung schon beschrieben, gibt es neben dem späten Testen und der daraus folgenden teuren Überarbeitungen noch einen weiteren Schwachpunkt, welcher mit dem Testen am Ende des Entwicklungszyklus einhergeht. Wer sollte sich Gedanken übers Testen machen? Klare Antwort: der Tester. Hierbei liegt der Irrtum, was im Folgenden erklärt wird.
Auf Basis des Anforderungsdokumentes überlegten sich die Tester während der Entwicklungsphase allein ohne die wesentliche Hilfe des Entwicklerteams Testfälle, anhand derer die Software für gut befunden werden sollte.[5] Der wesentliche Nachteil ist

[1] Finden eines Fehler im Code und das unmittelbare Bereinigen des Fehlers, zumeist aber nicht die „sauberste“ Lösung und womöglich später Ursache weiterer Fehler im laufenden System.

die Sicht der Tester auf das System. Ein Tester hat nur eine begrenzte Sicht auf eine Software, nämlich die Gesamttests im Anforderungsdokument, wie das System funktionieren soll. Es fehlt die Sicht des Entwicklerteams, ob die einzelnen Code-Module während der Implementation überhaupt getestet wurden und falls ja, was durch die Tests abgecheckt wurde.[1] Schließlich sollte das Testen die Aufgabe erfüllen, den Code einem Qualitätscheck zu unterziehen.[1] Es wird von Test-infiziert gesprochen, wenn das gesamte Entwicklerteam und Testteam für die Qualität des gelieferten Produktes sorgt.[1] Aus der Sicht der Entwickler wird durch das Testteam ermöglicht neue Arten des Testens zu finden, um mögliche Fehler zu finden und durch weitere Entwicklungen ausbessern zu lassen. Diese zusätzliche Sicht fehlt aber im Wasserfall-Modell, da die Testabteilung vom Entwicklungsteam räumlich getrennt ist und das wiederum den wichtigen Austausch von wertvollen Informationen unterbindet.[5]

Zusammenfassend können aus der Sicht eines Testers für die traditionelle Softwareentwicklung zwei Nachteile aufgezählt werden. Zum Einen kann das späte Testen und Finden eines Fehlers als teuer betrachtet werden. Zum Anderen wird durch das Phasendenken eine Isolation der Abteilungen hervorgerufen, die es schwierig macht, Qualität anzubieten. Diese Nachteile werden durch einen iterativen Weg, wie es in agilen Testansätzen praktiziert wird, behoben, was im Folgenden erklärt wird.

2.2 Iterativer - Agiler Testansatz

Die agile Softwareentwicklung wird in dieser Arbeit stärker fokusiert, da diese die Schwächen des späten Testens behebt.
Agile Methoden der Softwareentwicklung (lightweight methods) entstanden Mitte der 90er Jahre als Reaktion auf die bürokratischen, dokumentlastigen klassischen Methoden und deren Schwierigkeiten im Umgang mit sich schnell verändernden Anforderungen. Ihr Fokus liegt auf der eigentlichen Softwareentwicklung und nicht auf Prozess, Design oder Dokumentation. Um das Ziel der schnellen Anpassung an Änderungen zu erreichen, pflegen agile Methoden sehr kurze Iterationen, bei denen inkrementell entwickelt wird. Durch kurze Iterationen und kurze Release-Zyklen wird schnelles Feedback beim Kunden über den Fortschritt und Funktionsumfang eingeholt. Dieses schnelle Feedback dient zur Steuerung der weiteren Entwicklung. Im „Agile Manifesto“ drücken die führenden agilen Verfechter ihre Wertevorstellung der agilen Entwicklung aus.[3] Das Agile Manifesto stellt Individuen und deren Interaktion über Prozesse und Pläne dar. Die Interaktion der Individuen soll zu besseren Ergebnissen führen, als die Verfolgung eines Plans, der die gewünschte Realität schlecht wider-

spiegelt und träge auf Veränderungen reagiert. Ebenfalls wichtig ist auch die aktive Einbindung des Kunden in die Entwicklung, d.h. durch ständige Kommunikation und Feedback des Kunden werden seine Vorstellungen besser realisiert als durch strikte Implementierung des gedruckten Anforderungskatalogs. Ziel einer agilen Entwicklung ist es möglichst schnell dem Kunden eine funktionierende und ihm wertliefernde Software zu übergeben, auch wenn dazu eine geringere oder gar andere Funktionalität implementiert wird, als ursprünglich vereinbart. Dies bedeutet beispielsweise, dass vom Kunden kommunizierte Änderungswünsche umgesetzt werden, obwohl sie nicht im ursprünglichen Vertrag enthalten waren.[3]

2.3 Die 4 Quadranten - Grundpfeiler des agilen Ansatzes

Um ein tieferes Verständnis der agilen Entwicklung zu gewinnen werden im nächsten Unterkapitel die 4 Quadranten (siehe Bild 2.3.1) des agilen Ansatzes beschrieben, die als Leitfaden durch die agile Testentwicklung gesehen werden. Wie im iterativen Testansatz schon grob erklärt wurde, basiert das agile Testen auf dem Konzept des „Agile Manifesto“.[3] Dieses Konzept beschreibt die genauen Schritte, die das Agile Testen einhalten muss um effektives Arbeiten gewährleisten zu können. Individuen und deren Interaktionen stehen über Prozesse und Pläne. Die funktionierende Software steht über umfangreiche Dokumentation. Zusammenarbeit mit Kunden steht über Vertragsverhandlungen und das Feedback über den momentanen Status der Software steht über das Folgen des Planes.[1] - Auszug des Agile Manifesto, welches der Urstein des agilen Testan-

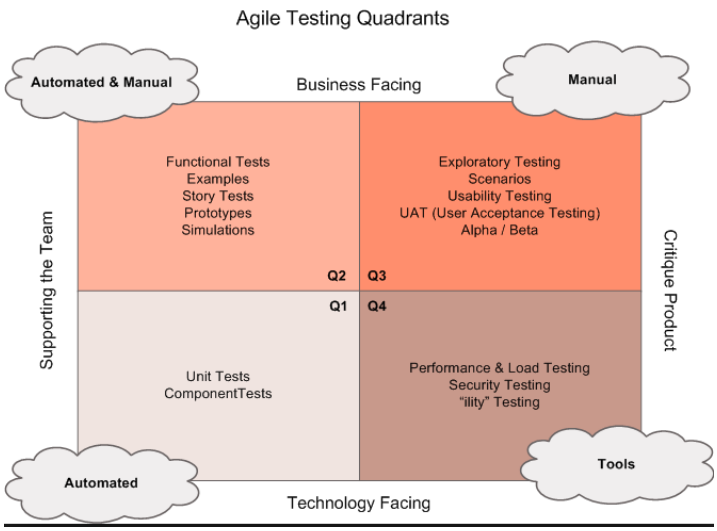

Abbildung 2.3.1: Testquadranten des agilen Testens

satzes ist. An den Agile Manifesto ausgerichtet hat Brian Marick die 4 Quadranten definiert, die im folgenden erklärt werden.[6] In diesen Quadranten beschreibt er welche Arten von Tests für verschiedene Ansprechpartner und Geschäftsleute benutzt werden, um das agile Manifesto anzuwenden. So gesehen sind die vier Quadranten der Leitfaden durch das Konzept des agilen Testansatzes und gehören zum Grundinventar, deswegen werden sie auch aufgezählt und erklärt.
Die vier Quadranten:

- **1 Quadrant** Dieser Quadrant wird als Kern der agilen Software Testentwicklung angesehen, dabei werden die einzelnen Codeabschnitte (Methoden) oder Komponente, was mehrere Methoden unter sich einigt, auf Fehler kontrolliert.[1] Bevor getestet werden kann, sollte im vornherein über das Testergebnis nachgedacht werden, welches Verhalten der Test testen soll, hierbei fällt das Schlagwort TDD[2]. Wie im Bild 2.3.1 im ersten Quadranten ersichtbar ist, wird vorallem mit JUnit Tests gearbeitet, welche automatisiert wiederholt werden und einen Überblick über die interne Codequalität bereitstellt.[1] Der Entwickler sollte vor der eigentlichen Programmierarbeit gründlich darüber im Klaren sein, welche Schnittstellen das zu implementierende System später haben soll. Er geht dadurch von der Anwendersicht an die später entstehende API heran, was beim Design von Vorteil ist. Darüber hinaus hat er durch die implementierten Testfunktionen Kontrolle darüber, ob der von ihm implementierte Code auch korrekt arbeitet.[7] Wie diese JUnit-Tests auszusehen haben wird nicht behandelt, da dies für den weiteren Verlauf dieser Arbeit nicht benötigt wird. Es sei nur erwähnt, dass es dem Entwickler selbst überlassen ist, welche Testzenarien erstellt werden.[7]
- **2 Quadrant** Quadrant 2 wie im Bild 2.3.1 ist im Vergleich zum ersten Quadranten höher liegend, was beschreibt, dass nicht das direkte Verhalten des Codes mit JUnit getestet, sondern das Systemverhalten funktional oder über GUI[3] bewertet wird.[1] Ob funktionale Tests oder Oberflächen Tests, beide basieren auf auf Anforderungen, die Nutzerseitig erstellt werden. Die Anforderungen werden in Form von User-stories (zu deutsch Nutzergeschichte) zur Verfügung gestellt. Die Nutzergeschichte legt im Wesentlichen fest wer, was, aus welchen Grund getan wird und ist für gewöhnlich nicht länger als 1-2

[2](Test Driven Development) Bei Test Driven Developement schreibt der Entwickler hier zunächst die Programme, die zum Testen der später zu implementierenden Funktionen oder Klassen benötigt werden. Wenn die Tests, die man auch als Entwickler-Tests bezeichnet, erst implementiert sind, kann der Entwickler sich an die Implementierung der eigentlichen Funktionen und Klassen machen.[7]
[3]Graphical User Interface die Benutzeroberfläche eines Programmes.

Sätze.[8] Die Wertigkeit von Nutzergeschichten liegt darin, dass das Erzählen einer Geschichte die Bedeutung dieser erklärt ohne es exakt zu definieren.[8] Somit bleibt es dem Entwickler und Tester über die Nutzergeschichte zu benutzen um funktionale oder grafische Tests zu erstellen.
Die Anforderungen (Nutzergeschichten) als Grundstein kann zwischen funktionale Tests oder grafische Tests unterschieden werden. Währenddessen funktionale Tests wie JUnit Tests sind, werden grafische „Tests“ als Kommunikationsgrundlage zwischen Entwickler und Nutzer eingesetzt.
Funktionale Tests überprüfen die Funktionalität (der Login Button einer Seite zum Beispiel). Dafür werden die Tests auf der Anwendungsschicht (erste Schicht im OSI Model) durchgeführt, um die Funktionen zu testen.[1]
Grafische Tests unterscheiden sich zu den funktionalen Tests insofern, da diese von Designern benutzt werden, um eine Abstimmung zwischen Entwickler und Nutzer zu erreichen.[1] Mit Mockups[4] und WireFrames [5] werden die Ideen der Nutzergeschichte visuell dargestellt, um den Entwicklern und auch den Nutzer eine Abstraktion ihrer Vorstellungen zu zeigen.

- **3 Quadrant** In dritten Quadranten wie im Bild 2.3.1 wird der Code manuell getestet, der in den ersten beiden Quadranten entworfen und getestet wurde. In dritten Quadranten soll der geschriebene Code ausschließlich von Menschen getestet werden, da diese mit all den Sinnen ein System anders wahr nehmen können, als ein Skript. Die Nutzer erkennen, ob das System den Ansprüchen, dem Look-and-Feel entspricht und zugleich kann dem Entwickler Feedback über das Implementierte gegeben werden. Diese Art des Testens wird UAT (User Acceptance Test, zu deutsch Nutzer Akzeptanz Test) genannt.[1] Diese UAT Tests gibt es in vielen Varianten. Die einfachste Variante ist das manuelle Testen der Nutzer selbst. Dabei wird mit einem Skript die Datenbank aufgesetzt, um eine produktive Umgebung zu erzeugen.
  Eine weitere Art sind die Usabiltity Tests. Wobei eine Gruppe von Nutzer gesucht wird, die das System testet und anschließend ihre Erfahrungen über das System mündlich oder durch Fragebögen austauschen.
  Die letzte Art des Testens sind die Exploratory Tests (zu deutsch Erforschender Test) und dienen dazu mögliche Begrenzungen herauszufinden. Ein Tester versucht durch kritisches Denken über die Software und seine Eingaben einen Fehler zu produzieren. Bei Exploratory Tests wird vor dem Test genau überlegt, wo mögliche logische Schwachstellen im System sind. Ein Tester wird zu einem

[4] zu Deutsch auch als Attrappe bezeichnet, dient zur Darstellung des Codes, um schnell Feedback über die Struktur der Buttons, Links oder Navigation einer Webseite zu geben[9]
[5] Ähnlich wie Mockups aber vom Funktionsumfang größer und daher für Präsentationen geeignet[9]

Nutzer, da er das gleiche Produkt bekommt und es benutzt wie der Nutzer. In dieser Art des Testens werden für gewöhnlich sehr kritische Fehler gefunden, da Eingaben getätigt werden, die nicht zu einen normalen Nutzerverhalten gehören.

- **4 Quadrant** Im letzten Quadranten wird anders als bei den vorherigen Quadranten die Geschäftsseite nicht mit einbezogen, sondern ausschließlich die technologische Seite betrachtet. Die technologische Seite steht für die kritischen Produkte des Systems, wie Performanz, Sicherheitsfaktor und Robustheit.[1] Tests, die die kritischen Produkte testen, werden von Tester ausgeführt, der sich proaktiv bei der Codeerstellung beteiligt und für die technologische Seite verantwortlich ist.[1] Der Tester wird die Tests verwenden die in den vorherigen Quadranten benutzt wurden. Darunter fallen auch die erforschenden Tests, die im dritten Quadranten benutzt werden, um mögliche Begrenzungen sicherzustellen. Diese Tests werden ebenfalls fürs Lasttesten eingesetzt, um bei kritischen Eingaben die Robustheit des Servers zu testen. Dabei werden wie im Bild 2.3.1 ersichtlich die Tests zu einem Performanztest erweitert, indem unteranderem Tools/Frameworks (wie zum Beispiel Locust, Apache JMeter,..) benutzt werden, um mehrere Aktionen parallel auszuführen und dadurch Last auf den Server zu erzeugen. Das Erhöhen der Last oder auch das Ausreizen der Schranken, die vom Server vorgegeben werden, können zu neuen Fehler führen, die durch die anderen Tests der vorherigen Quadranten unentdeckt blieben.

2.4 Vorteile des iterativen Testens

Nachdem der iterative Ansatz erklärt wurde, wird erläutert, worin sich die Vorteile des iterativen Ansatzes gegenüber dem späten Testens der traditionellen Softwareentwicklung liegen.
Die Erste der beiden Schwächen im traditionellen Ansatzes das späte Testen wird im agilen Ansatz durch kleinere qualitativ hochwertigere Prototypen behoben. Diese Prototypen passen sich an den Anforderungen der Nutzer an. Dabei wird nicht nur auf die Geschwindigkeit des Fortschritts bei der Implementierung geachtet, sondern primär auf die Qualität.[1] Durch kurze Iterationen von einer Woche bis maximal einem Monat wird ein Prototyp vorgestellt, der vom Testteam getestet wird.[1] Erst wenn das Testteam durch seine Tests die Qualität sicherstellen konnte, wird darauf weiterentwickelt. Die Tests dienen als Qualitätscheck, was für den Entwickler als

schnelles Feedback dient. In diesem Feedback liegt der wesentliche Vorteil gegenüber dem späten Testen. Wenn der Entwickler, nachdem er ein Code-Modul veröffentlicht hat, ein Feedback über die Wertigkeit seines Codes bekommt, kann er dieses benutzen und den Code ausbessern. Wobei das Feedback auch für mögliche Neuentwicklungen genutzt werden kann. Wenn das Feedback zu einer neuen Funktion kritische Fehler aufzeigt muss der Entwickler dafür sorgen, diese Fehler auszubessern. Zeigt das Feedback aber, dass die Fehler nicht kritisch sind und eher als Verbesserung vorgeschlagen werden können, kann der Entwickler selbst einschätzen und einbauen. Im späten Testen geschieht dieses Feedback erst nach dem großen Entwicklungsschritt, wo das Ausbessern oder Weiterentwickeln mehrerer Codesequenzen dementsprechend teuer wird.
Bis jetzt wurde lediglich das Testteam für die Qualitätssicherung und fürs Testen betrachtet, währenddessen wird für den iterativen Ansatz auch auf die Unterstützung der Entwickler vertraut. [1] Das Testteam setzt sich mit de m Kunden und dem Entwicklerteam zusammen und bespricht schon am Anfang, wie das Programm den Geschäftsprozess unterstützen soll und schließt mit Anforderungskriterien ab, die ebenfalls die Tests unterstützen und formen.[1] Diese ständige Kommunikation zwischen Tester und Entwickler muss vorhanden sein, um die Qualität zu sichern, da durch die neu entstehende Sicht der Entwickler und der Tester die Tests erweitert werden. Somit sind die Tests nicht mehr eine verordnete Prüfinstanz, sondern eine spezialisierte Dienstleistung (z.B. für Last- und Performance-Tests) oder eben als ein Service genutzt, das frühzeitig im Projekt eingebunden wird und zur Qualität beitragen soll.[5] Diese Kommunikation und das Miteinander in einem Team ist im Testverhalten der traditionellen Softwareentwicklung am Anfang zwar vorhanden, aber nicht im Hinblick auf Tests, die für Qualitätssicherung vorhanden sein sollen. Dies ist die weitere Stärke des agilen Ansatzes gegenüber der traditionellen Entwicklung. Das ein Team mit den Testern, Entwicklern und auch den Kunden bei der Testerstellung zusammenarbeitet.

3 Begriffe in Verbindung mit Performanztests

Im dritten Kapitel wird das nötige Vokabular für den im fünften Kapitel ausgeführten Performanztest erklärt. Darunter fallen sämtliche Begriffe der Testarten, Stufen der Nutzeranzahl und KPIs.

3.1 Indikatoren

Indikatoren werden oft als Schlüsselkennzahlen, Maßzahlen oder Key Performance Indicators (KPI) bezeichnet.[10] KPIs sind eine Teilmenge von SLA (Service Level Agreement) und sind beim Testen unausweichlich, da sie der Businessseite zeigen was das System leistet. Diesen Werten wird eine große Verantwortung zu teil, da sie einerseits vergleichsweise komplexe, betriebswirtschaftliche Sachverhalte vereinfacht und als Zahlen kenntlich machen und dadurch Vergleiche ermöglichen.[11] Andererseits sind KPIs als Gruppenübergreifende Vergleichsdaten zu sehen, die jeder Abteilung ermöglicht die Performance des Systems anhand von Zahlen zu bewerten und auch zu verstehen. Bezüglich der Fokussierung der Leistungsbetrachtung auf Dienstverfügbarkeit bzw. effizienter Ausführung lassen sich diese Indikatoren nach Ian Molyneaux wie folgt gruppieren und charakterisieren:

- effizienzorientierte Indikatoren
  - Datendurchsatz.
  - Ressourcennutzung.
- serviceorientierte Indikatoren
  - Verfügbarkeit.
  - Antwort- bzw. Reaktionszeit (Systembezogen).

Nach dieser Gruppierung fokussieren serviceorientierte Indikatoren die Problematik der effektiven Verfügbarkeit von Anwendungen oder Diensten für den Endbenutzer. Es wird betrachtet wie das System mit dem Nutzer interagiert und die geforderten Ressourcen zur Verfügung stellt.[10] Effizienzorientierte Indikatoren hingegen charakterisieren die Art und Weise der Nutzung und des Bedarfs von Ressourcen durch die Anwendung. Diese Indikatoren repräsentieren das System und zeig wie dieses die vorhandenen Ressourcen nutzt und dem Nutzer den Service anbietet.
Für den späteren Performanztest ist der serviceorientierte Indikator interessant, da die Nutzer einen Service in Anspruch nehmen und bei dem eine Bewertung erfolgt. Der Indikator wird mit der Messzahl „response-time"(zu deutsch Antwortzeit) bewertet. Die Antwortzeit gibt die Zeit an, die das System benötigt, um die Anfrage des Nutzers zu bearbeiten, oder aus der Sicht des Nutzers gesprochen, die Zeit die gewartet werden muss und keine weiteren Aktionen am System durchgeführt werden können. Dieser Indikator fließt meist nicht in die Wertung ein, da die restlichen Indikatoren, wie Verfügbarkeit und Ressourcennutzung wichtiger sind.[10] Das aber die Antwortzeit des Servers ein wichtiger Faktor ist steht nach Ian Molyneaux außer Frage. Der Endnutzer könnte aufgrund dieses Indikators seine nächsten Schritte auf der Web Applikation nochmals überdenken. Laut seiner Meinung würde das Nicht-Beachten der Antwortzeiten eines Systems ungeahnte Folgen für die Betreiber des Webservice[1] nach sich ziehen. Wenn dem Nutzer eine schlechte Performanz widerfährt, wird der nächste Klick wahrscheinlich auf den Webservice der Konkurrenz stattfinden.[10]
Um eine abschließende genaue Bewertung und gegebenenfalls Optimierung der Benutzer-Produktivität vornehmen zu können, muss zunächst die Antwort- und Reaktionszeit als Maßzahl geklärt werden, um sie interpretieren zu können. Hierzu lassen sich die Ergebnisse aus den Untersuchen von Ian Molyneaux heranziehen, der anhand aus textbasierten Applikationen seine Resultate erschließen konnte. Diese Antwortzeiten, die als Grundlage unserer späteren Evaluation dienen, werden in t (für Zeit) festgelegt:

- **Antwortzeit t > 15 Sekunden** Wartezeiten von mehr als 15 Sekunden sind inakzeptabel. Wenn die Antwortzeit eines Systems auch durch Weiterentwicklungen nicht reduziert werden kann, sollte eine auf andere Techniken zurückgegriffen werden. Ein Wechsel von synchron auf asynchron wäre von Vorteil, sodass der Nutzer während der Antwortzeit auf dem Server weitere

[1] Ein Webservice ist ein Dienst, welcher die Zusammenarbeit verschiedener Software-Anwendungen ermöglichen soll, die auf verschiedenen Plattformen laufen. Beispielsweise könnten durch Webservices Anwendungen mit einander kommunizieren, selbst wenn diese in unterschiedlichen Programmiersprachen geschrieben sind oder auf unterschiedlicher Hardware basieren.[12]

Aktionen ausführen kann.

- **Antwortzeit $4 < t < 15$ Sekunden** Diese Antwortzeit wäre für einen konversationartigen Arbeitsvorgang zwischen Nutzer und Server zu lang und würde die durch Beanspruchung vom Kurzzeitgedächtnis des Nutzers gestört werden. Weiter würde solch Antwortzeiten die Wartung des Systems von einem Entwickler hemmen oder die Eingabe von wichtigen Werten in die Datenbank zur Frustration führen. Bei Anforderungen mit weniger zeitabhängigen und weniger Konzentration fordernden Arbeitsabläufen wäre diese Antwortzeit tolerierbar.
- **Antwortzeit $2 < t < 4$ Sekunden** Benutzer werden bei Arbeitsvorgängen die höhere Konzentration benötigen, gestört. Diese Antwortzeit wäre bei Validieren nach Eingabe von Kreditkartennummer und Adresse tolerierbar aber nicht bei der Auswahl der Zahlungsart.
- **Antwortzeit $t < 2$ Sekunden** Wenn der Nutzer mehrere Arbeitsabläufe hintereinander erledigt die über einen bestimmten Zeitraum höhere Konzentration erfordert, sollte die Antwortzeit relative gesehen kurz sein. Je größer die Information an die sich erinnert werden muss, desto wichtiger eine Antwortzeit unter 2 Sekunden.
- **Antwortzeit $t < 1$ Sekunde** Ein Nutzer setzt ein bereits hochgeladenes Bild als Profilbild ein, diese bereits geladenen Informationen sollten nahezu sofort geändert werden können.
- **Antwortzeit $t < \frac{1}{10}$ Sekunde** Der Klick auf einen Schaltfläche sollte eine sofortige (unter 0.1 Sekunden) Aktion herbeirufen. Als Beispiel kann ein Computerspiel aufgezählt werden, dass nachdem ein Aktion angestoßen wurde, sofort eine Reaktion erkennbar ist.

Somit steht die Grundlage unserer KPI, die wir in dem späteren Lasttest noch benötigen werden, um das System möglichst genau bewerten zu können.

3.2 Erreichbare Ziele

Targets (zu deutsch Ziele) sind für mehrere Personen einer Softwareentwicklung wichtig, da sie ein Ziel angeben das erreicht werden soll. Dabei liegt die Wichtigkeit der Ziele nicht nur beim Kunden oder beim Tester.
Das Testteam sieht in diesen Zielen mögliche Anhaltspunkte welche Skripte sie benutzen, um die Leistung zu testen und um abklären zu können, wann das System für

die Nutzer bereit ist. Der Nutzer selbst soll durch die UAT (zu deutsch Akzeptanztest) die Möglichkeit besitzen sich ein Bild von dem Entwickelten zu machen und selbst zu entscheiden welchen Weg die Entwicklung einschlagen soll.[1]
Dabei ist auch die Abwägung von Leistung entscheidend. Laut der Meinung von Molyneaux ist für ein Unternehmen wichtig ein Ziel festzulegen. Jede Gruppe die mit dem System interagiert, auch die Nutzer selbst sollen bei der Zielerstellung beteiligt sein.[10] Dieses Ziel wird durch das Management, welches sich aus verschiedenen Abteilungen zusammensetzt, entschieden und soll für das Entwicklerteam, Testteam und Endnutzer Leitfaden sein.
Dabei wird diese Zielsetzung (oder auch Key Performance Targets) laut Molyneaux so beschrieben:

- service-orientierte Ziele (im Sinne von Schnelligkeit)
  - Verfügbarkeit(Availability/uptime).
  - Skalierbarkeit oder Nebenläufigkeit.
  - Antwortzeiten.
- Leistung im Sinne von Kapazität
  - Netzwerk Nutzung
  - Server Nutzung.

Diese Auflistung einzelner Ziele muss nicht unbedingt eigenständig betrachtet werden. In der Praxis wird eher eine Kombination dieser Ziele als Grundziel gesehen. Zum Beispiel: Ein verteiltes System soll gut skalieren, die Nebenläufigkeit möglichst transparent halten, dabei eine hohe Verfügbarkeit besitzen und den Nutzer schnell Resultate liefern.

3.3 Verschiedene Arten des Performanztests

Es gibt ein gewisses Repertoire an Testmethoden die benutzt werden können, um einen Performanztest auszuführen. Dabei wird das „Wie“ auf zwei Unterkategorien aufgeteilt. Zu Einem beschreiben die Testmethoden in welcher Art und Weise getestet werden kann und was in dem Ergebnis der Testmethode interpretiert wird. Zum Anderem wird das Intervall erklärt, mit welcher Anzahl an simulierten Nutzer getestet werden kann.

3.3.1 Testmethoden

Bei der agilen Herangehensweise kann unter vielen verschiedenen Testmethoden unterschieden werden, dabei ist eine strikte Trennung je nach Ziel einzuhalten. Laut Molyneaux können unter folgenden Testmethoden unterschieden werden:

- **Baseline test:** Dieser Test gilt als Grundtest und dient als Grundlage für weitere Vergleiche zu anderen darauffolgenden Tests. Für gewöhnlich wird dieser Test von einem einzigen virtuellen Nutzer, der während des Testes jeweils nur eine Transaktion[2] in einer bestimmten Periode durchführt.[10] Dieser Test wird als der „best case"-Testlauf angesehen. Das System kann sich voll und ganz auf nur einen Nutzer konzentrieren und seine Anfragen zügig nachgehen und mit maximaler Geschwindigkeit beantworten. Mit diesem Test kann ungefähr berechnet werden, wie das System mit einer größeren Anzahl an Nutzern ausgelastet wird, bevor weitere Tests durchgeführt werden müssen, die einen Systemabsturz ermöglichen können.[10]

- **Lasttest:** Auf diese Art von Test wird in meiner Arbeit besonders Wert gelegt, da dies die klassische Art des Performance Testes ist.[10] Mit jenem Test wird der gewöhnliche Gebrauch einer Software oder eines Systems getestet. Simultanes Arbeiten von mehreren virtuellen Nutzern, die das System unter Last setzen und über die Antwortzeiten des System Rückschlüsse gezogen werden können. Simultanität schließt Verzögerungen und Pausen eines virtuellen Nutzers bei der Eingabe ein.[10]

- **Stresstest:** Der Stresstest ist dem Lasttest ähnlich, dennoch gibt es einen wesentlichen Unterschied: Stress. Ein Stress Test versucht, anders als ein Lasttest, das System nicht auf das äußerste zu testen, sondern die Grenzen zu überschreiten und aufzuzeichnen wie das System darauf reagiert. Die Absicht ist das obere Limit der Infrastruktur des Systems herauszufinden. Stresstest bricht erst ab, wenn irgendwas bricht, sei es, dass sich keine Nutzer mehr einloggen können, oder die Antwortzeit wird so hoch, dass flüssiges Arbeiten nicht mehr möglich ist (...).[10] Die Resultate dieses Testes können unterschiedlich interpretiert werden, da der Test die Performanz und die Kapazität zugleich testet. Kapazität meint hierbei, wie viele Nutzer über den gedachten Limit für das System noch ertragbar ist, wenn in Zukunft die Anzahl der Nutzer steigt.

[2] Ist eine Folge von Arbeitsschritten die nach fehlerfreier und vollständiger Ausführung in einen wohldefinierten und konsistenten Zustand überführen

- **Stabilitätstest:** Der Stabilitätstest soll Aufschluss darüber geben, welche Probleme nach längerer Laufzeit des Systems oder durch einen unvorhergesehnen Anstieg an Nutzer auftritt. Ein Beispiel eines möglichen Fehlers wäre ein Speicherleck(englisch memory leak). Ein verteiltes System nutzt mehrere RAM[3] (oder Schnellspeicher), um dort Berechnungen auszulagern. Dieser Speicher könnte mit Laufzeit des Systems voll werden, da Reservierungen von Speicher durch Prozesse nicht benutzt und später auch nicht entfernt werden. Aufgrund der Überbelegung des Speichers könnte es zum Anstieg der Antwortzeit kommen und im schlimmsten Fall den Ausfall des Systems begünstigen.
- **Smoketest:** Smoke Test wird nur nach Codeänderungen durchgeführt, um zu testen, ob die Grundlegendsten Funktionen ohne Fehler durchgeführt werden. Diese Art von Test soll mögliche Fehler aufzeigen, die nach Hinzufügen einer Erweiterung auftreten können.
- **Isolationtest:** Isolationtests werden benutzt, um Arbeitsabläufe oder Transaktionen eines Codefragments genauer zu untersuchen, welches einen Einriss in die Performance verursacht.

3.3.2 Intervall der simulierten Nutzer

Jede Testmethode kann mit einer unterschiedlichen Anzahl an simulierten Nutzer ausgeführt werden. Im Wesentlichen wird die Realität des produktiven Systems nachgeahmt. Dieser Vorgang geschieht durch Pacing (zu deutsch: Angleichung).[10] Es wird versucht mit Pausen und Verzögerungen die Handhabung mit dem System nachzuahmen. Ein realistisches Nutzen eines System schließt mit ein, dass zwischen verschiedenen Arbeitsabläufen Zeit zum Überlegen gelassen wird. Ein kleines Beispiel: Wenn man auf Amazon einen Einkauf tätig werden die meisten Nutzer nicht sofort den Artikel kaufen, den sie sich wünschen. Vielmehr werden zuerst andere Artikel angeschaut und erst nach Abwägung von Preis/Leistung wird abgerechnet. Besonders bei der Eingabe von Kreditkartennummer und Personalien vergeht einige Gedenkzeit, bevor die nächste Aktion gesetzt wird.
Dabei ist es entscheidend Pausen auch beim Lasttest oder Stresstest immer einzubauen, um den Nutzer bestmöglich zu simulieren.[10] Im Folgenden wird zwischen 5 Typen der Angleichung unterschieden:

[3]Englisch Random Access Memory wird mit RAM abgekürzt und bezeichnet einen Datenspeicher der in der Laufzeit des Computer/Server verwendet wird.

- **Big Bang** Dieser Modus eines Testes startet mit einer gewissen Menge an virtuellen Nutzer, die zur gleichen Zeit, jedoch nicht synchron unterschiedliche Aktionen auf dem System ausführen. Durch die Asynchronität der Arbeitsabläufe wird zusätzlich ein menschlicher Nutzer simuliert.[10]
- **Ramp-up** Dieser Modus, der auch als die gewöhnlichste Art des Testen gesehen wird, startet mit einer kleinen Mengen an virtuellen Nutzer (0 oder 1 Nutzer) die in einem speziellen Intervall eine angegebene Maximalanzahl erreicht.[10]
- **Ramp-up (with step)** Dieser Modus nimmt das Ramp-up als Basis und fügt eine Pause zwischen den Anheben der Nutzeranzahl ein.[10] Diese Pause kann dazu genutzt werden, um einen kleinen Performanztest auf Basis der aktuellen Nutzerzahlen durchzuführen, dies geschieht innerhalb eines größeren umfassenden Performanztestes. Dadurch kann erkannt werden, wie das System den Nutzerzahlen abhängig, skaliert. Nicht alle derzeit existierenden Softwarelösungen beinhalten diese Funktion.
- **Ramp up (with step), Ramp down (with step)** Diese Erweiterung des Ramp-up mit steps führt eine Stagnation der Nutzerzahlen aus, um das System die Last abzunehmen. Dabei kann mit oder ohne Schritte die Stagnation der Nutzerzahlen durchgeführt werden.[10]
- **Delayed start** Dieser Modus beschreibt einen verzögerten Start des Testes (delay zu deutsch: Verzögerung) und kann mit jedem vorherigen Typen kombiniert werden. Mögliche Gründe für den Einsatz dieses Modus kann sein, dass ein Skript vor der Angleichung gestartet werden muss, was einiges an Zeit in Anspruch nehmen kann.[10]

4 Hybrider Test Ansatz mit sozialen Aspekten

In diesem Kapitel wird der Ansatz besprochen, der auf den Performanztest von vierten Quadranten des agilen Ansatzes basiert. Ich erweitere den agilen Ansatz mit den sozialen Aspekten von James Bach, als eine Erweiterung des Testens. Indem er eine Kommunikation zwischen Entwickler und Tester erklärte.[2] Diese Herangehensweise benutzt die schon bestehenden Arten und Methoden, aber beschreibt eine neue Anschauung wie an den Test herangegangen wird, nämlich durch ein klärendes Gespräch zwischen dem Testteam und Entwicklerteam.
In seiner Methodik geht James Bach von „innen nach außen“ (englisch inside-out), womit das Verstehen der Situation und das Abwägen der Gefahrenquellen gemeint ist. Um das Prinzip zu verstehen sollte sich der Code als das Innere einer Funktion betrachtet werden (inside) und das was der Code produziert, sprich die Aktionen die getätigt werden, sollte als das Äußere (out) betrachtet werden. Hierbei wird überlegt was in dem produzierten Code falsch gehen kann und welche Folgen diese Fehler nach sich ziehen.[2]
James Bach gibt noch eine weitere Überlegung an, die in den meisten Fällen auch praktiziert wird, die sogenannte (outside-in). Diese funktioniert entgegengesetzt der vorherigen Methodik, in der eine Liste aus verschiedenen Schwachstellen betrachtet wird und entlang dieser Liste getestet wird. Diese Liste von Schwachstellen kann von vornherein von der Geschäftsseite erstellt worden sein, um die Qualität sicherzustellen, oder es existieren ähnliche Fehlerquellen, die schon öfters vorkamen. Da bei diesen Ansatz schon eine gewisse Erfahrung zu den Fehlern vorhanden sein soll, kann dies in unseren Fall nicht eingesetzt werden, da das System neu entworfen wurde und es deswegen keine bekannten Fehlerquellen existieren. Weiter wird die Inside-Out Methodik bei Tests benutzt die recht speziell sind und wo es noch keine Erfahrungen diesbezüglich gibt. In unseren Fall ist es das Ausreizen der Last die auf ein System wirkt und auch das experimentelle Testen um Fehler zu produzieren, was ein sehr spezieller Testfall ist. Deswegen wird auch der erste Ansatz verwendet, bei dem folgende aufgelistete Fragen eine tragende Rolle spielen. Diese sollen immer im

Hinterkopf des Testers sein, wenn dieser ein Gespräch mit den Entwickler eingeht.

- **Schwachstellen:** Was sind die potentiellen Schwachstellen vom programmierten Code?
- **Gefahren:** Welche Eingaben können getätigt werden, die die Schwachstelle aufdeckt und zu einem Fehler führt?
- **Opfer:** Wer und Was würde von dem Fehler betroffen sein und Wie schlimm wären die Konsequenzen?

Diese Fragen sollen zur Erstellung weiterer Testfälle führen, die schnellst möglich die Qualität des Systems sicherstellen sollen, da sie sich auf die maximale Gefahrenquelle beziehen und dafür Tests geschrieben werden. Mit diesen Tests können genau diese Gefahrenquellen, durch die der Kunde an weitere Aktionen gehindert wird, getestet und somit auch die Qualität gesichert werden. Dabei ist der Wissensaustausch durch die Kommunikation mit dem Entwickler sehr wichtig, wie diese aussieht wird im Folgenden erklärt. Vor dem Wissensaustausch sollte sich der Tester Gedanken über die Funktionsweise des Programm machen, was schlussendlich entwickelt wird. Ein Tester dient als Schlüsselstelle zwischen Entwickler und Kunde um Fehlerfälle abzufangen und die Qualitätsansprüche der geforderten Ansprüche zu garantieren. Ist dies nicht der Fall muss solange getestet und nachgebessert werden bis die Anforderungen erreicht wurden. Was der Tester hierbei wissen muss, ist wie der Kunde das Produkt schlussendlich benutzen wird und welche Probleme dem Kunden möglicherweise an der Produktivität hindern. Im Gespräch zwischen Entwickler und Tester soll der Tester die Chance nutzen das Programmierte zu verstehen und die Schwachstellen herauszufinden. Dabei sollte eine leicht distanzierte Stellung zu dem Produkt seitens des Testers eingenommen werden und mit möglichst vielen Fragen die Randfälle des Programms herausgefunden werden. Dabei spielt das Fragen eine wesentliche Rolle, da durch das Fragen der Tester ein gutes Bild von dem Programmierten an sich bekommt und über die Gefahrenquellen mehr und mehr in Erfahrung bringt.[13] Dazu soll auch sichergangen werden, dass die Anforderungen seitens der Kunden im Programm berücksichtigt wurden. Besonders wichtig hierbei ist das generelle Verhalten des Entwicklers, welche Wortwahl dieser benutzt und wie sehr er durch die Fragen gestresst wird. All dies kann Aufschluss darüber geben, wie sehr sich der Entwickler über die Anforderungen Gedanken machte, was bei zögernden Verhalten oder Unwissenheit wiederum Ansätze für weitere Tests empfiehlt. Seitens der Entwickler kann bei Fragen ein Nachdenken angeregt werden, ob es neben seiner noch eine schnellere oder sicherere Lösung existiert. Ziel diesen Wissensaustausches soll eine klare Bild der Funktionsweise vom Programm existieren, mit dem der Tester Testfälle produzieren kann.

Dieser Ansatz wurde im Gespräch mit den Entwicklern in der Industrie angewendet. Das Thema des Gesprächs war die Steigerung der Performanz der Webapplikation. Dabei wurde das natürliche Nutzerverhalten im Bezug auf Suchen und Übertragen von Inhalten der Webapplikation als äußerst effektiv beschrieben, um die Performanz des Systems zu testen. Das Verhalten der Nutzer auf der Webapplikation setzt sich aus Suchen von Inhalt über ein Suchwort und das Anschauen in unterschiedlicher Auflösung zusammen und soll in einem Performanztest abgebildet werden. Diese Überlegung das Nutzerverhalten so gut es nahe zu kommen, sorgte für den Algorithmus, der im folgenden Unterkapitel erklärt wird.

4.1 Algorithmus des simulierten Nutzers

Der Algorithmus besteht einerseits aus der Google Search Statistik und dem Framework[1] Locust, welche beide im Folgenden erklärt wird. Die Google Search Statistik konnte verwendet werden, da die Abfrage der Datenbank vom Unternehmen vergleichbar ist mit der Suche eines Begriffes in Google.[15] Das System das getestet wird, zeigt auf der Startseite eine bestimmte Anzahl an Videos an, diese Anzahl wird durch Filter, die der Kunde setzen kann, reduziert. Setzt der Kunde keine Filter mehr, hat dieser die Wahl ein Video aus der Menge übergebliebener Videos auszuwählen. Diese Anzahl, die auf einer Webseite angezeigt wird, ist auf 10 Videos begrenzt und das weitere Laden neuer Videos folgt dem gleichen Schema, welches sich in Google Suche wiederfindet. Diese Art der „Pagination“ (zu deutsch Seitenumbruch) ist State of the art und findet dort Verwendung, wo durch mehrere Objekte durchgezählt werden muss. Wird die Google Search Statistik genauer betrachtet wird erkennbar, dass diese nicht nach einer Funktion, sondern nach Klickergebnissen berechnet wurde. Diese Werte wurden mit Hilfe der Suchmessungen der Nutzer errechnet, so gesehen eine kleine Fallstudie welchen Link der Nutzer nach der Eingabe seines Suchbegriffes klickt. Aus dieser Statistik kann die Wahrscheinlichkeit entnommen mit der der Nutzer den ersten Link und den letzten Link ausführt.
Wegen der Einfachheit des Codes und um die Berechnung der Wahrscheinlichkeiten dynamischer zu gestalten wurde eine Funktion gesucht, die der Interpolation der Google Statistik ähnlich ist. Diese wurde in Windows Excel gefunden und steht unter den Synonym „Power“. Diese Funktion wird in (4.1) vorgeführt, welche für unsere

[1] Ein Framework ist ein modernes Rahmenwerk, das dem Programmierer den Entwicklungsrahmen für seine Anwendungsprogrammierung zur Verfügung stellt und damit die Software-Architektur der Anwendungsprogramme bestimmt.[14]

Zwecke vollkommen ausreicht. Die verwendete Funktion beschreibt die Wahrscheinlichkeit (y) mit der der Link (x) ausgeführt wird.

$$y = 35,61 * x^{-1,128} \tag{4.1}$$

In der Statistik wird ein weiterer Abschnitt erkennbar, dass ab dem Seitenumbruch (Links 11-20) die Klickwahrscheinlichkeit auf annähernd 0 Prozent fällt, welches 143% weniger ist als die Wahrscheinlichkeit den 10 Link zu drücken. Dieser Abschnitt wird auch in meiner Implementierung mit der Funktion (4.2) ausgedrückt, welche gleich gehandhabt wird wie die Funktion 4.1, nur mit anderen Werten.

$$y = 3,8609 * x^{-2,466} \tag{4.2}$$

Diese Funktionen dienen als Basis des simulierten Nutzer und werden in Framework Locust verwendet. Bevor die weiteren hilfreichen Funktionen von Testframework Locust erklärt werden, die den simulierten Nutzer noch realistischer gestalten, sollte Locust an sich noch kurz eingeführt werden.

4.2 Locust

Locust ist ein kleines Framework welches auf der Skriptsprache Python basiert und wurde für Lasttests konzipiert.[16] Die Idee hinter Locust ist eine Webseite mit mehreren hunderten Nutzer gleichzeitig anzusteuern und dadurch Last zu erzeugen, die das System unter Druck setzt. Dabei bietet Locust die Möglichkeit durch eine Weboberfläche die Nutzeranzahl aktiv zu verfolgen und auch abhängig vom System die Last zu erhöhen. Die markantesten Punkte liegen aber bei den zusätzlichen Funktionen der Ausführwahrscheinlichkeit und Wartezeit, die die Entwickler von Loucst einbauten, um Nutzerverhalten zu simulieren. Dabei soll die Wartezeit das menschliche Nachdenken zwischen einzelnen Schritten simulieren und die Ausführwahrscheinlichkeit eine spezielle Gewichtung der Arbeitsschritte erlauben. Der letzte Punkte soll noch weiter erläutert werden. Ein kleines Beispiel welches die Ausführwahrscheinlichkeit erklärt ist das Schauen eines Videos auf Youtube. Ein Nutzer schaut das Video und kann währenddessen die Auflösung des Videos verändern, Untertitel einblenden, das Video auf anderen Plattformen teilen, oder auch das Video auf seinen Account abspeichern, um es später anzuschauen. Diesen Funktionen kann eine gewisse Gewichtung gegeben werden, damit diese auch ausgeführt werden. Somit wird ein Nutzerverhalten nachgeahmt, das die implementierten (optionalen) Zusatzfunktionen benutzt und das System zusätzlich mehr unter Last setzt.[16]

4.2.1 Wartezeit und Ausführwahrscheinlichkeit

Locust bietet die Möglichkeit zwischen jedem Arbeitsschritt eine zusätzliche Wartezeit und Ausführwahrscheinlichkeit zu erstellen. Dabei funktioniert die Wartezeit nach einem einfachen Prinzip, bei der der Entwickler eine Zeitspanne definiert, die aus den minimalen und maximalen Wartezeiten besteht. Locust wählt dann zwischen den beiden Schranken willkürlich eine Zahl aus, die gewartet werden muss, bis der nächste Arbeitsschritt begonnen werden kann. Mit diesen Wartezeiten wird zwischen den einzelnen Arbeitsschritten, das Warten sichergestellt. Dabei ist die Angabe der Wartezeit in Millisekunden anzugeben, wobei als „default“ (=Grundwert) der Nutzer eine Sekunde zwischen den einzelnen Arbeitsschritten wartet. Durch diese Wartezeit kann das Verhalten des Menschen noch besser simuliert werden, da zwischen den einzelnen Schritten immer eine gewisse „Gedenkpause“ besteht.

Zu der Wartezeit kann jedem Arbeitsschritt eine Wahrscheinlichkeit gegeben werden mit der dieser ausgeführt wird. Dabei wird jedem definierten Arbeitsablauf eine Zahl hinzugefügt, mit der die Wahrscheinlichkeit berechnet wird. Diese Zahl ist die Gewichtung, mit der der Tester einzelne Arbeitsschritte öfters ausführen lässt als andere. Es werden zwar alle Arbeitsschritte in ungeordneter Reihenfolge ausgeführt aber mit einer bestimmten Gewichtung. Diese Gewichtung legt der Entwickler fest indem er die Zahl an den Arbeitsschritt erhöht.

4.2.2 Zusammenfassung

Zusammenfassend hat der Algorithmus, der den Nutzer simuliert, drei Konfigurationsparameter. Der erste Parameter ist die Statistik der Google Search, die die Klickhäufigkeit der Nutzer nach getätigten Suchwort widerspiegelt. Die anderen beiden Parameter werden von Framework Locust bereitgestellt, das eine Ausführwahrscheinlichkeit und Wartezeit zur Verfügung stellt. Durch das Einstellen dieser Parameter lässt sich ein Nutzer simulieren, der nach einem Suchbegriff das erste Video mit hoher Wahrscheinlichkeit abspielen lässt und darauf basierend weitere Tätigkeiten mit unterschiedlicher Gewichtung nach einiger Gedenkzeit ausführt.

4.3 Reportingeigenschaften

James Bach kritisiert das Quadranten Modell des agilen Ansatzes. Seiner Meinung nach ist die Entwicklung einer Software sowie das Testen dieser nicht strikt nach einem gewissen Schema zu führen.[17] Das strikte Folgen der Quadranten und demnach auch die Informationsbefriedigung der Geschäftsseite sei garnicht nötig.[17] Die Informationsbefriedigung wird durch das Generieren von Reports auf Basis der durchgeführten Tests ermöglicht und den Geschäftsleuten präsentiert. Nach James Bach ist lediglich die Kommunikation mit den Entwicklern und Demonstration vom Implementierten wichtig, um den Tester das Verständnis des Codierten zu vermitteln.[17] Anders als James Bach, der wie schon erwähnt Reports als unnötig erachtet, wird in diesen hybriden Ansatz das Reporting als äußerst wichtig gesehen und dient auch als Vergleichsmaterial über die einzelnen Versionen eines Systems hinweg. Dabei unterstützt Locust mit der fertigen Weboberfläche, die gleich nachdem Locust gestartet wurde, eine Live Ansicht von Locust selber und den wichtigsten Parameter wiedergibt (Live Monitoring). Dabei fließt jeder Durchlauf eines jeden simulierten Nutzers in eine Wertung ein, die später in Form einer Tabelle auf den ausführenden Computer gespeichert wird. Fehlerhafte Aufrufe, die vom Server nicht bearbeitet werden konnte, oder Aufrufe, die zu einem Fehler im System führten, werden auf der Weboberfläche eigens angezeigt und auch eigens abgespeichert. Wie schon angedeutet muss zwischen zwei wesentliche Arten von Reporting unterschieden werden.

- **Live Monitoring** Zu Einem wird durch das Live Monitoring der Lasttest manuell gesteuert und je nachdem wie sich der Server verhält die Anzahl der Nutzer vergrößert oder verkleinert. Diese Funktion wird in meinen Lasttests besonders häufig benutzt um zu sehen, ob das System mit der simulierten Nutzerzahl zurecht kommt. Ein Lasttest soll das System unter gewöhnlichen Gebrauch testen und dazu nicht zu sehr belasten (Lasttest ist kein Stresstest). Ich benutze mit dem beschriebenen Vorgang eine manuelle Ramp-up mit Schritten, die im agilen Ansatz ausführlich beschrieben wurde.

- **Reporting** Zum Anderen können Reports als Grundlage für Weiterentwicklungen, Vergleichsvariablen und Reporting an sich betrachtet werden. Mit Weiterentwicklungen wird das stufenweise Herangehen an einer für das System stärkeren Last betrachtet, bei der Serverseitig Fehler passieren können, die von Locust aufgefangen und auf der Weboberfläche angezeigt werden. Diese Fehler können exportiert werden und dienen als Grundlage für Weiterentwicklungen, da diese mit dem Entwicklerteam kommuniziert werden können.
  Aber auch als Vergleich zu vorherigen Versionen des Systems kann ein Report

benutzt werden. Die Durchführung eines Performanztestes nach einer vermeintlichen Verbesserung des Systems und der Vergleich des Exportes mit dem vorherigen Export kann Aufschluss darüber geben, wie sehr sich die Veränderungen gelohnt haben. Ist keine Verbesserung hervorgetreten, kann dem Entwickler sofort Feedback gegeben werden und ein anderer Lösungsweg eingeschlagen werden. Schlussendlich kann Reporting dazu benutzt der Geschäftsseite den momentanen Stand der Entwicklungen zu präsentieren, da durch grafische Aufbereitung bildlich die Verbesserung aufgezeigt werden kann. Dazu kommt, dass ein Tester als Qualitätssicherer dient und deswegen die Qualität auch bewerten muss. Reports helfen dem Tester dabei, das Produkt zu bewerten und leichter Vergleiche zwischen den Entwicklungsschritten zu ziehen.

5 Evaluation des neuen Test-Ansatzes

In diesem Kapitel wird ein Performanztest durchgeführt, welcher die Leistung des Systems testet und den im Kapitel 4 erläuterten Algorithmus benutzt. Es handelt sich hierbei, um einen speziellen Test für das System einer Firma, deswegen wurden auch die Resultate des Testes und die Angaben im Setup mit einem Faktor multipliziert, um die Leistung des Systems nicht öffentlich zu stellen. Hierbei ist der Fokus aber nicht auf das Reproduzieren des Testes zu setzen, sondern auf den Einsatz des Algorithmus und die Herangehensweise an das Problem. Schlussendlich wurde aufgrund des Performanztestes mit intelligenten Lasteinheiten das Problem gefunden, dass bei steigender Anzahl der Anfragen an den Server die Anzahl der Requests/s stagniert, da der Speicher des Systems den angefordeten Inhalt nicht ausliefern konnte.

Bevor die Testresultate vorgestellt werden, wird die Webapplikation als Ganzes beschrieben und danach das Setup des Systems beschrieben, welches sich in einem Hardwareteil und einem Softwareteil gliedern lässt. Dabei befasst sich der Hardwareteil mit der verwendeten Hardware des Systems und welche Architektur das System zugrunde liegt. Währenddessen der zweite Teil die Software für den Performanztest beschreibt, wie das Testframework zum Einsatz kommt und wie während des Tests die Performanz kontrolliert wird.

5.1 Das getestete System

Die Firma hat das Videoportal als eine Website und Applikation für Endgeräte konstruiert, um das Präsentieren der intern produzierten Videos der Öffentlichkeit zur Verfügung zu stellen. Ob auf der Webseite oder in der Applikation kann nach einzelnen Suchwörtern gesucht werden und die Videos dazu in unterschiedlichen Formaten

wiedergegeben werden. Der Inhalt kann in verschiedenen Genres, wie Extremsport, Musik, Kultur, Festivals, GoPro und vieles mehr unterteilt werden. Zusätzlich können auch Liveereignisse (= englisch Liveevents Bild 5.1) ausgestrahlt werden, die in einem Jahr passieren werden oder auch auf vergangene Geschehnisse zurückgegriffen werden. Weiters wird die Applikation für verschiedene Endgeräte, wie SmartTV und Konsolen oder auch Smartphones, bereitgestellt. Durch letzteres kann auch mobil auf die Daten zugegriffen und Videos angeschaut werden.

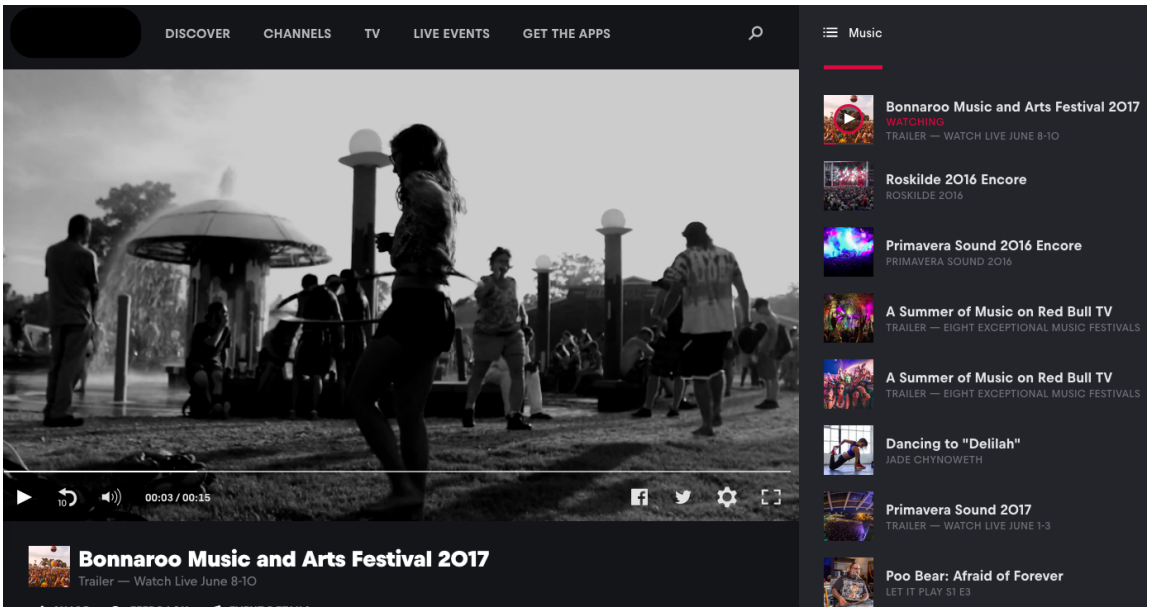

Abbildung 5.1.1: Screenshot von der Weboberfläche

Einen Überblick über die angebotenen Funktionen liefert das Bild 5.1. Der Entdeckermode oder auch das Anzeigen aller verfügbaren Genres (=Channels) ermöglichen dem Nutzer den Inhalt der gesamten Applikation zu sortieren und anzuschauen. Für öffentliche Nutzen kann der TV Bereich genutzt werden, wo willkürlich gewählte Inhalte angezeigt werden. Weiteres wird dem Nutzer auch die Möglichkeit geboten, Videos in unterschiedlicher Auflösung zu konsumieren. Ob jetzt hochauflösendes Full-HD (1080p) oder weniger auflösende Einstellungen wie 720p bis hin zu 240p können manuell eingestellt werden. Dem System kann auch die Kontrolle über die Auflösung gegeben werden, in dem „Auto“ als automatisches Suchen der besten Auflösung ausgewählt wird. Dabei wird je nach Auslastung und Übertragung die Auflösung automatisch vom Endgerät angepasst. Weiter kann sämtlicher Inhalt, wie bei beliebten

Videoportalen (Youtube), auf anderen Plattformen geteilt werden und somit auch vervielfältigt werden.

5.1.1 Hardware des Systems

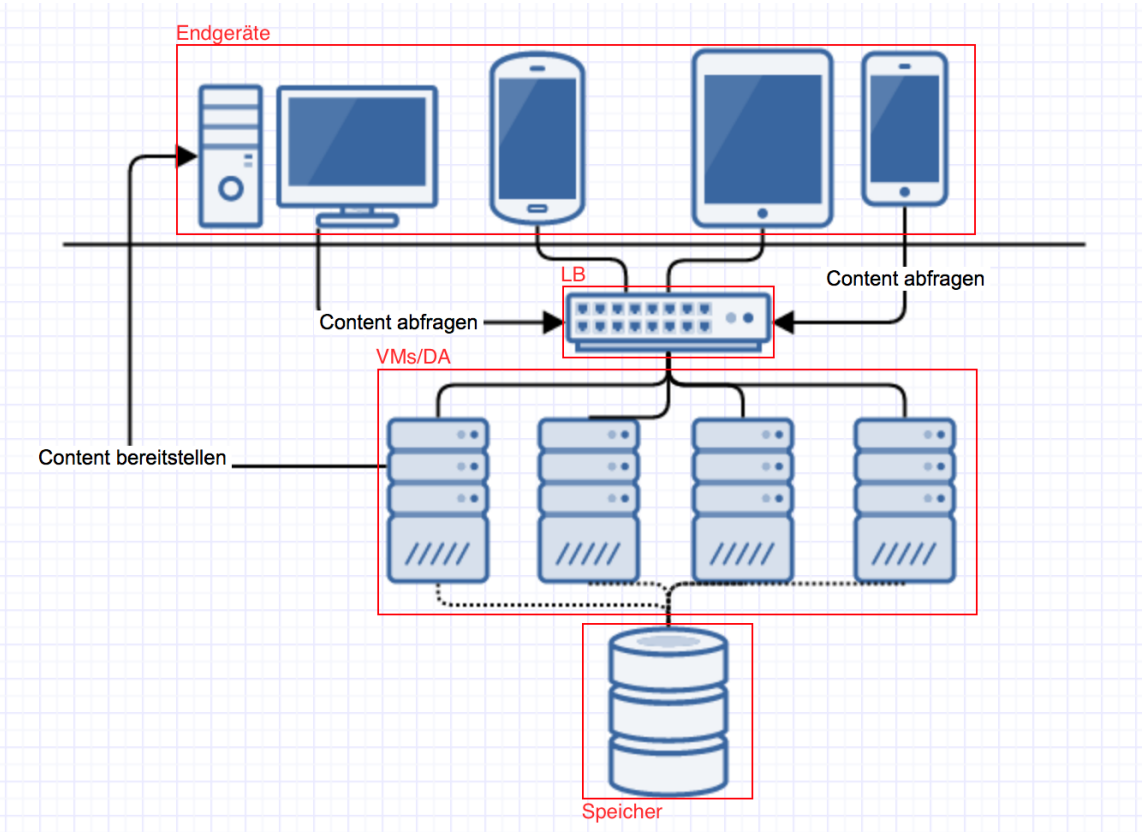

Abbildung 5.1.2: Architektur der Hardware

Der erste Block in 5.1.2 sind die Endgeräte die auch als wichtigster Block gesehen werden kann, da hier der Endkunde das Video streamt und den angebotenen Service bewertet. Hierbei kann jedes Endgerät benutzt werden, welches die Applikation installiert hat oder über die Internetseite die Webapplikation ansteuert. Der zweite Block in 5.1.2 ist der Loadbalancer[1], der für die Skalierung eines System verantwortlich ist. Dabei liegt die Aufgabe des Loadbalancers lediglich darin, die Anfragen zum

[1]Das Loadbalancing oder zu deutsch Lasten Verteilung beschreibt ein Stück Code, welches selbst geschrieben werden kann oder eingekauft wird (fertige Lösungen wären ldirectord oder POUND),

Inhalt einer Webapplikation zu weiteren Server weiterzuleiten. Wurde der Loadbalancer gut entworfen und mit wesentlich viel Logik versetzt, kann ein System die Last schnell aufnehmen und auch schnell auf die Delivery Agents/Virtuelle Maschinen (der nächste Block in 5.1.2) weiter verteilen. Die Testumgebung besitzt mehrere Virtuelle Maschinen, kurz VM. Virtuelle Maschinen sind von der Hardware vergleichbar mit einem herkömmlichen Server, lediglich die Nutzung dieser ist im Gegensatz zu herkömmlichen Server unterschiedlich. Virtuelle Maschinen nutzen nur Teile der Hardware je nachdem wie diese konfiguriert wurden. So gesehen sind VM eine virtuelle Repräsentation von echter verbauter Hardware durch Software. Vorteile der VM liegt beim schnellen Aufsetzen neuer VMs durch einfache Setup Skripte.[18] Diese VMs dienen als Delivery Agents[2], die den Inhalt an die Endgeräte ausliefern, falls diese nicht vom Speicher geladen werden müssen. Der Speicher ist eine Datenbank mit Verbindung zu einem Filesystem und ist der letzte Block in 5.1.2. Die Datenbank beinhaltet Links zum direkten Inhalt des Filesystems.

Wenn eine Anfrage an das System ergeht versucht der Loadbalancer die Anfrage an den einen Delivery Agent weiterzuleiten, der im Vergleich zu den anderen Delivery Agents weniger belastet ist. Ein Delivery Agent besitzt unteranderem eine CPU, Arbeitsspeicher und einen Cache[3]. Wird Inhalt abgefragt, der nicht im Cache liegt wird vom Hauptspeicher abgefragt, was wesentlich mehr Zeit im Anspruch nimmt als die Abfrage vom Cache.

5.1.2 Software des Systems

Zum automatisierten Testen der Systemperformanz wird nicht die Weboberfläche (5.1) benutzt, sondern die API-Schnittstelle, wird stattdessen diese kurz erläutert. Die Nutzung der VM, Loadbalancer und des Speichers wird über eine Schnittstelle durchgeführt. Auf dieser Schnittstelle wird mit REST gearbeitet. REST steht für Representational State Transfer und präsentiert ein Schicht, die es ermöglicht das Client und Server schneller miteinander kommunizieren. Diese Art des Austausches von Information ist weit verbreitet und gehört zudem zum Best Practice der Client/Server

dass die Last auf ein verteiltes System verteilt. Dabei stehen mehrere Rechenknoten einen Loadbalancer zur Verfügung, die die Last aufnehmen können.citeloadbal

[2] „Deliver“ zu deutsch ausliefern, meint im Sinne eines CMS den Inhalt, der für den Nutzer bereitgestellt wird.

[3] Ein Cache ist ein sehr schneller Speicher mit verhältnismäßig kleiner Speicherkapazität, der als Puffer zwischen der Zentraleinheit (CPU) und dem Arbeitsspeicher angeordnet ist.[19]

Kommunikation.[20] Hierbei wird mittels Methoden Typen (Get = Bekommen, Put = Setzen) Datensätze abgefragt oder erstellt. In diesen Datensätzen können Metadaten[4] oder Links zu Videos zu den Delivery Agents widergegeben werden. Der Performanztest nutzt schon fertige Get-Methoden, um den Inhalt abzufragen und fügt weitere Filter hinzu, um Inhalte auch zu mischen. Somit wird keine Oberfläche benutzt, die den Inhalt darstellen muss.

5.1.3 Belastbarkeit des Systems

Das Videoportal muss hohe Lasten aushalten können, da mehrere tausende Benutzer gleichzeitig darauf zugreifen können. Deswegen wird das System einer Last ausgesetzt, um das mögliche Performanz Problem herauszufiltern. Das Problem bei dem System liegt an der Anzahl der Nutzer. Wenn ein Nutzer auf dem System agiert sind keine Engpässe auf den Server zu messen. Sobald aber die Nutzerzahl steigt, steigt auch die Ladezeit der Videos. Die Zeit, die gewartet werden muss, bis das Video startet, ist bei einer spezifischen Anzahl von Nutzer noch „ertragbar“. Ertragbar meint hierbei eine Antwortzeit des Servers unter 1 Sekunde. Steigt aber die Nutzeranzahl über diesen spezifischen Wert, erhöht sich die Antwortzeit des Servers. Besonders der Nutzer wird dadurch eingeschränkt, da dieser auf Videos warten muss, bis diese abgespielt werden können. Das Warten auf Videos kann unter Umständen weitreichende Folgen haben, die sich auf das Geschäft auswirken, deswegen wird versucht die Performanz zu verbessern. Zu diesen Zweck sollte durch das Testen herausgefunden werden, ab welcher Benutzerzahl dieses Performanzproblem auftritt.

5.2 Versuchsaufführung

Um die Belastbarkeit des Systems zu testen wurde das System mit einem Stresstest und einem Rampup mit 5 Minuten Zeitspanne benutzt. Wie die Nutzer umgesetzt wurden, wird in 4.1 erläutert. Nach allen 5 Minuten wurde die Anzahl der Nutzer verdoppelt. Somit ergab sich eine Nutzerzahl von 10, 20, 40, 80 ,160, 320 und 640 parallelen arbeitenden Lasteinheiten. Insgesamt dauerte der Test 35 Minuten und zeigte den Fehler des Speicherausnutzung.

[4] Beschreibt zusätzliche Informationen die mit wichtigen Informationen gemischt werden[21]

Das Bild 5.2.1 zeigt grafisch, wie sich die genutzte Bandbreite von 10 Nutzer auf 20 Nutzer fast linear verdoppelt, sowie die Bandbreite von 20 auf 40, 40 auf 80 und 80 auf 160 Nutzer. Das Verdoppeln der Bandbreite ist in diesen Fall ok, um den Nutzer den Inhalt der Webapplikation als fließende Videoerfahrung zu gewährleisten. Wenn das System skaliert und die Nutzeranzahl steigt, ist eine Auslastung der Rechenknoten/VMs normal. In diesen Fall ist die Verdoppelung der Bandbreite noch im Rahmen des ertragbaren, da generell die Last steigt und die Bandbreite hoch genug ist, um flüssiges Streamen von Inhalten zu gewährleisten. Von 160 Nutzer auf 320 Nutzer wird das lineare Steigen der Bandbreite gebrochen und keine Verbesserung der Bandbreite erzielt, was in Anbetracht der letzten Aufrufe negativ ist. Dieses Verhalten ist nicht gut, da das System die Bandbreite nicht ausnutzt und nur eine minimale Verbesserung erzielte. Bei den Sprung von 320 Nutzer auf 640 Nutzer kommt es zur Verringerung der Bandbreite, was schlecht ist für die Performanz. Bei dem letzten Schritt stagniert die Performanz was in 5.2.2 noch besser betrachtet werden kann.

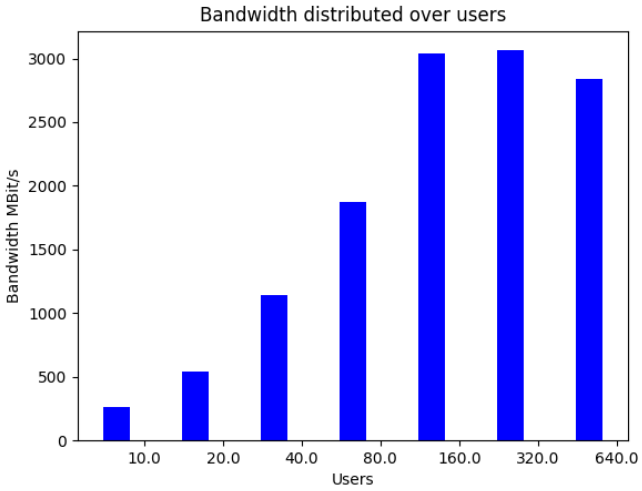

Abbildung 5.2.1: Bandbreitenutzung

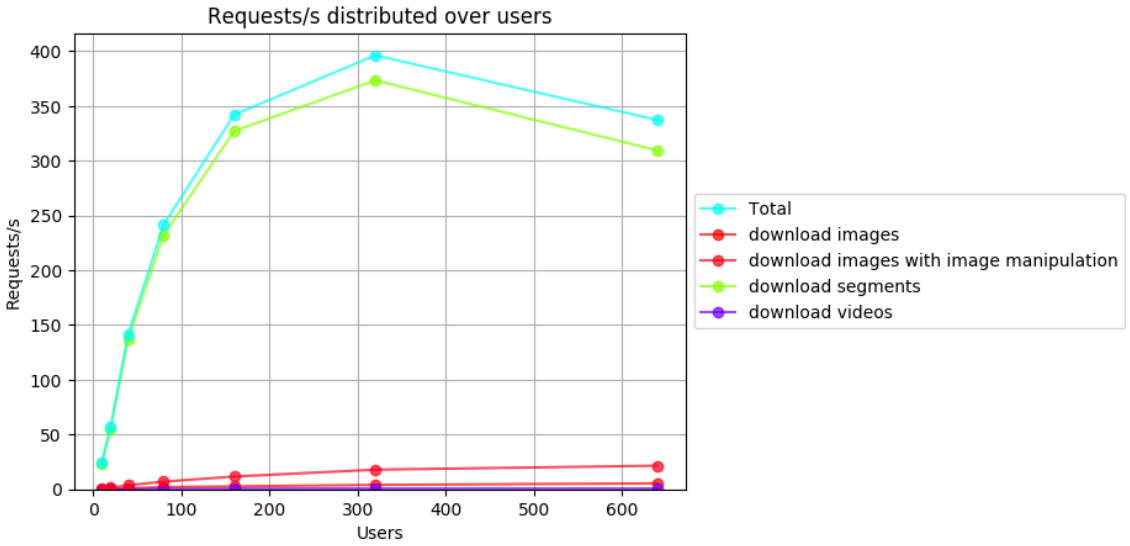

Abbildung 5.2.2: Anzahl der Requests/s

Die Anzahl der Requests/s zeigt den Performanzeinbruch noch deutlicher, wie in Bild 5.2.2 gut zwischen den Schritt von 320 Nutzer zu 640 Nutzer betrachtet werden kann. Die Anzahl an Requests/s der vorherigen Schritte verlaufen größenteils linear wie von 10 auf 20 Nutzer ersichtlich ist. Hierbei verdoppelt sich die Anzahl der Requests/s und von 20 auf 40 Nutzer wird sogar das 2.5 fache an Requests/s erreicht. Der Schritt von 40 auf 80 Nutzer verhält sich im Gegensatz zu den vorherigen Sprung nicht skalierend, da hier nur ca 1.7 fache an Requests/s erreicht wird. Der nächste Schritt von 80 auf 160 Nutzer zeigt eine noch schlechtere Skalierung, da sich die Anzahl an Requests/s um 1.5 fache erhöht. Der vorletzte Schritt zeigt eine schlechte Skalierung der Anzahl an Requests/s um das 1.1 fache. Der letzte Schritt von 320 auf 640 Nutzer zeigt ein sublineare Skalierung, wo das System mit zunehmender Anzahl an Nutzer ein Stagnierung an Requests/s erfährt.

Mit der Unterstützung von Grafana, wurde die serverseitige Auslastung während des Testes mitanalysiert, wie in Bild 5.2.4 und Bild 5.2.3 ersichtlich. Grafana wird benutzt, um die Last auf den Server zu kontrollieren und die CPU-, Speicher/Festplatte-, RAM-Speicher-Auslastung grafisch darzustellen.[22]

In Bild 5.2.3 sieht man den Verlauf der Speicherzugriffe während des Performanztestes. Zwischen den Zeitraum von 13:24 und 13:25 kam es zu einer Stagnation der Speicherzugriffe, wie auch zwischen 13:34 und 13:36. Dies hat keine relevanten Einflussfaktoren auf den Performanztest. Wird aber die CPU Auslastung der Delivery Agents betrachtet, wie in Bild 5.2.4, wird die CPU Auslastung immer größer und fällt in den kritischen Bereich (orange makiert) hinein.

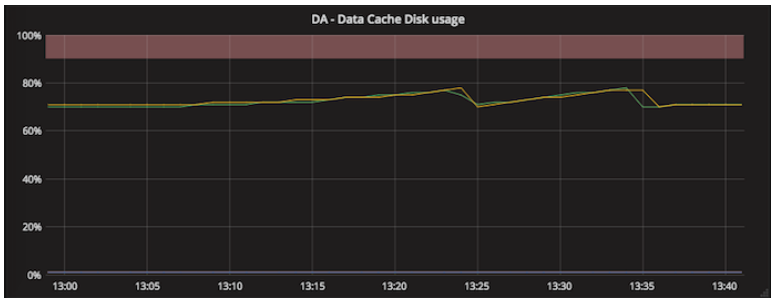

Abbildung 5.2.3: Speicherzugriffe

Hierbei liegt der Fehler, der durch ein Performanztest entdeckt wurde. Der genaue Grund, warum die Anzahl der Requests/s stagniert wurde noch nicht herausgefunden, aber der Tests zeigt ein Bottleneck bei der Speichernutzung, da hierbei eine Stagnation erkennbar ist und sich trotzdem die CPU-Auslastung erhöht.

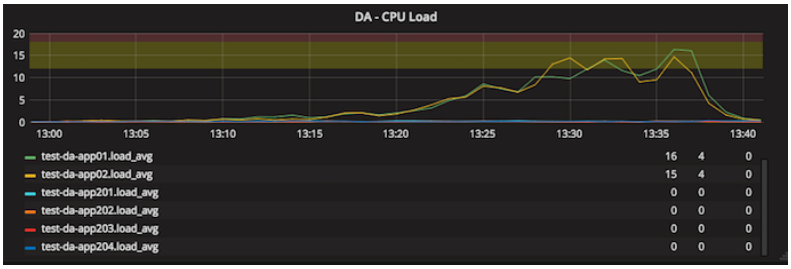

Abbildung 5.2.4: CPU Auslastung

6 Zusammenfassung

Jegliches Testen von Software unterstützt die Qualität und somit auch die Kundenzufriedenheit. Dabei gibt es aber verschiedene Herangehensweisen. Sei es das späte Testen, der agile Ansatz oder ein hybrider Mix aus Konversation und Basiswissen über agile Ansätze. Alles sorgt für eine Qualitätssicherung der Software. Dabei muss für jedes Team individuell entschieden werden, welcher Ansatz gewählt und gefolgt wird. Für meinen Teil ist letztendlich der hybride Ansatz der effektivste Ansatz für jede Situation.

In dieser Arbeit wurden zwei Punkte ausgearbeitet. Der erste Punkt befasste sich mit den Schwachstellen des Testansatzes vom Wasserfall-Modell (späte Testen) und wie diese mit dem agilen Ansatz ausgebessert wurden. Der andere Punkt befasste sich vor Allem mit dem neuen, aufgrund von Recherchen und Eigeninitiative evaluierten Ansatz, der durch den Einsatz von sozialen Komponenten die Erstellung von Performanztests vereinfacht und beschleunigt.

Zur Evaluierung wurde ein Performanztest mit simulierten Nutzern durchgeführt und dabei die Google Search Statistik als Basis für die Entscheidungen der Nutzer benutzt. Der Test wies ein Bottleneck im Testsystem der Firma auf, welches daraufhin behoben wurde.

Im Gegensatz zu den späten Testen und auch dem agilen Ansatz wurde durch Reden über die Art und Weise wie der Test den Entwicklungsprozess unterstützen soll, der Performanztest gestaltet. Somit konnte die Qualität gesichert werden und der Entwicklungsprozess konnte sich neueren Problemen widmen.

Literaturverzeichnis

[1] L. C. J. Gergory, *Agile Testing A PRACTICAL GUIDE FOR TESTERS AND AGILE TEAMS*. Addison Wesley, 2008.

[2] J. Bach, *Heuristic Risk-Based Testing*. James Bach, 1999.

[3] J. Weiss and E. Tan, "Klassische vs. agile methoden der softwareentwicklung," 2004.

[4] J.-P. Riehle, "Klassiker des projektmanagements: Wasserfall-modell vs. agiles vorgehen." https://www.pinuts.de/blog/webstrategie/projektmanagement-wasserfall-gegen-scrum.

[5] U. Mayer-Schwalb, "Qualität ist wenn eins ins andere greift – das test service center." http://www.anecon.com/blog/qualitaet-ist-wenn-eins-ins-andere-greift/.

[6] I. E. Certification, "What are test pyramid and testing quadrants in agile testing methodology?." http://istqbexamcertification.com/what-are-test-pyramid-and-testing-quadrants-in-agile-testing-methodology/.

[7] A. Müller, "Unit-tests (komponententests)." http://www.evocomp.de/softwareentwicklung/unit-tests/unittests.html.

[8] L. Koskela, "Acceptance tdd explained."

[9] M. Treder, "Wireframing, prototyping, mockuping – what's the difference?." https://designmodo.com/wireframing-prototyping-mockuping/.

[10] I. Molyneaux, "The art of performace testing," *O'Reilly*, vol. Edition 1, no. ISBN 978-0-596-52066-3, Januar 2009.

[11] S. Gabler, "Gabler wirtschaftslexikon."

[12] Dualstudent, "Definition: Webservice." http://www.derdualstudent.de/definition-webservice.html.

[13] B. P. James Bach; Cem Kaner, *Lessons Learned in Software Testing.* John Wiley, 2002.

[14] http://www.itwissen.info/Framework-framework.html.

[15] J. Allen, "How much is a google top spot worth?." https://searchenginewatch.com/sew/study/2050861/how-much-google-top-spot-worth.

[16] http://locust.io/.

[17] J. B. M. Bolton, "The new agile testing quadrants, bringing skilled testers and developers together." https://www.dropbox.com/s/1hnsf4kafa6nri2/new2002.

[18] VMWare, *VMware Infrastructure Architecture Overview.* VMWARE, 2006.

[19] http://www.itwissen.info/definition/lexikon/cache-Cache.html, 11 2016.

[20] I. Ros, "Rest api documentattion best practices." https://bocoup.com/blog/documenting-your-api.

[21] WhatIs.com, "metadata." http://whatis.techtarget.com/definition/metadata.

[22] G. Labs, "The open platform for beautiful analytics and monitoring." https://grafana.com/.

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Passau, den 28. Juni 2017

Kevin Glöckl