universität
wien

BACHELORARBEIT

Titel der Bachelorarbeit

Testen verteilter Microservices im Rahmen der neu gestalteten Open Models Laboratory Anwendungssystemarchitektur

Verfasser

Christian Kersjes

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, im Oktober 2015

Studienkennzahl lt. Studienblatt: A 033 521
Studienrichtung lt. Studienblatt: Medieninformatik
Betreuer: Univ.-Prof. Mag. Dr. Hans-Georg Fill, Privatdoz.
Mitwirkung: Dipl.-Wirtsch.Inf. Univ. Dr. Dominik Bork

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Admin Administrator ... 28
API Application Programming Interface ... 22
CAS Central Authentication Service ... 25
CAS TGT CAS TicketGrantingTickets ... 32
CPU Central Processing Unit ... 2
DoS Denial of Service ... 14
EJB Enterprise JavaBeans ... 21
FTP File Transfer Protocol ... 23
GB Gigabyte ... 25
GHz Gigahertz ... 25
GUI Graphical User Interface ... 20
HTTP Hypertext Transfer Protocol ... 23
HTTPS Secure Hypertext Transfer Protocol ... 23
ID Identifier ... 32
IoC Inversion of Control ... 21
ISO Internationale Organisation für Normung ... 6
ISTQB International Software Testing Qualification Board ... 6
J2EE Java Platform Enterprise Edition ... 21
JDBC Java Database Connectivity ... 23
JVM Java Virtuell Machine ... 27
LDAP Lightweight Directory Access Protocol ... 22
LT LoginToken ... 32
MTBF Mean Time Between Failure ... 16
MTTR Mean Time To Repair ... 16
OMiLAB Open Models Laboratory ... 23
O/R-Mapper Object/Relational Mapper ... 21
POJO Plain Old Java Object ... 21
PSM Projektstrukturmanager ... 25
RAM Random-Access Memory ... 25
REST Representational State Transfer ... 20
SOAP Simple Object Access Protocol ... 23
SSH Secure Shell ... 26
TCP Transmission Control Protocol ... 27
UDP User Datagram Protocol ... 27
URI Uniform Resource Identifier ... 13
URL Uniform Resource Locator ... 25
VM Virtuelle Machine ... 25
WAR Web application ARchive ... 19
XML Extensible Markup Language ... 22

1 Einleitung

Software ist heutzutage ein wichtiger, nicht mehr wegzudenkender, Teil unserer Gesellschaft [1]. Dabei hat sie in jeden erdenklichen Bereich Einzug gehalten. Dazu zählen auch aller Art von kritischen Bereichen, in denen es sehr wichtig ist, dass die eingesetzte Software höchsten Qualitätsansprüchen genügt. Aber auch in nicht kritischen Bereichen wird immer mehr Wert auf Qualität gelegt, was nicht zuletzt dem gesteigerten Qualitätsverlangen der (End-)Kunden geschuldet ist.

Dies, in Verbindung mit immer umfangreicherer und komplexerer Software, führt dazu, dass es für immer mehr Unternehmen unumgänglich ist, ihre Anwendungen umfangreich zu testen, um ein gewisses Maß an Softwarequalität bieten zu können. Zusätzlich wirkt sich das Testen auch auf die Qualität des Entwicklungsprozesses aus, da Fehler schon frühzeitig gefunden werden [2].

Diese Arbeit geht auf die verschiedenen Aspekte des Softwaretests ein, wobei die Hauptausrichtung der Arbeit das Testen verteilter Microservice Anwendungen ist. Nach dem konzeptuellen Teil werden die beschriebenen Konzepte in der Fallstudie angewendet. Der Themenschwerpunkt dieser Fallstudie ist der Lasttest von Microservices. Vor allem dieser praktische Teil ist eine große Motivation für diese Bachelorarbeit, da es kaum verwandte Arbeiten zu diesem Thema gibt.

Die nötigen theoretischen Grundlagen zu der Fallstudie werden im zweiten Kapitel näher beschrieben. Dabei werden unter anderem die theoretischen Grundlagen zu den einzelnen Testsichten, Testarten und zur Testspezifikation beleuchtet. Im dritten Kapitel wird auf die spezifischen Einzelheiten der Fallstudie eingegangen. Es wird beschrieben, was Microservices und das Spring Framework sind. Außerdem wird detailliert dargestellt, was Lasttests sind und welche Werkzeuge dabei zur Verfügung stehen. Im vierten Kapitel wird die praktische Umsetzung der Fallstudie schriftlich erläutert, indem die Zielsetzung, Vorbedingungen, Testsettings, Testbeschreibung und Ergebnisse beschrieben werden. Dies führt im fünften Kapitel zur Related Work und endet im sechsten Kapitel mit der Conclusio, welche die Bachelorarbeit abschließt.

2 Grundlagen des Softwaretest

Beim Testen werden definierte Soll-Zustände mit gemessenen Ist-Zuständen verglichen. Dies unterscheidet den Test von einem Experiment. Dem Test liegen erwartete Ergebnisse zugrunde, während das Experiment ergebnisoffen ist.

Heutzutage muss Software höchsten Anforderungen genügen, da die zugrunde liegende Architekturen immer komplexer werden, die Qualitätsansprüche der Kunden wachsen und der Geschäftserfolg immer mehr von der eingesetzen Software abhängig ist. Hinzu kommt, dass der Zeitdruck zur Umsetzung immer mehr steigt. Softwaretests können das Vertrauen in die Anwendung erhöhen und Fehler aufdecken.

Mit Hilfe moderner Softwaretests können Fehler (Abweichung zwischen Soll-Wert und Ist-Wert) zu jedem Zyklus der Softwareentwicklung erkannt und beseitigt werden. Dazu sollten die verschiedenen Tests parallel zur laufenden Entwicklung erfolgen, damit Fehler frühest möglich erkannt werden und nicht in die kommenden Zyklen übernommen werden.

Es ist in Unternehmen oft so, dass es kein Test-Management gibt [3] und die verantwortlichen Projektleiter erst nachträglich, stichprobenartig, einzelne Funktionen der Software testen. Das Resultat sind meistens viele Fehler, welche sich nicht in der geplanten Projektzeit beheben lassen. Das führt zu Verzögerungen der Softwareeinführung und zu einem Mehraufwand durch Fehlerkorrekturen, was wiederum zu erhöhten Kosten des Projektes führt. Um das Projekt nicht noch weiter zu verzögern, wird meist auf die Durchführung entsprechender Regressionstests verzichtet. Dies birgt das Risiko, dass es Fehler in der Software gibt, welche bisher unbemerkt geblieben oder neu hinzugekommen sind.

Um dies alles zu vermeiden, ist es wichtig, ein systematisches Test-Management zu etablieren. Dieses Test-Management integriert für jede Entwicklungsstufe eine dementsprechende Teststufe. Um dies konsistent umsetzen zu können, muss die Testimplementierung über eine Testfallbeschreibung hinaus gehen und zusätzlich die Definition eines Datensets beinhalten. Damit sind überlegte Testdaten gemeint. Dieser Datenbestand sollte Daten beinhalten, von denen man weiß, zu welchen Ergebnissen sie führen. Diese standartisierten Daten müssen laufend den aktuellen Gegebenheiten angepasst werden.

Das übergeordnete Ziel eines Softwaretests ist das Messen der Softwarequalität anhand definierter Anforderungen.

Um ein systematisches, standardisiertes Test-Management umzusetzen, wird die Qualität von Software in mehreren Normen festgelegt, welche unter der Internationale Organisation für Normung (ISO) Norm ISO/IEC 25000 zusammengefasst wurden (für die dementsprechende Testerqualifizierung wurde das International Software Testing Qualification Board (ISTQB) initiiert, welches Zertifizierungen für Tester anbietet.)

Diese ISO Norm definiert Softwarequalität wie folgt:

"Software-Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen." [4, p. 20]

und unterscheidet die Qualität anhand von sechs Merkmalen:

- **Funktionalität**
  Besitzt die Software die geforderten Funktionen?
- **Zuverlässigkeit**
  Behält die Software ihr Leistungsniveau unter bestimmten Bedingungen bei?
- **Benutzbarkeit**
  Wie groß ist der Aufwand des Users zur Nutzung der Software?
- **Effizienz**
  Wie ist das Verhältnis Leistungsniveau/eingesetzte Betriebsmittel?
- **Wartbarkeit**
  Wie hoch ist der Aufwand für Änderungen an der Software?
- **Übertragbarkeit**
  Wie leicht kann die Software portiert werden?

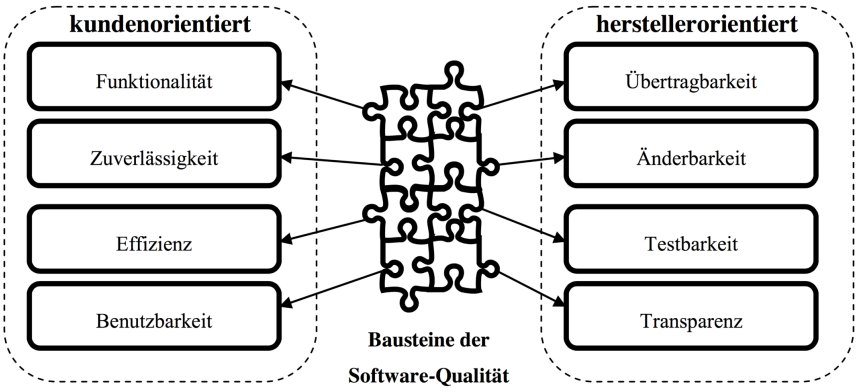

Abbildung 1: Qualitätsmerkmale eines Software-Produkts [5, p. 7]

Ein Teilmerkmal, welches alle sechs Qualitätsmerkmale ausmacht, ist die Konformität. Es beschreibt die Übereinstimmung/Erfüllung der Software mit dem jeweiligen Qualitätsmerkmal und wie weit die Normen erfüllt werden.

Allerdings hat der Softwaretest auch seine Grenzen. So kann er nur nachweisen, dass noch Fehler vorhanden sind, nicht aber, dass keine Fehler mehr vorhanden sind. Um dies nachweisen zu können, müssten alle möglichen Funktionen mit allen möglichen Eingabewerten getestet werden. Dies ist, außer bei sehr einfachen Gegebenheiten, aufgrund der herrschenden Komplexität nicht durchführbar. Deshalb sollten Strategien überlegt werden, wie mit einer kleinen Testanzahl eine möglichst große Testabdeckung realisiert werden kann.

Wenn man davon ausgeht, dass nicht alles getestet werden kann, stellt sich die Frage, wann das Testen beendet werden kann. Dazu werden ökonomische oder statistische Testendkriterien bestimmt. Ökonomische Testendkriterien können zum Beispiel die Finanzen oder die Zeit sein. In diesem Fall endet der Test bei Erreichen einer vorbestimmten Zeitmarke oder wenn die finanziellen Mittel des Projektes erschöpft sind.

Als statistisches Testendkriterium wird meist die Überdeckung verwendet. Dabei wird die Überdeckung unterschiedlich definiert. Zum einen kann das Testende erreicht werden, wenn jede Programm-Funktion einmal getestet wurde. Zum anderen kann sich die Überdeckung auch auf den Softwarecode an sich beziehen. Das würde bedeuten, dass das Testende erreicht ist, wenn beispielsweise 90 Prozent vom Code getestet wurden.

2.1 Testsichten

Es gibt beim Softwaretest zwei unterschiedliche Sichten auf, beziehungsweise in das zugrunde liegende System, also dem Testobjekt. Diese Testsichten werden durch die Box-Tests definiert.

"Blackbox-Tests, Whitebox-Tests und Greybox-Tests beschreiben unterschiedliche „Erleuchtungsgrade“ eines Testobjekts zur Testdurchführung, d. h. die interne Programmstruktur wird beim Test nicht (schwarz), ein wenig (grau) oder vollständig (weiß) ausgeleuchtet."[4, p. 36]

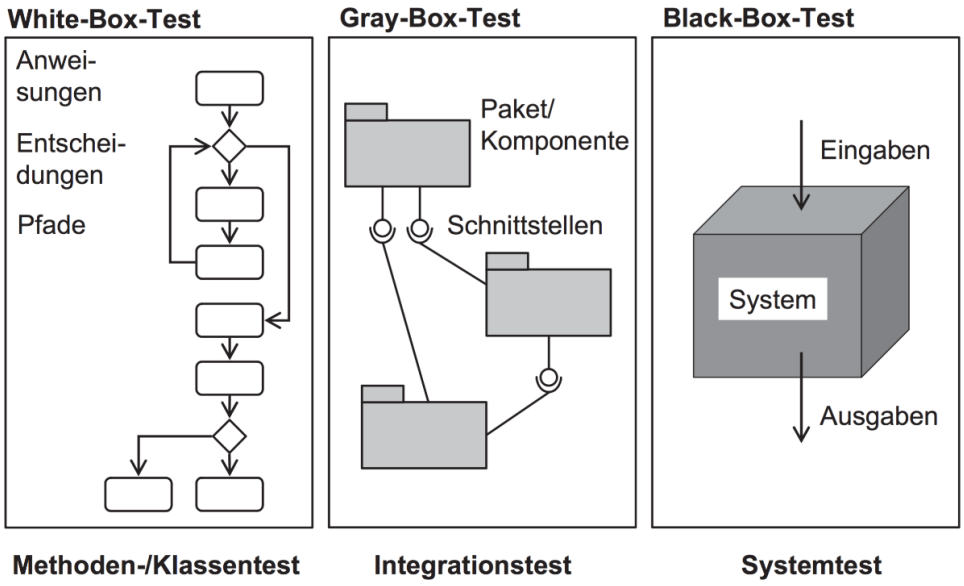

Abbildung 2: Unterschiedliche Sichten auf das Testobjekt [6, p. 29]

Überträgt man die differenzierten Sichten von Abbildung 2 auf das in der Software Entwicklung häufig genutzte V-Modell, ergeben sich die Testebenen wie in Abbildung 3 dargestellt. Diese Testebenen zeigen auch die zeitliche Positionierung innerhalb des Entwicklungsprozesses auf.

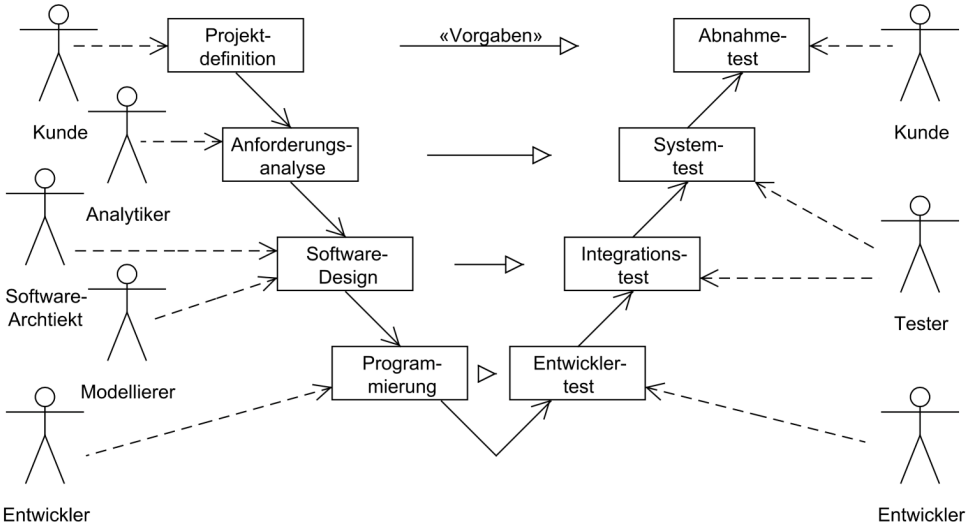

Abbildung 3: V-Modell mit Entwicklung und Test [6, p. 27]

In Abbildung 3 ist der "Entwicklertest" synonym zu dem Komponententest aus der Abbildung 4 zu sehen. Der Komponententest besteht aus den Methoden- und Klassentests, welche unter die White-Box Tests fallen (siehe Abbildung 2). Außerdem zeigt die Abbildung 3 indirekt auch die Stufen der Validierung und Verifizierung auf, wie es in der Abbildung 4 zu sehen ist.

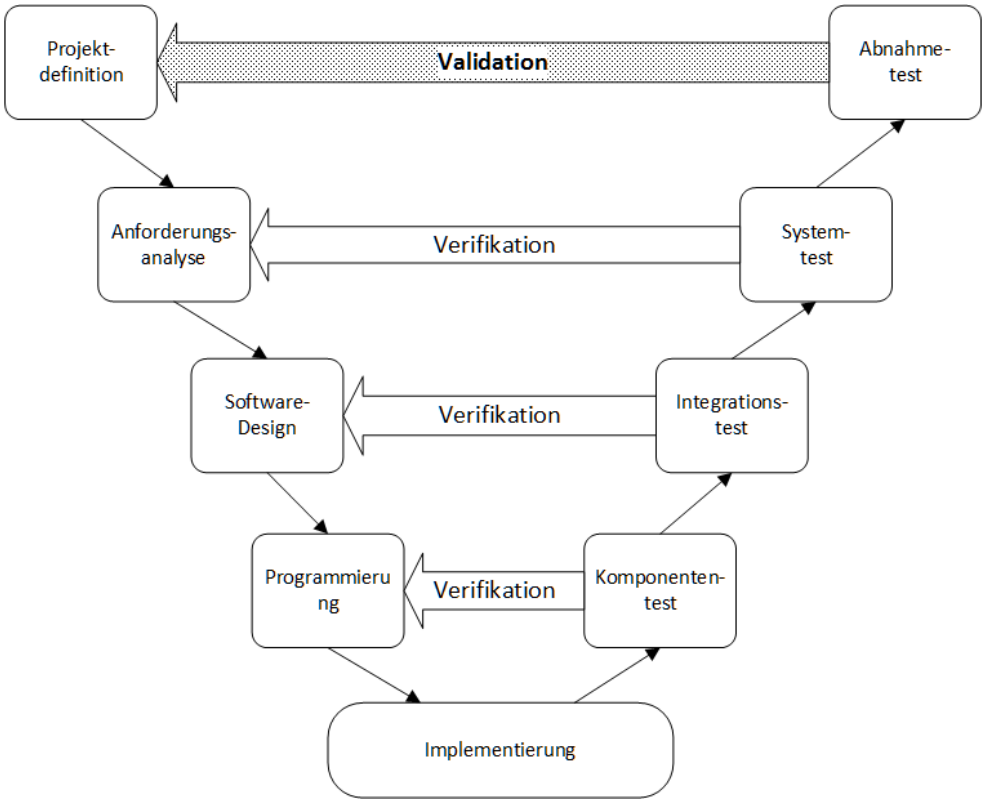

Abbildung 4: V-Modell Verifikation und Validierung

Beim Softwaretest wird zwischen Validierung und Verifizierung unterschieden. Die gesamte Softwareentwicklung ist so aufgebaut, dass aufgrund von Anforderungen Spezifikationen abgeleitet werden und auf Grund dieser Spezifikationen findet eine Implementierung statt. Wird nun gegen eine Spezifikation getestet, liegt eine Verifizierung vor (es wird, ohne Programmablauf, mittels formaler Methoden, die Korrektheit bezüglich der Spezifikation bewiesen). Testet man gegen eine Anforderung (siehe Abbildung 4), nennt man es Validierung (es wird geprüft, ob die Anforderungen stimmig sind). Das ist nicht mit dem Punkt "Anforderungsanalyse" zu verwechseln, welcher namentlich eher an eine Validierung denken läßt. Die Anforderungsanalyse untersucht die Anforderungen, welche in der "Projektdefinition" festgelegt wurden, um ein grundlegendes Verständnis des Projekts zu schaffen. Als Ergebnis wird dabei eine Anforderungsspezifikation geliefert, gegen die getestet wird. Wie oben beschrieben, bedeutet das Testen gegen eine Spezifikation, dass eine Verifizierung vorliegt.

2.1.1 Blackbox-Test

Beim Blackbox-Test liegen keine Informationen über die innere Struktur des Testobjektes vor. Hierbei wird das System gegen seine Spezifikationen getestet, wobei die Information, wie es im Inneren des Systems abläuft, nicht relevant ist, sondern es kommt nur auf das Ergebnis an (Analyse der Eingaben und Ausgaben der zu testenden Anforderung). Es wird die korrekte Umsetzung der Spezifikation bezüglich Fehlerfreiheit und Vollständigkeit überprüft.

Als Grundlage zur Umsetzung von Blackbox-Tests dienen die Blackbox-Verfahren. Diese Blackbox-Verfahren werden oft mit den Blackbox-Tests gleichgesetzt, meinen aber eine unterschiedliche Ebene innerhalb der Blackbox-Tests. Die Blackbox-Verfahren sind Methoden um Testfälle zu definieren, welche dann im Blackbox-Test ablaufen.

Wie schon erklärt, ist es nicht möglich, Software zu 100 Prozent zu testen. Um ein sinnvolles und hinreichendes Maß zu erreichen ist es notwendig, die Anforderungsspezifikationen zu analysieren und daraus Testfälle abzuleiten, welche eine hinreichende Testabdeckung bieten. Dazu gibt es mehrere Verfahren: die Äquivalenzklassen-Analyse, die Grenzwert-Analyse, die Ursache-Wirkungs-Analyse und das anwendungsfallbasierte Testen.

Da es nicht möglich ist, dass alle Varianten eines Testfalls getestet werden, wird versucht, anhand der Bedingungen aus der Spezifikation die Eingabewerte mit identischen Verhalten in äquivalenten Klassen aufzuteilen: den sogenannten Äquivalenzklassen.

Bedingung	zulässige Äquivalenzklasse	Nr.	unzulässige Äquivalenzklasse	Nr.
Anzahlung	Ganze Zahl 2.000 bis 10.000	Z1	Keine Eingabe Kleiner als 2.000 Größer als 10.000 Nicht ganze Zahl Nicht numerische Eingabe	U1 U2 U3 U4 U5

Tabelle 1: Äquivalenzklassen [4, p. 40]

Im Prinzip beinhalten die Äquivalenzklassen unterschiedliche Werte, welche aber vom Testobjekt gleich verarbeitet werden (siehe Tabelle 1). Diese Unterteilung in Äquivalenzklassen nennt man Äquivalenzklassen-Analyse.

Aufbauend auf die Äquivalenzklassen wird nun die Grenzwert-Analyse durchgeführt, um Fehler in den kritischen Grenzbereichen zu finden. Mit Grenzwert ist der Grenzbereich der Äquivalenzklasse gemeint. Bei Tabelle 1 Äquivalenzklasse Z1 zum Beispiel der Wert 2.000 und 10.000.

Bedingung	Grenzwert zulässige Äquivalenzklasse	Nr.	Grenzwert unzulässige Äquivalenzklasse	Nr.
Anzahlung	2.000 10.000	Z1-GW1 Z1-GW2	0 1.999 10.001 [Größte eingebbare Zahl]	U2-GW1 U2-GW2 U3-GW1 U3-GW2

Tabelle 2: Grenzwerte zu Äquivalenzklassen [4, p. 43]

Bei der Grenzwertanalyse werden die sogenannten beidseitigen Grenzwerte genommen. Damit sind die Grenzwerte plus minus eins gemeint. In dem Fall Z1: 1.999, 10.001 (siehe Tabelle 2).

Die Ursache-Wirkungs-Analyse untersucht die Kombinationen von Äquivalenzklassen, welche verschiedene Ergebnisse liefern, auf ein korrektes Zusammenspiel hin. Es werden die Abhängigkeiten von Anforderungsbedingungen analysiert. Dabei gilt auch für die Ursache-Wirkungs-Analyse, dass nicht alle Kombinationsmöglichkeiten getestet werden können und eine möglichst repräsentative Auswahl getroffen werden muss.

Ursache	**Regeln**
Die Äquivalenzklassen	Definieren von Kombinationsmöglichkeiten der Äquivalenzklassen. Abbildung von fachlichen Abhängigkeiten der Klassen untereinander.
Wirkungen	**Entscheidungen**
Ausgabeauflistung der möglichen Programmreaktionen.	Trifft die Wirkung auf die Regel zu?

Tabelle 3: Entscheidungstabelle: Aufbau

Zur praktischen Umsetzung hat sich eine Entscheidungstabelle etabliert, welche die Kriterien: Ursache, Regeln, Wirkungen und Entscheidungen enthält (siehe Tabelle 3). Ein praktisches Beispiel könnte wie folgt ausschauen:

Ursache			**Regeln**			
			1	**2**	**3**	**4**
Leasing	Ja	Z1	X	X		
	Nein	Z2			X	X
Kaufpreis	<=20.000	Z3	X		X	
	> 20.000	Z4		X		X
Wirkungen			**Entscheidungen**			
Preisnachlass 500€		W1	N	N	N	J
Übernahme Versicherung		W2	N	J	N	N

Tabelle 4: Entscheidungstabelle: Praktische Umsetzung

In dem Beispiel aus Tabelle 4 geht es darum, dass gewisse Anforderungen erfüllt sein müssen, um eine bestimmte Wirkung zu erzielen. Zum einen ist es eine Kombination aus dem Faktor "Das Auto ist geleast" und der "Kaufpreis des Autos liegt über 20.000 €", dann "Übernahme der Versicherung". Wenn also Z1+Z4 erfüllt sind, dann tritt W2 ein. Zum anderen ist die Kombination der Äquivalenzklassen "Das Auto ist nicht geleast" und der "Kaufpreis des Autos liegt über 20.000 €", dann wirkt "Preisnachlass 500 €" (W1 = Z2+Z4)

Das Verfahren, welches gerade in der objektorientierten Softwareentwicklung relevant ist, ist das anwendungsfallbasierte Testen. Diesem Verfahren liegt das im Zuge der Softwareentwicklung erstellte Use Case Diagramm zugrunde.

Das Use Case Diagramm ist ein Verhaltensdiagramm und beschreibt die Abhängigkeiten und Beziehungen zwischen Anwendungsfällen und Akteuren. Dabei wird das erwartete Systemverhalten dargestellt. Ein Akteur stößt ein Use Case an, was zu einem eindeutigen Ergebnis für den Akteur führt. Dabei ist es meistens so, dass ein Akteur zwar einen Anwendungsfall anstößt, aber dieser Anwendungsfall wiederum mit anderen Anwendungsfällen in Beziehung steht. Dies bildet den zugrunde liegenden Geschäftsprozess ab. Alle diese einzelnen Fragmente wurden in der Anforderungsbeschreibung spezifiziert.

Das anwendungsfallbasierte Testen greift mittels des Anwendungsfall Diagramms auf die spezifizierte Anforderungsbeschreibung zu und überprüft dabei die vollständige und korrekte Umsetzung, indem es anhand der Use Cases passende Testszenarien entwirft.

2.1.2 Whitebox-Test

Der Whitebox-Test ist im Gegensatz zum Blackbox-Test, ein Strukturtest. Das bedeutet, dass die innere Struktur (der Quellcode) des Testobjektes auf Vollständigkeit und Korrektheit hin analysiert wird. "Whitebox-Tests sagen etwas über die Art der Testdurchführung aus und können in jeder Teststufe eingesetzt werden." [4, p. 35]

Synonym zum Blackbox-Test - Blackbox-Verfahren dienen hier die Whitebox-Verfahren als Grundlage zur Umsetzung der Whitebox-Test. Mittels der Whitebox-Verfahren leiten sich die Testfälle aus den Applikationsstrukturen ab. Es gibt zwei zentrale Qualitätsfragen, welche mittels dem Instrument Testüberdeckungsgrad (Code Coverage Test) beantwortet werden:

- Gibt es Programmteile, die beim Test nicht durchlaufen wurden?
- Gibt es einen Programmcode, der nie durchlaufen werden kann?
  [4, p. 73]

Der Testüberdeckungsgrad sagt aus, wieviel Prozent des Codes mittels des absolvierten Tests durchlaufen wurden. Wenn dabei herauskommt, dass gewisse Teile des Codes nicht durchlaufen wurden, kann es zwei mögliche Ursachen geben. Zum einen kann es daran liegen, dass es nicht genügend Testfälle gibt und zum anderen, dass die Codepassagen nicht erreicht werden können, da der Programmcode falsch ist.

Praktisch relevant sind vier Arten der Testüberdeckungsgrade, welche jeweils eine unterschiedliche Aussagekraft haben. Das sind die Anweisungsüberdeckung (Statement Coverage), Entscheidungsüberdeckung (Decision Coverage), Bedingungsüberdeckung (Condition Coverage) und die Zweigüberdeckung (Branch Coverage).

- **Anweisungsüberdeckung**
  Werden alle Anweisungen mittels der Testfälle ausgeführt?

- **Entscheidungsüberdeckung**
  Wurde jedes mögliche Entscheidungsergebnis (wahr, falsch) durch die Testfälle herbeigeführt?

- **Bedingungsüberdeckung**
  Basiert eine Entscheidung auf mehrere kleineren Teilbedingungen? Wenn ja, werden diese durch die Bedingungsüberdeckung analysiert. Teilbedingungen können zum Beispiel if-Statements sein (if... or... then...)

- **Zweigüberdeckung**
  Wurde jeder Zweig der Applikation durch die Testfälle ausgeführt?
  100 Prozent Zweigüberdeckung = 100 Prozent Entscheidungsüberdeckung = 100 Prozent Anweisungsüberdeckung

Das Bestimmen der Testüberdeckungsgrade kann schon bei relativ kleiner Software sehr komplex und unübersichtlich werden, weshalb es praktisch nur mittels Tools (in Java zum Beispiel Clover) möglich ist.

2.1.3 Greybox-Test

Der Greybox-Test ist eine Art von Verbindungspunkt zwischen dem Whitebox-Test und Blackbox-Test. Er verknüpft die zwei unterschiedlichen Sichtweisen auf das Testobjekt. Anders als bei den zwei anderen Box-Tests, stehen dem Greybox-Test dazu keine eigenen Greybox-Verfahren zur Verfügung.

Beim Greybox-Test wird die Greybox überprüft. Bei der Greybox handelt es ich um eine Komponente, welche über Schnittstellen nach außen kommuniziert (Blackbox-Test Anteil), wobei die innere Struktur

trotzdem analysiert werden kann (Whitebox-Test Anteil). Es wird getestet, ob die Anforderungen richtig umgesetzt wurden und ob dabei der Ablauf und Aufbau der inneren Strukturen korrekt vorliegt. Dies trifft zum Beispiel häufig auf Integrationstests zu.

2.2 Testarten

Bei jeder Art zu testen geht es darum, bestimmte Qualitätsmerkmale zu evaluieren, wobei die wesentlichen Merkmale in einer Applikation Funktionalität, Effizienz und Zuverlässigkeit sind. Dieses Kapitel zeigt auf, welche Möglichkeiten es zum Testen dieser Merkmale gibt.

2.2.1 Funktionalität

Unter dem Qualitätsmerkmal "Funktionalität" werden die Testarten zusammengefasst, welche sicherstellen, dass die Funktionen im vollen Umfang, wie gefordert, umgesetzt wurden und die einzelnen Komponenten fehlerfrei und funktionierend als System zusammen arbeiten.

Hier sind vor allem der Unit-Test, Integrationstest, Systemtest, Webservice-Test und Sicherheitstest zu nennen.

- **Unit-Test**
  "Eine Software verfügt in der Regel über einzelne Komponenten, so genannte Units oder Module. Durch einen Unit Test wird eine einzelne Unit auf ihre Funktionalität getestet. Das Testen einzelner Module erlaubt eine einfachere Prüfung der programmierten Funktionalitäten. Hierbei werden keine Schnittstellen und Abhängigkeiten zu anderen Modulen berücksichtigt." [7, p. 66]

  Werkzeuge: JUnit (Java) [8], CUnit (C) [9], CppUnit (C++) [10], JSUnit (JavaScript) [11], NUnit (.Net) [12], PHPUnit (PHP) [13]

- **Webservice-Test**
  Der Fokus dieser Arbeit liegt auf dem (Last-)Testen von Microservices, folglich von verteilten Systemen. Dort spielen Webservices eine große Rolle. Aus Testsicht verhält sich ein Webservice wie eine eigenständige Komponente und fällt damit unter den Komponententest.

  Aufgrund der Webservicebeschreibung werden Testfälle erstellt und ausgeführt. Dies geschieht nicht, wie beim Unit-Test auf Codeebene, sondern durch die eindeutigen Uniform Resource Identifiers (URIs), die als Schnittstelle dienen. Dazu werden sogenannte Testtreiber benötigt. Das sind Testschnittstellen, die bei Komponenten zum Einsatz kommen, welche nicht direkt aufrufbar sind. Der Testtreiber kann diese Komponenten aufrufen und/oder steuern.

  Werkzeuge: SoapUI [14], HP LoadRunner [15], Apache JMeter [16]

- **Integrationstest**
  Der Integrationstest prüft das spezifikationsgemäße Zusammenspiel der, mittels Unit-Test auf Fehlerfreiheit getesteten, Komponenten. Dazu gibt es zwei Vorgehensweisen:

  Bottom-up Vorgehensweise
  Hier geht man schrittweise beim Testen vor und kombiniert zunächst einzelne Komponenten, geht dann weiter zu logisch verknüpften Teilsystemen, hin zum Testen des Gesamtsystems.

  Top-down Vorgehensweise
  Dies ist der genau entgegengesetzte Weg zur bottom-up Vorgehensweise. Dieser Ansatz fängt mit dem Testen des Gesamtsystems an (wenn die einzelnen Komponenten noch nicht implementiert sind, werden Teststubs genutzt, also Komponenten ohne Funktion, welche nur die Eingaben entgegennehmen

und gewünschte Ergebnisse zurückliefern.) und arbeitet sich nach unten vor. Dabei werden die Teststubs nach und nach mit den realen Teilsystemen ersetzt, wodurch man leichter das Auftreten von Problemen auf konkrete Bereiche der Software eingrenzen kann.

Werkzeuge: Meistens bieten die Tools für Unit-Tests auch Werkzeuge für den Integrationstest an.

- **Systemtest**
  Der Systemtest ist die Weiterführung des Integrationstests auf Basis des Gesamtsystems, aber mit der Sicht auf fachliche Geschäftsprozesse. Es werden die in den den Spezifikationen definierten Anwendungs-/Geschäftsfälle getestet. Dieser Test ist die Vorstufe zum Akzeptanztest beim Kunden und findet nicht mehr in der Entwicklungsumgebung statt, sondern in einer, möglichst nah an der Produktivumgebung angelegten, Testumgebung.

  Werkzeuge: Es können auch hier die Tools des Integrationstests genutzt werden. Zusätzlich gibt es noch die Möglichkeit, Capture Replay Tools einzusetzen. Dabei werden alle vom Tester durchgeführten Eingaben (das können Tastatureingaben aber auch Mausklicks sein) aufgezeichnet und gespeichert. Das Capture Replay Tool kann dann diese Testfälle automatisch wiederholen. Darunter fallen die Tools Selenium [17] und CitraTest [18].

- **Sicherheitstest**
  Beim Sicherheitstest geht es um die Überprüfung der "...Eignung, Korrektheit, Unumgänglichkeit und Wirksamkeit der in einem System eingesetzten Sicherheitsvorkehrungen." [4, p. 34] In diesem Zusammenhang ist es nicht ausreichend einen automatischen Security Scan durchzuführen, vielmehr müssen sich die Tester über die Sicherheitsanforderungen und Sicherheitsvorkehrungen, die an das System gestellt werden, im Klaren sein. Dazu zählen nicht nur die Sicherheitsmechanismen der Applikation, sondern auch die systemtechnischen Mechanismen.

  Der Sicherheitstest ist meistens dreigeteilt. Zum einen wird geprüft, ob die bestehenden Sicherheitsvorkehrungen geeignet sind. Dann müssen die Systemschwachstellen identifiziert werden, um danach das System mittels geeigneter Werkzeuge einem Penetrationstest zu unterziehen, um die zuvor bestimmten Sicherheitsschwachstellen zu beweisen oder zu widerlegen. Dazu nutzen die gängigen Werkzeuge Angriffsszenarien, wie zum Beispiel Session-Hijacking, Brute Force Attacken, Denial of Service (DoS) Übergriffe, SQL-Injektion und Man in the Middle Attacken.

  Werkzeuge: OpenVAS (vom deutschen Bundesamt für Sicherheit in der Informationstechnik (BSI)) [19], SecureAssist (Java, PHP, .NET) [20], HP WebInspect [21]

2.2.2 Effizienz

Das Qualitätsmerkmal "Effizienz" umfasst die Testarten, welche überprüfen, dass ein Anwendungssystem nicht nur im Regelbetrieb, sondern auch in Ausnahmesituationen stabil ist und ein annehmbares Zeitverhalten gewährleistet.

Unter den Effizienztests subsummieren sich der Performanztest, Lasttest (in dem Zusammenhang auch der Latenztest) und der Speicherlecktest.

- **Performanztest**
  Nachdem das System fertig gestellt wurde, muss der realistische Betrieb der Anwendung getestet werden. Dabei prüft der Performanztest den Zeit- und Speicherplatzverbrauch des Systems unter typischen Systembedingungen. Typische Systembedingungen sind Bedingungen, welche in den Anforderungen definiert wurden.

Es geht nicht darum, die Grenzen des Systems auszuloten (das wäre der Lasttest), sondern das System unter typischen Bedingungen zu testen. Diese beruhen auf Erfahrungswerten, beispielsweise wieviele User werden im Normalfall das System gleichzeitig nutzen, zu welchen Uhrzeiten, wie lange und wie häufig.

Um diese "Alltagsbedingungen" zu testen, müssen die Geschäftsprozesse als Testszenarien umgesetzt werden. Zur Ermittlung der Testfälle bietet sich das Verfahren des anwendungsfallbasierten Testens an (siehe Kapitel 2.1.1).

Die Metrik, mit welcher häufig die Performanz gemessen wird, ist der Datendurchsatz des Systems. Das ist die Datenmenge, welche in einer gewissen Zeit von der Anwendung verarbeitet werden kann unter den oben genannten Alltagsbedingungen. Diese Metrik kann, je nach Anwendungsart, variieren. Bei einer Webanwendung kann das relevante Kriterium die Antwortzeit sein. Also die Zeit, die ein Nutzer auf das Ergebnis seiner Anfrage warten muss. Dies darf nicht als eine absolute Zahl gesehen werden, sondern als eine Reihe von relativen Zahlen je nach Höhe der Nutzeranzahl, die auf das System gleichzeitig zugreifen.

Werkzeuge: ApacheBench (Webanwendungen) [22], Ants Load (.NET, ASP) [23]

- **Speicherlecktest**
  Der memory leak test überprüft die Anwendung auf Softwarefehler, welche dazu führen, dass Teile des Arbeitsspeichers nicht wieder freigegeben werden. Speicherlecks können sehr starke Folgen für die Leistung des Systems haben. Typische Symptome sind nachlassende Antwortzeiten und Übertragungsraten bis hin zu Abstürzen der Anwendung.

  Speicherlecktest sind Performancetests, die über eine lange Zeit (bis zu einer Woche) durchgeführt werden. Dabei wird darauf geschaut, ob sich die oben genannten typischen Symptome einstellen. Ist dies der Fall, muss die Software noch einmal mittels Unit-Test in Verbindung mit sogenannten Memory-Checker überprüft werden, da der Speicherlecktest nur die Symptome feststellen kann, nicht aber die Ursache.

  Werkzeuge: Die selben Werkzeuge wie beim Performanztest

- **Lasttest**
  Siehe Kapitel 3.3

2.2.3 Zuverlässigkeit

Unter Zuverlässigkeit eines Softwaresystems versteht sich der Punkt, dass das System mit fehlerhaften Eingaben umgehen kann (Grenzwerte und ungültige Äquivalenzklassen). Dies wird aber auf dieser logischen Ebene schon bei den Funktionstests überprüft. Das Qualitätsmerkmal "Zuverlässigkeit" hat den Fokus eher auf der technischen Ebene des Systems.

Darunter fallen der Verfügbarkeitstest und der Ausfallsicherheitstest, wobei sich der Ausfallsicherheitstest aus dem Failovertest, Failbacktest und dem Restart- Recoverytest (Wiederherstellbarkeitstest) zusammensetzt.

- **Verfügbarkeitstest**
  Die Verfügbarkeit wird von der DIN 40041 "... als die Wahrscheinlichkeit, ein System zu einem gegebenen Zeitpunkt in einem funktionsfähigen Zustand anzutreffen" [24] beschrieben.

Demnach ist die Verfügbarkeit das Verhältnis der mittleren fehlerfreien Zeit (Mean Time Between Failure (MTBF)) während eines festgesetzten Betrachtungszeitraums zum gesamten Betrachtungszeitraum. Der gesamte Betrachtungszeitraum ist die Summe aus MTBF und der Zeit, die zwischen dem Eintreten des fehlerhaften Zustandes bis zur Wiederherstellung des fehlerfreien Zustandes vergeht (Mean Time To Repair (MTTR)).

$$Verfügbarkeit = \frac{\text{MTBF}}{(\text{MTBF}+\text{MTTR})}$$

In der DIN 40041 ist auch von der Ausfallrate (Unverfügbarkeit) die Rede, welche sich wie folgt berechnet.

$$Ausfallrate = 1 - Verfügbarkeit$$

Damit ist auch der Verfügbarkeitstest beschrieben. Er überprüft die Verfügbarkeit bzw. Ausfallrate des Systems und wird in Prozent der Betriebszeit angegeben. Wenn zum Beispiel in Verträgen eine Verfügbarkeit von 99,5 Prozent angegeben ist, darf das System bei 100 Tagen Laufzeit 0,5 Tage ausfallen.

Der Verfügbarkeitstest ist im Prinzip eine dauerhafte Messung im Produktionsbetrieb. Es sollten aber auch schon im Rahmen eines Pilotbetriebs Messungen durchgeführt werden, um abschätzen zu können, ob die in den Verträgen vereinbarte Verfügbarkeit eingehalten werden kann.

– **Ausfallsicherheitstest**
Ein System ist nur dann zuverlässig, wenn es gegen einen Ausfall (Failover) gesichert ist. Da eine 100 prozentige Ausfallsicherheit nicht gewährleistet werden kann, muss zumindest gewährleistet werden, dass das System beim Ausfall keine Daten verliert und es in einem definierten Zustand neu gestartet werden kann.

Die Ausfallsicherheit kann durch technische Maßnahmen wie Redundanzen erzielt werden. Das können Ersatzkomponenten wie ein Notstromgenerator sein, aber auch die doppelte oder dreifache Auslegung von Systemkomponenten.

Im Falle eines Failover wird die Funktion einer fehlerhaften Komponente automatisiert von einer redundanten Komponente übernommen. Dabei kann es sich zum einen um technische Komponenten wie Hardware jeglicher Art (Firewall, Server, Netzwerk usw.), aber auch um Teile der Softwareanwendung, wie zum Beispiel die Datenbank, handeln. Der Ausfallsicherheitstest besteht in der Regel aus dem Failovertest, Failbacktest und dem Restart-/Recoverytest (Wiederherstellbarkeitstest).

Failovertest
Hier wird geprüft, ob bei einem Komponentenausfall die redundanten Ressourcen rechtzeitig zugeschaltet werden und ob das System danach funktionsfähig ist.

“Der Failover-Test wird anhand von Notfallszenarien durchgeführt. So kann überprüft werden, wie sich das System bei einem Teilausfall von Systemkomponenten verhält und ob ein Totalausfall des Systems verhindert werden kann.“ [7, p. 74]

Mit Notfallszenarien sind Testfälle gemeint, welche sich aus der Bestimmung von, für den Betrieb kritischen, Komponenten ableiten. In den Notfallszenarien wird beschrieben, wie das System auf den Ausfall zu reagieren hat und wie dieser Ausfall herbeigeführt werden kann.

Praktisch sieht der Failovertest so aus, dass ein Performancetest durchgeführt wird, um dann parallel dazu die definierten Fehlerfälle herbeizuführen. Dabei wird analysiert, wie sich das System verhält.

Failbacktest
Der Failbacktest baut auf dem Failover auf. Es gab einen Ausfall und das System hat dementsprechend darauf reagiert, indem es auf eine redundante Komponente geschaltet hat. In der Zwischenzeit wurde der Fehler beseitigt (Failback) und auf das ursprüngliche System zurück gewechselt.

Nun prüft der Failbacktest, ob das Zurückschalten der ausgefallenen Systemkomponente korrekt erfolgte, indem es mittels Perfomancetest schaut, ob die Komponente dieselben Leistungswerte wie vor dem Ausfall hat.

Wiederherstellbarkeitstest
Wenn es trotz aller Vorkehrungen doch zu einem Ausfall gekommen ist, wo das Failover nicht gegriffen hat, muss das System in einem Zustand sein, in dem es keine Daten verloren hat und es die Möglichkeit gibt, die Datenbanken und Dateisysteme wieder in einen konsistenten Zustand zu bringen.

Der Restart-/Recoverytest hat die Aufgabe zu überprüfen, ob nach einem Ausfall die korrekte Widerherstellbarkeit des Systems vorhanden ist. Es dürfen zum einen keine Daten verloren gehen und zum anderen keine Daten mehrfach angelegt und verändert worden sein.

2.3 Anforderungs- und Testspezifikation

Die Testspezifikation beinhaltet die zur Testdurchführung erarbeiteten Testfälle und Testszenarien. Diese werden in dem Dokument beschrieben und begründet. Es kann nur das getestet werden, was auch spezifiziert wurde. Dazu müssen, aus der Anwendersicht ausgehend, Testfälle konstruiert werden.

Ein typischer Testzyklus kann wie folgt ausschauen.

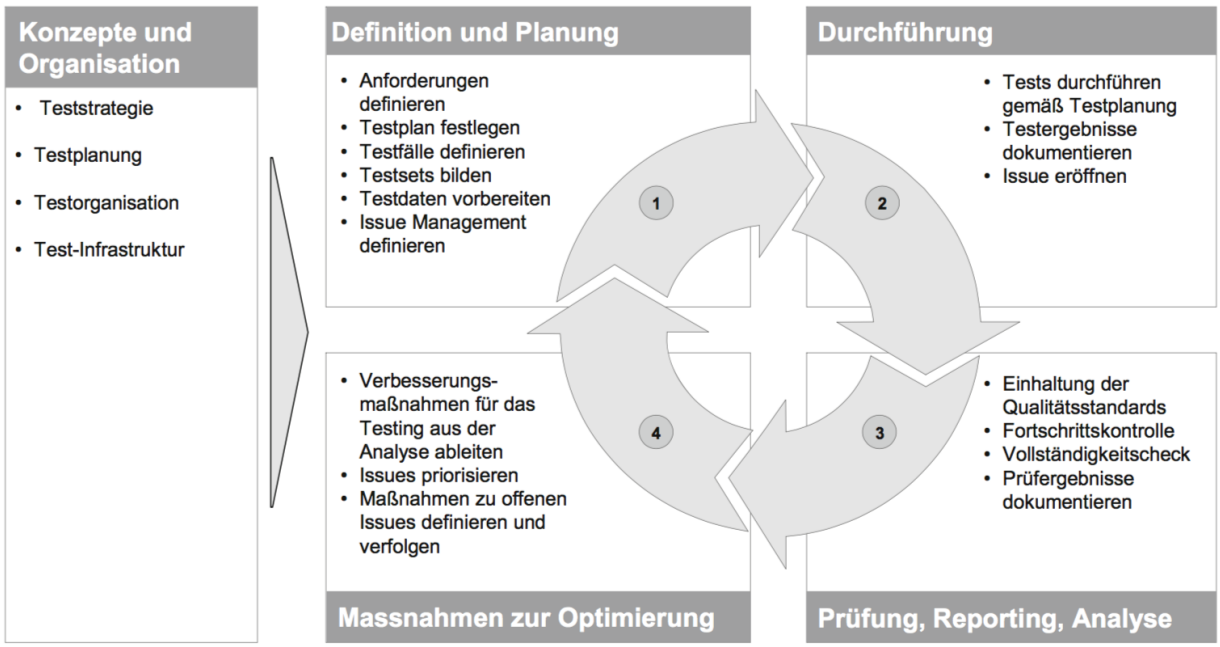

Abbildung 5: Testzyklus [7, p. 13]

Von der Definition und Planung geht es zur Testdurchführung. Danach werden die Ergebnisse beim Punkt Prüfung, Reporting, Analyse präsentiert, um darauf basierend Maßnahmen zur Optimierung durchzuführen.

2.3.1 Testplanung

Der Testplan beinhaltet "...die Ziele, den Umfang, die Methoden, die Ressourcen, einen Zeitplan und die Verantwortlichkeiten für die beabsichtigte Testaufgabe..." [25, p. 24] und spielt dadurch eine wichtige Rolle beim Testen. Er definiert den gesamten Testprozess.

2.3.2 Testdaten

Ein wichtiger und oft unterschätzter Punkt ist das Testdatenmanagement, welches für die Planung und Bereitstellung der Testdaten verantwortlich ist.

In dem Rahmen muss geklärt werden, für welche Tests automatisch Testdaten erzeugt werden können, wo auf anonymisierte Datenbestände geachtet werden muss, von wem die Testdaten kommen, wer sie wie übergeben bekommt und wie man die Testdaten zur Archivierung und Testwiederholung speichert.

Bei den Testdaten ist zwischen den Eingabedaten und Zugriffsdaten (hier wird zum Beispiel nur lesend darauf zugegriffen) zu unterscheiden. Diese Testdaten müssen mit den Testfällen aufeinander abgestimmt sein. Dazu ist es hilfreich, die Zusammengehörigkeit zwischen Testfällen und Testdaten zu dokumentieren. Dabei ist auf das Testdatenvolumen zu achten, dieses sollte möglichst klein ausfallen aber dennoch so groß wie nötig sein.

2.3.3 Testdurchführung

Sind die Tests im Vorfeld gut geplant worden, sollte es bei der Testdurchführung keine Probleme geben. Trotz ordnungsgemäßer Planung kann es zum Beispiel schon durch kleine Konfigurationsfehler zum Testabbruch oder falschen Ergebnissen kommen.

Daher ist es unumgänglich, das zu testende System während des Testablaufs genau zu beobachten. Es ist empfehlenswert, die Tests mehrfach zu wiederholen und die jeweiligen Testergebnisse zu sichern. Im Nachhinein können so die Ergebnisse verglichen werden, um auf Abweichungen zu kommen.

2.3.4 Automatisierte Test

Um für jede Testwiederholung möglichst gleiche Voraussetzungen zu schaffen ist es angebracht, seine Tests zu automatisieren.

Gerade nach einer Änderung an der Software ist es wichtig zu wissen, ob die gewünschte Funktionalität noch gegeben ist. Um dies zu überprüfen, sollte es nach jeder Änderung einen Test geben. Dazu wird die Software jeden Tag zu einer bestimmten Uhrzeit getestet (Regressionstest).

Würde dies jedes Mal manuell geschehen, wäre der Aufwand viel zu groß, anders bei automatisierten Tests. Dort muss der Test nur noch geringfügig angepasst werden, um dann automatisiert abzulaufen. Dabei liefert der automatisierte Test als Metrik die Anzahl der erfolgreichen Testfälle. Dadurch ist die Softwarequalität messbar und die Nebeneffekte der Änderungen auf andere Teile des Systems sind erkennbar.

2.3.5 Testergebnisse

Die Tester definieren im Vorfeld der Tests die möglichen Resultate und vergleichen diese mit den beim Test wirklich produzierten Testergebnissen. Dieser Vergleich kann entweder positiv oder negativ ausfallen.

Darüber hinaus kann man die Testergebnisse verschiedener Testzyklen miteinander vergleichen, um Tendenzen zur Qualitätsentwicklung zu erkennen, ob die Softwarequalität gleich bleibt, sich verbessert oder verschlechtert.

3 Lasttest von Microservices

Die in der Fallstudie zu (Last-)testende Anwendung basiert auf einer Microservices Anwendungssystemarchitektur, welche unter Zuhilfenahme des Java Spring Framework implementiert wurde. Dieses Kapitel erklärt vorbereitend auf die Fallstudie die Grundlagen zu Microservices, dem Spring Framework und Lasttests.

3.1 Microservices

Gerade im Javaumfeld gibt es sehr viele monolithische Anwendungen. In diesem Fall geht es nicht um die Größe der Anwendung, sondern um die Softwareverteilung. Meist sind es modularisierte Programme, welche aber dann in einem einzelnen Web application ARchive (WAR) File ausgeliefert werden.

Das Problem solcher Anwendungen ist es, dass die komplette Software bei kleinsten Änderungen neu kompiliert und getestet werden muss. Des Weiteren fehlt oftmals die Möglichkeit, die Anwendung schnell und unkompliziert zu erweitern.

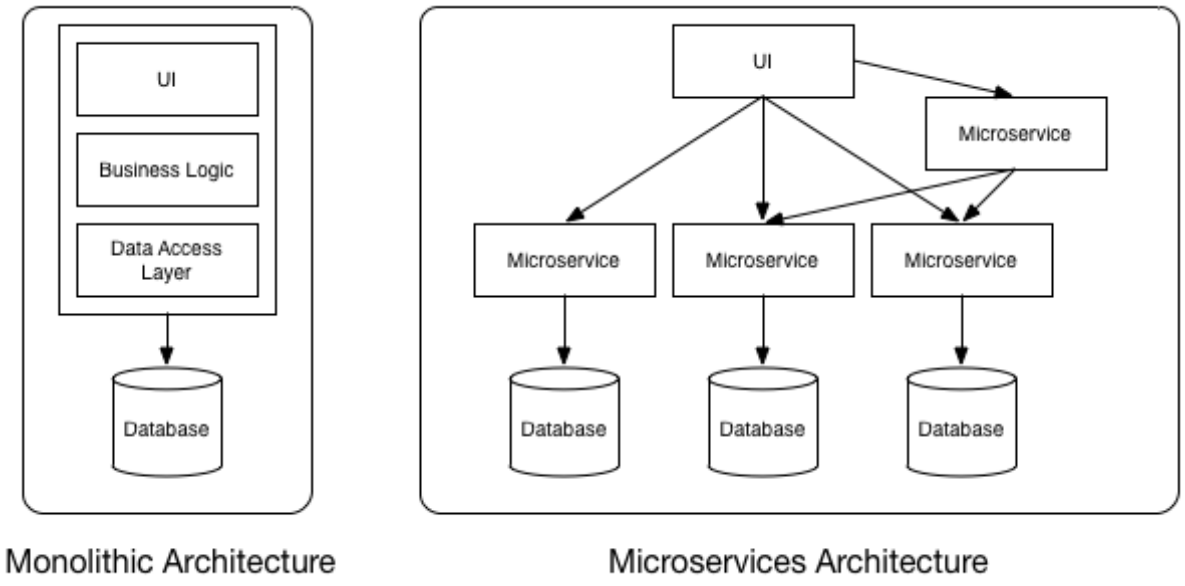

Abbildung 6: Aufbau Microservices und monolithische Anwendungen [26]

Um diesem Problem zu begegnen, bietet sich eine Architekturänderung, hin zur Microservice Anwendungsarchitektur, an. Das Ziel dieser Architektur ist es, möglichst kleine, voneinander unabhängige, Services zu schaffen (siehe Abbildung 6), welche sich getrennt voneinander verteilen lassen. Diese Microservices sollen möglichst unabhängig voneinander sein, so dass sie ohne die anderen Services start- und kompilierbar sind und jeweils auf einem eigenen Prozess basieren.

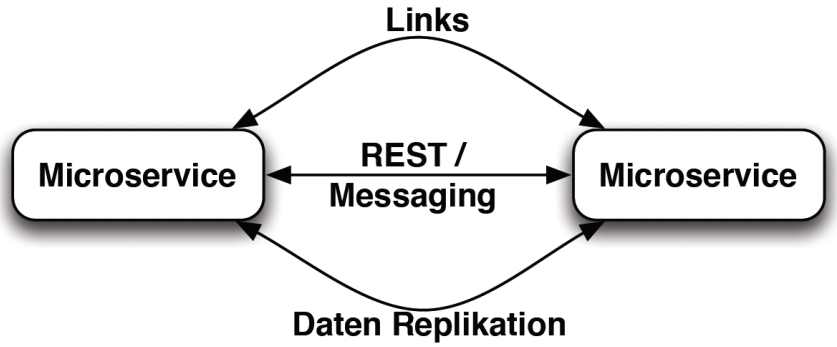

Abbildung 7: Kommunikation der Microservices [27]

Die einzelnen Services kommunizieren über definierte Schnittstellen nach außen, welche meistens dem Representational State Transfer (REST) Schema zugrunde liegen (siehe Abbildung 7)

Wichtig zu sagen ist außerdem, dass Microservices über einen reinen Service hinaus gehen, da sie auch meist ihr eigenes Graphical User Interface (GUI) beinhalten und die Logik bis zur Datenspeicherung hinab reicht, was eine Reduzierung des Übertragungs-Overhead zur Folge hat. Die Summierung der einzelnen Microservices stellt eine einzelne Applikation dar, welche auf derselben Hardware, oder aber räumlich verteilt laufen kann.

Die dem Microservice zugrunde liegende Idee ist keine neue, sie basiert auf dem Gesetz von Conway aus dem Jahre 1968. Dieses, von dem Informatiker Melvin Edward Conway formulierte, Gesetz besagt, dass Systemstrukturen durch die Kommunikationsstrukturen der Einrichtung, welche die Systeme entwickelt, vorbestimmt sind.

"Organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations." [28]

Ein Beispiel ist, dass ein Softwareprojekt aus drei Teams besteht: einem GUI-Team, einem Datenbank-Team und einem Team für die Geschäftslogik. Nach dem Gesetz von Conway würde nun auch die Software so aufgebaut sein, dass es eine GUI-, eine Datenbank- und eine Geschäftslogik-Komponente gibt. Basierend darauf gibt es eine neuere Interpretation dieses Gesetzes, in der man dazu übergeht, den Ansatz umzukehren. Die Organisation bildet nun die Struktur der Softwarearchitektur ab.(vergleiche [27])

Daraus "... ergibt sich ein Dualismus: Die Architektur und die Teamstruktur sind nur zwei Seiten derselben Medaille. Beide Komponenten beeinflussen sich so stark, dass sie eigentlich untrennbar sind. Das wirft folgende Frage auf: Wenn die Organisation und Kommunikation großer Projekte so komplex ist, kann die Softwarearchitektur dieses Problem gegebenenfalls lösen?" [27]

Aus diesem Gedanken ist die Microservice Systemarchitektur entstanden. Um die Komplexität zu senken und die eigentlichen, zugrunde liegenden, Abläufe des Systems abzubilden, ist man dazu übergegangen, die Struktur nach fachlichen Belangen aufzuteilen, wie zum Beispiel Suche, Bestellung und Registrierung. Diese Komponenten müssen über Schnittstellen kommunizieren, um miteinander arbeiten zu können. Eine Suche führt zum Beispiel zu einer Bestellung, also müssen die Daten von der Suchkomponente zu der Bestellkomponente übergeben werden. Dazu kommt, dass jede Komponente unabhängig voneinander entwickelt und veröffentlicht werden kann, solange die Schnittstellen korrekt definiert sind. Aus dieser Idee der Komplexitätsreduzierung, der funktionalen Unabhängigkeit und als wichtigstem Punkt, dem unabhängigen Deployment, sind die Microservices entstanden. Im Prinzip stellen sie eine neue (durch die Kommunikation untereinander) Art der Modularisierungstechnik dar.(vergleiche [27])

3.2 Spring Framework

Das Spring Framework ist ein Open Source Framework, welches eine leichtgewichtige Alternative zu der komplexen Java Platform Enterprise Edition (J2EE)-Plattform bietet und dabei sehr viel Funktionalität bereitstellt. Es hat das Ziel, die J2EE Entwicklung einfacher zu gestalten und dabei bestmögliche Programmierpraktiken zu forcieren.

- **Bereiche abdecken, zu denen andere Frameworks nicht gelangen**
  Damit ist die Integration bestehender Technologien und das Anbieten von Funktionalität zur Abstraktion anderer Frameworks gemeint, um ein leichtes Auswechseln dieser zu gewährleisten (zum Beispiel eine transparente Sicht auf Object/Relational Mapper (O/R-Mapper))

- **Modularisierung**
  Durch den modularen Aufbau von Spring ist es möglich, nur Teile des Frameworks einzusetzen. Zum Beispiel kann man das Framework nutzen, um nur ein bestimmtes Problem der Software zu lösen und den Rest ohne Spring umzusetzen.

- **Einfache Bereitstellung von Funktionalität**
  Einfache, häufig vorkommende Aufgaben, können ohne großen Programmieraufwand umgesetzt werden.

- **Best-Practise**
  Spring basiert auf bewährte Programmierpraktiken und forciert auch dessen Umsetzung bei der Implementierung von Spring in die eigene Anwendung. Zum Beispiel arbeitet Spring viel mit Interfaces, um von der Implementierung zu abstrahieren.

- **Keine Abhängigkeit von Spring**
  Nutzt man Spring, ist die Anwendung nur lose an Spring gekoppelt und die Bereiche die Spring nutzen, können ohne Probleme auf andere Technologien wechseln.

- **Zentrale Konfiguration**
  Spring besitzt eine konsistente Konfiguration über alle Bereiche hinweg.

- **Testbarkeit**
  Aufgrund des starken Fokus auf Interfaces und Plain Old Java Objects (POJOs) können Springanwendungen sehr leicht und vollständig getestet werden.

Um das umzusetzen, steht bei Spring die Entkoppelung der verschiedenen Komponenten der Anwendung im Fokus. Dazu basiert Spring auf einem POJO-orientierten Programmiermodell. POJO bedeutet Plain Old Java Object und meint eine einfache Java Klasse ohne komplexe Strukturen wie es die in J2EE genutzten, Enterprise JavaBeans (EJB) darstellen. Das bedeutet, dass die Java Objekte ohne vielfältige externe Abhängigkeiten, wie zum Beispiel zwingend zu implementierende Schnittstellen, Namenskonventionen und Annotationen auskommen.

Spring setzt auf eine konsequente Umsetzung bestehender Standards und ist dadurch in vielen bestehenden Anwendungen leicht integrierbar und Plattformunabhängig.

Spring setzt dabei auf ein objektorientiertes Umsetzungsparadigma namens Inversion of Control (IoC), welches die Funktionen eines Anwendungsprogramms bei einer Standardbibliothek registriert, um dann später bei Bedarf von dieser aufgerufen zu werden. Damit ist gemeint, dass nicht die Anwendung, sondern das Framework den Kontrollfluss steuert. Nach dem Prinzip “don’t call us, we’ll call you“ , welches zum Beispiel durch das Listener/Observer Pattern umgesetzt wird.

Zusätzlich gibt es viele Erweiterungen für Spring, welche dem Framework zusätzliche Funktionalitäten bieten.

- **Spring Boot**
  Entwicklung eigenständig lauffähiger Spring-Anwendungen, welche ohne Extensible Markup Language (XML)- Konfiguration auskommen und alle notwendigen Klassenbibliotheken beinhalten.

- **Spring ORM**
  O/R-Mapper mit Unterstützung für Hibernate.

- **Spring LDAP**
  Einfacher Zugriff auf Lightweight Directory Access Protocol (LDAP)-Systeme.

- **Spring MVC**
  Zur Erstellung von Webanwendungen auf Basis der Servlet Application Programming Interface (API).

- **Spring Security**
  Absicherung von Java Anwendungen (Applikationen und Webanwendungen).

- **Spring Web Services usw.**
  Zur Erstellung von Webservices.

3.3 Lasttest

Durch einen Lasttest kann geprüft werden, wie sich das Gesamtsystem verhält, wenn die Last auf das System steigt. Dies ist ein wichtiger Punkt für die Anwendung, denn selbst, wenn diese alle gewünschten Funktionen korrekt umgesetzt hat, wird sie bei den Nutzern nicht bestehen, wenn sie zu langsam ist und nicht korrekt funktioniert.

Gerade bei Webanwendungen, welche im Grunde Microservices sind, stellt sich die Frage, wieviele gleichzeitige User auf den Service zugreifen können. Diese Frage der maximalen Grenzen klärt der Lasttest.

Um diese Grenzen auszuloten gibt es mehrere Wege, um die gewünschte Last zu erzeugen. Zum einen kann man die Anzahl der gleichzeitigen Zugriffe auf das System erhöhen, des weiteren kann man die Pausen zwischen den Zugriffen variieren oder auch die absolute Zugriffsanzahl pro Zeiteinheit. Es ist auch möglich, das System mittels großer Datenmengen unter Last zu setzen, aber dies fällt nicht unter den Lasttest, sondern unter den Begriff Massentest/Volumentest und wird hier nicht näher beleuchtet.

Beim Lasttest werden die Werte solange erhöht, bis das System abnormale Reaktionen zeigt. Dadurch werden Schwachstellen, die sogenannten Bottlenecks, aufgezeigt. Gerade bei Microservices ist es nicht so einfach diese Schwachstellen zu lokalisieren, da sie überall auftreten können. Vom Webserver hin zur Applikation, bis zur Datenbank/Datenbankserver oder auch in der Netzwerkinfrastruktur (Router, Firewall usw.).

Folgende Daten können Teil einer Lasttestauswertung sein:

- Statistiken zu den clientseitig gemessenen Antwortzeiten in Abhängigkeit von der Last
- Statistiken zum erreichten Durchsatz in Abhängigkeit von der Last
- Anteil der auftretenden Fehler während des Testablaufs, Auswirkung dieser Fehler auf die Signifikanz der Ergebnisse
- Auslastung der Prozessoren und Plattensysteme in der Testumgebung
- Netzwerkauslastung und Netzwerklatenzzeiten
  [29, p. 40]

3.4 Testwerkzeuge

Es gibt eine Menge an Tools, um einen Lasttest durchzuführen. Dabei ist es gar nicht so einfach, das für sich passende Produkt zu finden. Hier ein kleiner Ausschnitt.

Apache JMeter [16] - Selenium [17] - Desmo J [30] - The Grinder [31] - JUnitPerf [32] - EJP [33] - SimpleProfiler [34] - JPROF [35] - EclipseProfiler [36] - P6SPY [37] - JStress [38] - LoadrunnerTM [39] - PerfectLOAD [40] - JProbeTM [41] - HP LoadRunner [15]

Bei der Fallstudie ist eine Vorgabe, dass das Testwerkzeug Open Source und von einer aktive Community umgeben ist. Dies schränkt die Auswahl ein, da am Markt doch sehr viele kommerzielle Produkte vertreten sind. Nach genauerer Recherche sind die Tools Selenium und Apache JMeter in die engere Auswahl gekommen. Beide sind Open Source und haben eine aktive Community.

	Apache Jmeter	Selenium	Desmo J	The Grinder	P6SPY	HP LoadRunner	JUnitPerf
Aktive Community	X	X	--	--	--	--	--
Grafische Auswertung	X	--	--	--	--	--	--
HTTP/HTTPS	X	X	--	X	--	X	--
JDBC	X	--	--	--	X	X	X
Leicht anpassbar	X	--	--	X	X	--	X
Open Source	X	X	X	X	X	--	X
POST/GET	X	X	--	X	--	X	--

Tabelle 5: Entscheidungstabelle: Testwerkzeug

Das Tool der Wahl ist JMeter von der Apache Software Fondation geworden, da es auf einer abstrahierten Ebene ansetzt. Beim Selenium Framework gibt es die Möglichkeit, Lasttests auf Basis von Java Threads selber zu programmieren. Das "Problem" an der Sache ist, dass das Selenium Framework ein generelles Framework zum Testen von Webanwendungen ist und dabei auch die Möglichkeit bietet, Lasttests durchzuführen.

Apache JMeter hingegen ist ein auf Lasttest spezialisiertes Werkzeug, welches auf einer höheren Abstraktionsebene aufsetzt. Es hat eine GUI und liefert gleich eine grafische Auswertung der Messwerte mit. Es bietet obendrein auch die Anpassbarkeit der Tests mittels einer Skriptsprache, ohne aber die Komplexität einer Programmierung, auf Basis von Java Threads, zu fordern. Apache JMeter beschränkt sich dabei nicht nur auf Hypertext Transfer Protocol (HTTP) und Secure Hypertext Transfer Protocol (HTTPS), sondern kann ebenso mit File Transfer Protocol (FTP), Simple Object Access Protocol (SOAP) und Java Database Connectivity (JDBC) umgehen.

4 Fallstudie: Lasttest der OMiLAB Microservices Anwendungssystemarchitektur

Das Open Models Laboratory (OMiLAB) ist Teil der Forschungsgruppe Knowledge Engineering an der Universität Wien unter der Leitung von Herr Prof. Dr. Dimitris Karagiannis.

Das OMiLAB ist ein Ort (physikalisch wie virtuell) zur Entwicklung von Modellierungsmethoden. Dazu baut es auf den Gedanken der Offenheit auf und lädt weltweit Wissenschaftler ein, ein Teil der OMiLAB Gemeinschaft zu werden. Dies können alle, an Modellierung interessierte, Personen oder Institutionen vor Ort in den Räumlichkeiten des OMiLAB wahrnehmen oder mittels der Internetplattform, welche unter http://www.omilab.org zu erreichen ist. Diese Plattform stellt eine Reihe von nützlichen Werkzeugen rund um das Themengebiet bereit und bietet eine Vielzahl von Möglichkeiten, um einen Diskurs anzuregen und sich zu vernetzen.

“The lab’s idea is to act as a facilitator to the development and application of scientific methods to communities who value models, and implicitly modelling methods. It is open to all application domains which can benefit from the use of models; it promotes openness of community projects and encourages communities to share their projects to the extent feasible.“ [42] Eine genauere Beschreibung des OMiLABs ist unter [43] und eine Beschreibung der Open Model Initiative unter [44] zu finden.

4.1 Zielsetzung

Im Zuge der Neuentwicklung der Webseite http://www.omilab.org soll im Rahmen der Fallstudie das Lastverhalten der Applikationen getestet werden. Die Webseite basierte bisher auf der Portallösung Liferay [45] und wurde, unter Zuhilfenahme des Java Spring Frameworks, durch eine Eigenentwicklung auf Basis von Microservices ersetzt.

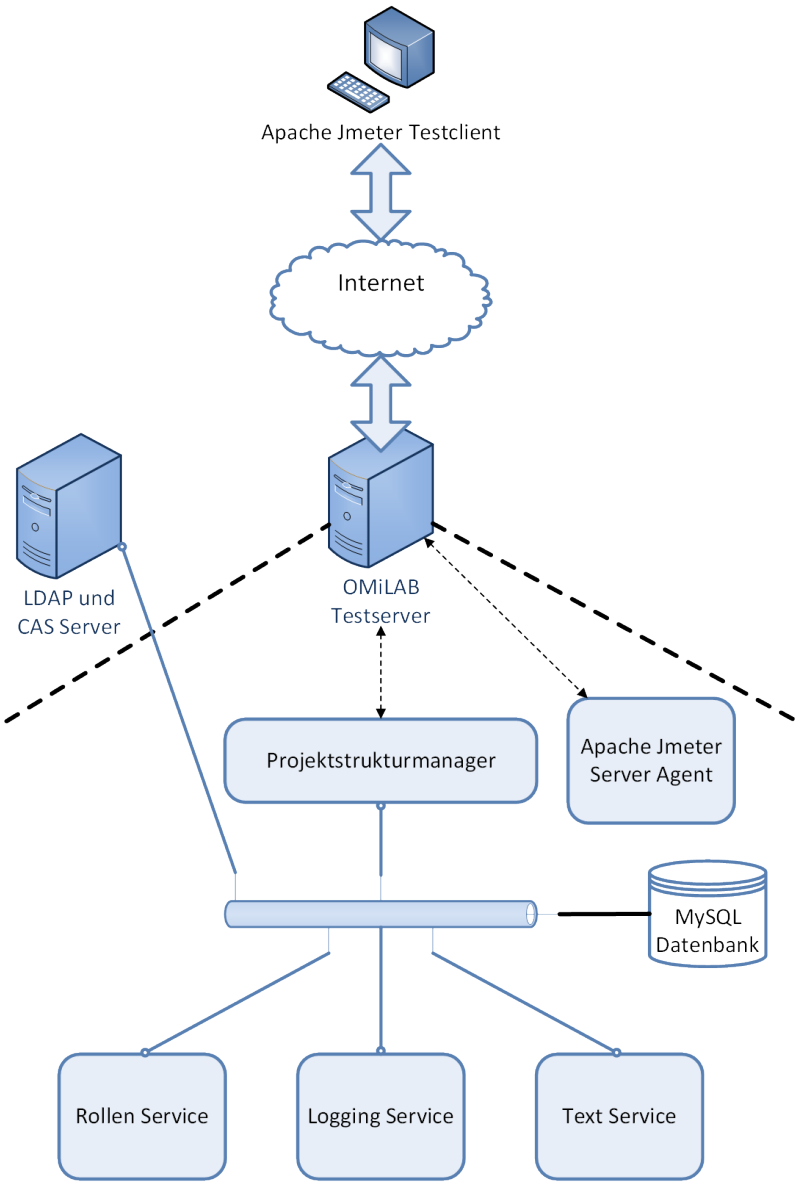

Abbildung 8: Architekturmodell Testumgebung

Die Microservices Anwendungssystemarchitektur ist dabei so aufgebaut, dass es einen sogenannten Projektstrukturmanager (PSM) gibt (siehe Abbildung 8), an den die Browseranfragen gestellt werden. Der PSM verarbeitet diese Anfragen aufgrund der geforderten Uniform Resource Locator (URL) (REST-Schema) und stellt das Ergebnis dem Browser bereit.

Im Hintergrund sind verschiedene Services an den PSM gekoppelt. Diese Services übernehmen die eigentlichen Aufgaben, da der PSM nur vermittelnd tätig ist und die Navigationsstruktur der Seite zur Verfügung stellt. Die eigentlichen Inhalte werden von den Services an den PSM übergeben und dieser bettet sie in die Webseite ein. Ein Beispiel für einen Service ist der Role Service. Dieser Service verwaltet die Rollen der User im Zusammenhang mit den verschiedenen Projekten, welche auf der Webseite präsentiert werden.

Ist der User zum Beispiel Admin des Projektes, darf er die Projektbeschreibungen ändern. Dieser Vorgang würde dann den Text Service ansprechen, welcher dem PSM die Änderungsfunktionalität zur Verfügung stellt. Der zentrale Punkt der Architektur ist der Projektstrukturmanager.

4.2 Vorbedingungen

Um die Lasttests durchzuführen, wird die oben beschriebene Applikation benötigt. Diese liegt in einer selbstinstallierenden Form vor, das bedeutet, dass sich beim ersten Aufruf alle Abhängigkeiten selbst auflösen und installieren. Abhängigkeiten können Elemente wie die Datenbank sein, aber auch Namespaces und so weiter. Die Anmeldung an das System geschieht über ein Central Authentication Service (CAS) Ticketsystem in Verbindung mit einer LDAP Instanz.

Im Rahmen der Applikationsinstallation, wird eine, dem Produktivsystem entsprechende, Serverumgebung benötigt. In diesem Fall ist es eine Virtuelle Machine (VM), welche vom OMiLAB zur Verfügung gestellt wird. Des weiteren muss diese VM von außen (über das Internet) erreichbar sein.

4.3 Testeinstellungen

Die Infrastruktur setzt sich aus den folgenden Komponenten zusammen.

Testclient

Hardware:

MacBook Pro	
CPU:	Intel Core i7 2,3 Gigahertz (GHz)
Speicher:	16 Gigabyte (GB) Random-Access Memory (RAM)

Software:

Safari Webbrowser	v 9.0
Apache JMeter	v 1.0
Apache Directory Studio	v 2.0.0
MySQL Workbench	v 6.3.3.0

Virtuelle Maschine (VMWare)

Hardware:

CPU:	Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00 GHz
Speicher:	4 GB RAM
Festplatte:	24 GB

Software:

Ubuntu:	v 14.04
Tomcat:	v 7
JDK:	v 1.7
MySQL:	v 5.5
JMeter Server Agent:	v 2.2.1
OMiLAB Microservices:	v 0.1.1-SNAPSHOT

Offene Ports:

22	Secure Shell (SSH)
80	HTTP
443	HTTPS
1099	JMeter Server Agent
3306	Datenbank
8000	Projektstrukturmanager / Applikation
8080	Tomcat

LDAP:

Administrator:	admin
Normaler User:	testUser, kersjes

Applikation:

Administrator:	kersjes
Projektmember:	testUser

Nachdem die Infrastruktur aufgebaut und die Software installiert ist, müssen noch die Testsettings eingestellt werden. Dabei hervorzuheben sind die Programme Apache JMeter, Apache JMeter Server Agent und die OMiLAB Microservices.

OMiLAB Microservices
Beim ersten Aufruf des Projektstrukturmanagers (/psm) erscheint eine Eingabemaske, in der man grundlegende Einstellungen vornehmen muss. Nach dem Absenden der Eingabemaske installieren sich die Microservices von allein (inklusive der initialen Erstellung und Befüllung der Datenbank).

Apache JMeter und Apache JMeter Server Agent
Bei diesen beiden Anwendungen sind die Startparameter hervorzuheben.

```
➜  ~  cd /usr/local/Cellar/jmeter/2.13/bin/
➜  bin git:(master) ✗ JVM_ARGS="-Xms6144m -Xmx6144m" jmeter; exit;
```

Abbildung 9: Startsettings Apache JMeter

Der Start vom JMeter (Abbildung 9) erfolgt mit dem Parameter JVM_ARGS (was nicht ein JMeter, sondern ein Java Virtuell Machine spezifischer Parameter ist). Dies gibt an, wieviel Speicher die Java Virtuell Machine (JVM) beim Start (Xms) und wieviel Speicher sie als Maximum (Xmx) belegen darf. Dieser Parameter ist ein wichtiger Punkt, wenn es um die Performance der Anwendung geht. Je größer der zugebilligte Speicher ist, umso mehr gleichzeitige Userzugriffe kann JMeter simulieren.

```
christian@omi-ck:~/ServerAgent$ ./startAgent.sh --udp-port 0 --tcp-port 1099
INFO    2015-08-11 11:07:32.503 [kg.apc.p] (): Binding TCP to 1099
INFO    2015-08-11 11:07:32.548 [kg.apc.p] (): JP@GC Agent v2.2.0 started
```

Abbildung 10: Startsettings Apache JMeter Server Agent

Um auf dem Server den Apache Server Agent zu starten (Abbildung 10), muss man dem Server Agenten mitteilen, an welchem Port er lauschen soll. Dies kann ein User Datagram Protocol (UDP)- und/oder Transmission Control Protocol (TCP)-Port sein. Wie oben in den Einstellungen der Virtuellen Machine zu sehen, wurde hier der TCP-Port 1099 definiert. Der Apache Server Agent dient dazu, direkt vom Apache JMeter aus, auf serverseitige Daten, wie zum Beispiel Speicherverbrauch und CPU Last, zuzugreifen.

4.4 Testbeschreibung

Die Lasttests laufen so ab, dass im Apache JMeter sogenannte Testpläne erstellt werden. Diese Testpläne stellen eine Übersetzung der Testszenarien dar, welche im Vorfeld ausgearbeitet wurden. Er ist im Prinzip eine Art von Container, welcher alle Komponenten des Testplans beinhaltet. Diese Komponenten können wiederum Container für andere Komponenten sein. Daraus ergibt sich ein hierarchischer Aufbau, welcher bei der Ausführung Schritt für Schritt abgearbeitet wird.

Um die Testpläne zu erstellen, bietet JMeter eine Reihe von Funktionalitäten an. Ein Testplan ist grundlegend erst einmal so aufgebaut, dass eine grafische Oberfläche bereit gestellt wird, in der man, unterstützt durch verschiedene Kontextmenüs, nach und nach den gewünschten Plan erstellen kann.

Thread-Gruppe
Am Anfang eines jeden Testplans steht die Definition zumindest einer Thread-Gruppe. Diese Thread-Gruppe definiert die Anzahl der Threads, welche die Benutzer simulieren. Zusätzlich bestimmt man eine Anlaufzeit, innerhalb derer alle Threads gleich verteilt starten. Außerdem kann noch angegeben werden, wie oft die Thread-Gruppe aufgerufen werden soll.

Nach der Erstellung der Thread-Gruppe bietet JMeter mehrere Möglichkeiten, seine Testszenarien abzubilden. Die wichtigsten Elemente sind der Logik-Controller, die Konfigurations-Elemente, der Timer, der Sampler und der Listener.

Logik-Controller
Der Logik-Controller bietet die Möglichkeit, die Abarbeitung der einzelnen Komponenten des Testplans zu manipulieren. Es kann zum Beispiel angegeben werden, dass einzelne Komponenten nur einmal durchlaufen oder dass sie mehrmals wiederholt werden. Es sind aber auch Steuerungen möglich, dass gewisse Komponenten zufällig ausgewählt werden und so weiter.

Timer
Hat man in der Thread-Gruppe eine mehrfache Wiederholung definiert, kann man mit dem Timer bestimmen, wie lange jeder Thread nach einem Durchlauf angehalten wird, bevor er erneut abläuft. Dazu stehen verschiedene Timervarianten, wie zum Beispiel konstante oder Zufalls-Timer, zur Verfügung.

Um die eigentlichen Serveranfragen zu definieren, stehen die Sampler zur Verfügung. Diese Sampler gibt es für die verschiedensten Protokolle, wie zum Beispiel für HTTP, FTP, JDBC, LDAP und so weiter.

Konfigurations-Element
Durch die Konfigurations-Elemente lassen sich für die Sampler und Thread-Gruppen verschiedene Elemente anlegen, um Default-Parameter zu definieren. Diese Default-Parameter gelten dann für eine Gruppe von gleichen Objekten.

So kann man zum Beispiel für das oben gezeigte HTTP-Request Objekt einen default Server eintragen. Das ist sehr hilfreich innerhalb eines komplexen Testplans, da es dort eine große Menge an HTTP-Request Objekten geben kann, welche auf denselben Server zugreifen.

Listener
Zur Verarbeitung der ermittelten Testergebnisse gibt es die Listener. Die Verarbeitung der Ergebnisse reicht von einer textuellen Beschreibung hin zu einer visuellen in Form verschiedener Graphen. Dies kann auf Threadebene bis hin zur Testplanebene geschehen.

Diese Beschreibung der einzelnen Elemente bietet nur einen kleinen Überblick der Möglichkeiten, welche zur Verfügung stehen. Im Laufe der folgenden konkreten Testplanbeschreibung wird noch vertiefend auf einzelne Elemente eingegangen. Doch zunächst einmal muss geklärt werden, was getestet werden soll.

Testszenarien
Da es sich um eine Neuimplementierung der Software handelt, wird zunächst mittels eines Stresstests ausgelotet, was die Grenzen des zugrunde liegenden Systems (Spezifikationen siehe Punkt Testsettings), im Bezug zu den einzelnen OMiLAB Benutzergruppen (Administrator (Admin), angemeldeter User, anonymer User) sind, wobei der Admin Änderungen vornimmt, der anonyme User nur lesend unterwegs ist und der angemeldete User nur liest, aber im Gegenzug zum anonymen User sich noch ein- und ausloggt.

Szenario	Benutzergruppe	Einloggen	Seite lesen	Seite bearbeiten	Ausloggen
SZ1	Administrator	X	-	X	X
SZ2	Angemeldeter User	X	X	-	X
SZ3	Anonymer User	-	X	-	-

Tabelle 6: Stresstest Benutzergruppen

Nachdem nun die Systemgrenzen klar sind, geht man daran, ein realistisches Testszenario zu entwerfen. In diesem Fall eine Mischung der verschiedenen Benutzergruppen, welche gleichzeitig agieren.

Um realistische Zahlen zu bekommen, wird auf die Google Analytics Daten, der bisherigen omilab.org Seite, zugegriffen. Diese Daten besagen, dass es bei den Seitenzugriffen Peaks von 200 User am Tag gibt, welche eine durchschnittliche Sitzungsdauer von 2:04 Minuten haben.

In den Testdurchläufen mittels Apache JMeter wird die Sitzungsdauer nicht berücksichtigt, beziehungsweise beträgt sie dort 0. Es ist demnach so, dass jeder User ohne Unterbrechung von Seite zu Seite wechselt. Dies bedeutet eine höhere Last im Gegenzug zu Anfragen mit Bedenkzeit. Um eine möglichst realistische Messung durchführen zu können, gibt es die Möglichkeit der Umrechnung, was die echten Werte im Testumfeld bedeuten. Dies geschieht mittels der

Response Time Law [29, p. 37]

$$R = \frac{N}{(X_0 - Z)}$$

R = Antwortzeit
N = Anzahl der Benutzer
Z = Bedenkzeit
X_0 = Systemdurchsatz

Gegeben ist eine maximale Antwortzeit von 1,5 Sekunden (dies ist ein typischer Wert, von dem man in der Praxis ausgehen kann und welcher den User nicht zu ungeduldig werden läßt [46]). Des Weiteren liegt eine durchschnittliche Sitzungsdauer (Bedenkzeit) von 2:04 Minuten (124 Sekunden) vor. Diese Bedenkzeit bezieht sich auf alle Seiten, nicht auf eine einzelne. Google Analytics liefert auch noch eine zusätzliche Information und zwar, dass ein durchschnittlicher User 2,7 Seiten besucht, weshalb es realistisch ist, die Bedenkzeit pro Seite auf 46 Sekunden zu setzen.

Des Weiteren gibt es ein Besucher Peak von 200 Besuchern am Tag. Dabei macht es Sinn einen Buffer von 100 Usern hinzuzufügen, um auch zukünftige Tage mit stärkerem Zugriff zu berücksichtigen. Die Statistiken zeigen nicht auf, wie sich die User über den Tag verteilen, was aber indirekt den Konfigurationsmöglichkeiten von JMeter entgegen kommt. Somit wird der gleichzeitige Zugriff der 300 User getestet.

Es sind alle Variablen bis auf den Systemdurchsatz (X_0) gegeben.

$$X_0 = \frac{N}{(R+Z)}$$

$$X_0 = \frac{300}{(1{,}5+46)} \qquad X_0 \approx 6{,}32$$

Nun muss die Bedenkzeit von 0 Sekunden berücksichtigt werden, mit der im JMeter gearbeitet wird.

$$6{,}32 = \frac{N}{(1{,}5+0)}$$

$$N = 1{,}5 * 6{,}32 \qquad N \approx 9{,}48$$

Dies sagt aus, dass der Lasttest mit 10 virtuellen Nutzern durchgeführt werden kann, was in etwa 300 realen Nutzern mit einer Verweildauer von 46 Sekunden pro Seite entspricht.

Um eine gute Mischung beim Testen zu erhalten, kann ein realistisches Testszenario so ausschauen

Szenario	**Benutzergruppe**	**Einloggen**	**Seite lesen**	**Seite bearbeiten**	**Ausloggen**
SZ4	2 Administratoren	X	-	X	X
SZ5	3 angemeldete User	X	X	-	X
SZ6	5 anonyme User	-	X	-	-

Tabelle 7: Realistisches Testszenario

Das bedeutet, dass sich gleichzeitig zwei Administratoren anmelden, Änderungen vornehmen und sich wieder abmelden, sich drei User anmelden, verschiedene Seiten lesen und sich wieder abmelden, sowie 5 anonyme User (nicht angemeldete Gäste) auf den Seiten surfen.

Testplan
Aus den Testszenarien wurden vier Apache JMeter Testpläne erstellt. Drei Testpläne, welche den Stresstestszenarien (SZ1-SZ3) entsprechen und ein Testplan, welcher die Szenarien SZ4 bis SZ6 zusammenfasst (dieser Testplan entsprich dem Lasttest).

Stresstest

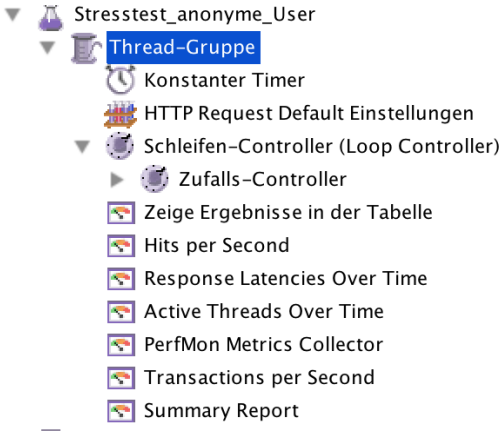

Abbildung 11: Stresstest anonymer User

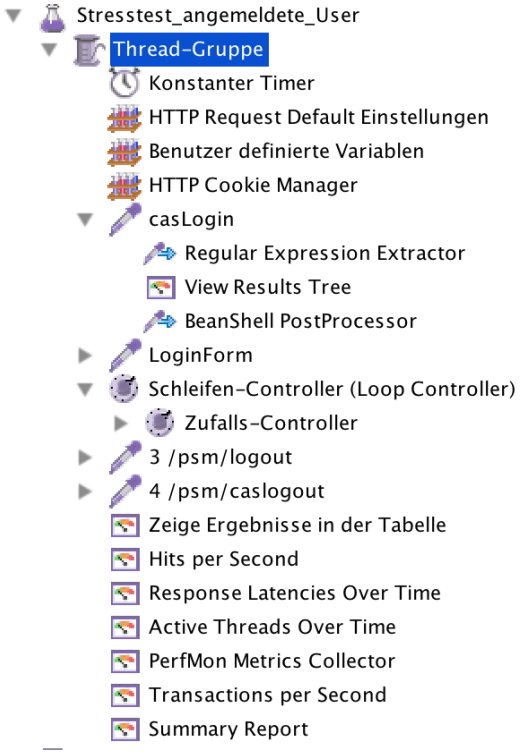

Abbildung 12: Stresstest angemeldete User

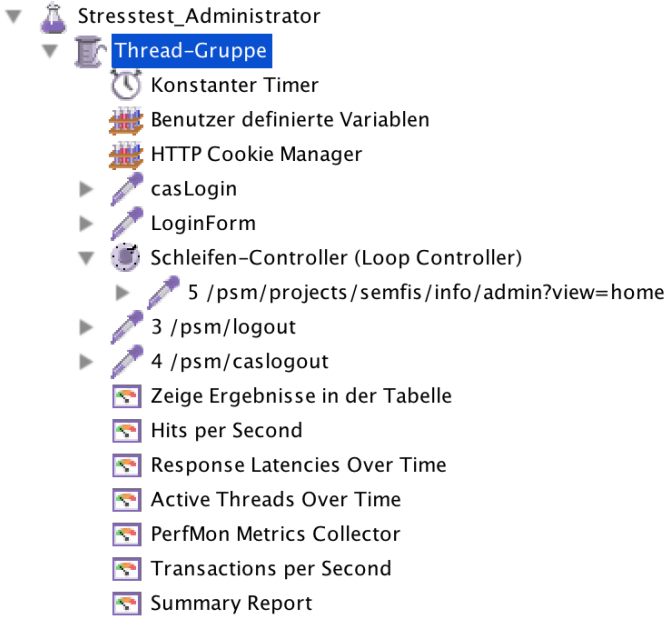

Abbildung 13: Stresstest Administrator

Wenn man sich die Testpläne (Abbildung 11 bis 13) anschaut, sind sie selbsterklärend, allerdings gibt es ein paar Besonderheiten.

- **Schleifen-Controller (Loop Controller)**
  Der Schleifen-Controller (siehe Abbildung 11 bis 13) gehört zu den Logik-Controllern und bietet die Möglichkeit festzulegen, wie häufig die enthaltenen Komponenten wiederholt werden sollen. Hier ist der Wert 3 festgelegt, was die realistische Eigenschaft widerspiegelt, dass der User im Durchschnitt ca. 3 Seiten pro Sitzung besucht.

- **Zufalls-Controller**
  Dieser Controller (Abbildung 11 und 12) gehört ebenfalls zu den Logik-Controllern und macht es möglich, dass eine der enthaltenen Komponenten zufällig ausgewählt wird. Im Zusammenhang mit dem Schleifen-Controller verhält es sich so, dass 3 zufällig gewählte Seiten bearbeitet bzw. besucht werden.

- **Benutzer definierte Variablen**
  Dieses Konfigurations-Element (Abbildung 12 und 13) dient dazu Werte abzuspeichern, so dass alle Komponenten des Testplans darauf zugreifen können.

- **HTTP Cookie Manager**
  Diese Komponente (ebenfalls Teil der Konfigurations-Elemente. Siehe Abbildung 12 und 13), verwaltet die Cookies für die Thread-Gruppe, in der sie definiert wurde. Dies ist ein wichtiger Punkt für das Login und Logout der Anwendung.

- **Regular Expression Extractor und BeanShell PostProcessor**
  Der Regular Expression Extractor ist ein Post-Processor von JMeter (Abbildung 12), welcher es ermöglicht, mittels Perl Regular Expressions, dynamische Inhalte aus einem Server Response auszulesen. Das ist die zentrale Komponente, welche benötigt wird, um das Login und Logout der Anwendung durchführen zu können.

  Der BeanShell PostProcessor ist, wie der Name schon nahelegt, ein Post-Processor, welcher es ermöglicht, BeanShell Skripte zu programmieren und auszuführen.

  Die Anwendung arbeitet beim Login/Logout mit einer Identitätsmanagement Web-Applikation namens CAS (entwickelt von der Yale University). Mittels des Aufrufs des Sampler “casLogin“ wird auf die Login URL des CAS Servers weitergeleitet und im Response Header hängt der Server einen eindeutigen LoginToken (LT) an. Dieser wird durch den Regular Expression Extractor ausgelesen und von dem BeanShell PostProcessor in die benutzerdefinierte Variable “LT“ eingetragen.

  Befindet man sich nun auf der Login Form (Sampler “LoginForm“), wo die Authentifizierung mittels Username und Passwort stattfindet, wird der LoginToken dem Server Request angehängt. Ist die Authentifizierung erfolgreich, validiert der CAS Server, ob der mitgegebene LoginToken in der Datenbank eingetragen ist. Ist dies der Fall, schickt er im HTTP Response ein CAS Cookie (CAS TicketGrantingTickets (CAS TGT)) mit, welcher vom HTTP Cookie Manager verwaltet wird. Die zum Cookie gehörige Identifier (ID) wird ebenfalls mittels des Regular Expression Extractor ausgelesen und vom BeanShell PostProcessor abgespeichert.

  Bei den einzelnen Seitenaufrufen ist es wichtig, innerhalb der Sampler den Punkt “Folge Redirects“ zu aktivieren, da jede Seite automatisch eine Seite aufruft, wo das CAS Ticket auf Gültigkeit überprüft wird. Meldet der User sich ab (hier simuliert mittels der Sampler “casLogout“), überprüft der CAS Server anhand des CAS TGT Cookies, ob der User angemeldet ist und wenn dies der Fall ist, so widerruft der Server das Ticket und macht es somit ungültig.

Lasttest

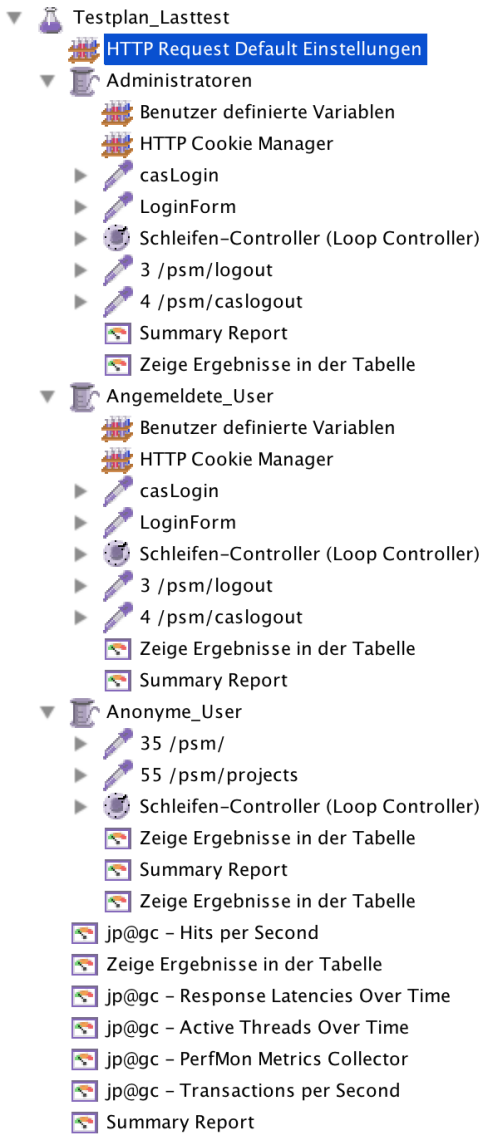

Abbildung 14: Testplan: Lasttest

Beim Lasttest sind die Stresstest Testpläne in einem Testplan zusammen gefasst (siehe Abbildung 14) und von den Thread-Eigenschaften her so eingestellt, dass es den definierten Szenarien SZ4, SZ5 und SZ6 entspricht. Dieser Testplan besteht also aus drei Threadgruppen gleichzeitig, wobei die drei Threadgruppen vom Apache JMeter parallel ausgeführt werden. Somit ist es so, als ob gleichzeitig 2 Administratoren, 3 angemeldete und 5 anonyme User auf der Seite arbeiten.

Die Elemente

- Hits per Second
- Ergebnisse in der Tabelle
- Response Latencie Over Time
- Active Threads Over Time

– PerfMon Metrics Collector
– Transactions per Second
– Summary Report

sind Listener Elemente und präsentieren die Ergebnisse der Testpläne.

4.5 Ergebnisse

Aussagekräftige Metriken der Tests sind die Punkte

– Anzahl der Threads
– CPU- und Speicherverbrauch
– Antwortzeit
– Fehlverhalten

Um relevante Testergebnisse zu bekommen, wurden die Tests am späten Abend, jeweils an drei aufeinander folgenden Tagen durchgeführt. Die Ergebnisse der unterschiedlichen Testserien sind sehr ähnlich und lassen die Vermutung zu, dass es sich um aussagekräftige Ergebnisse handelt, welche hier präsentiert werden.

Stresstests
Ein erstes fehlerhaftes Verhalten macht sich beim "Stresstest_anonyme_User", bei ca. 250 Threads bemerkbar. Beim "Stresstest_angemeldete_User", bei ca. 155 Threads und beim "Stresstest_Administrator", bei ca 65 Threads. Diese Werte zeigen, wieviele einzelne User der unterschiedlichen Usergruppen gleichzeitig unter den gegebenen Konditionen, wie der Punkt "Response Time Law" ausführlich beschreibt, mit der Applikation interagieren können.

Schaut man sich nun den CPU- und Speicherverbrauch an, erkennt man anhand der Graphen das Testverhalten der einzelnen Stresstests.

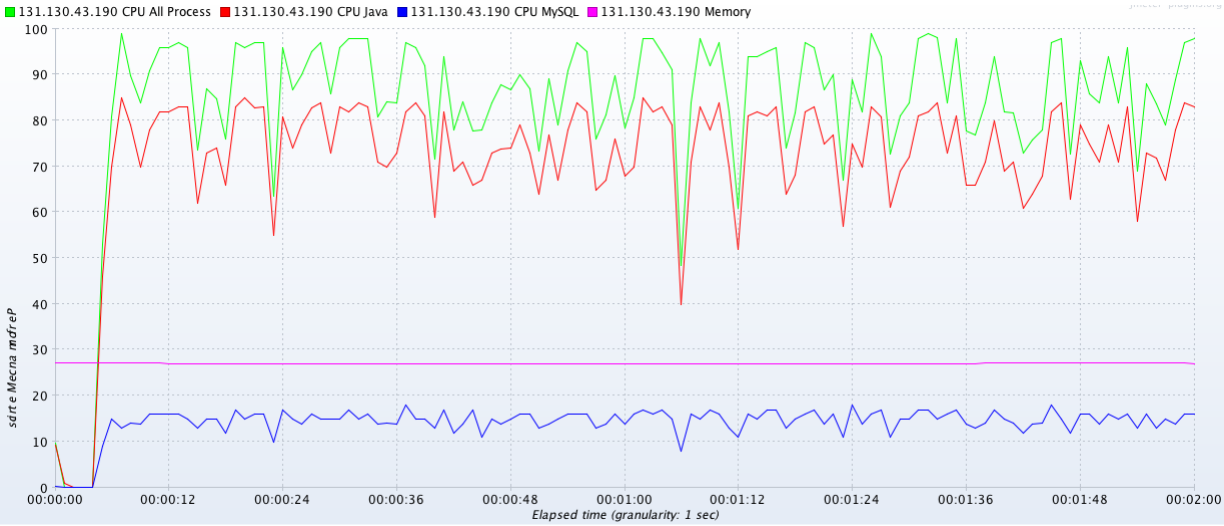

Abbildung 15: Stresstest: Anonyme User - CPU und Speicher

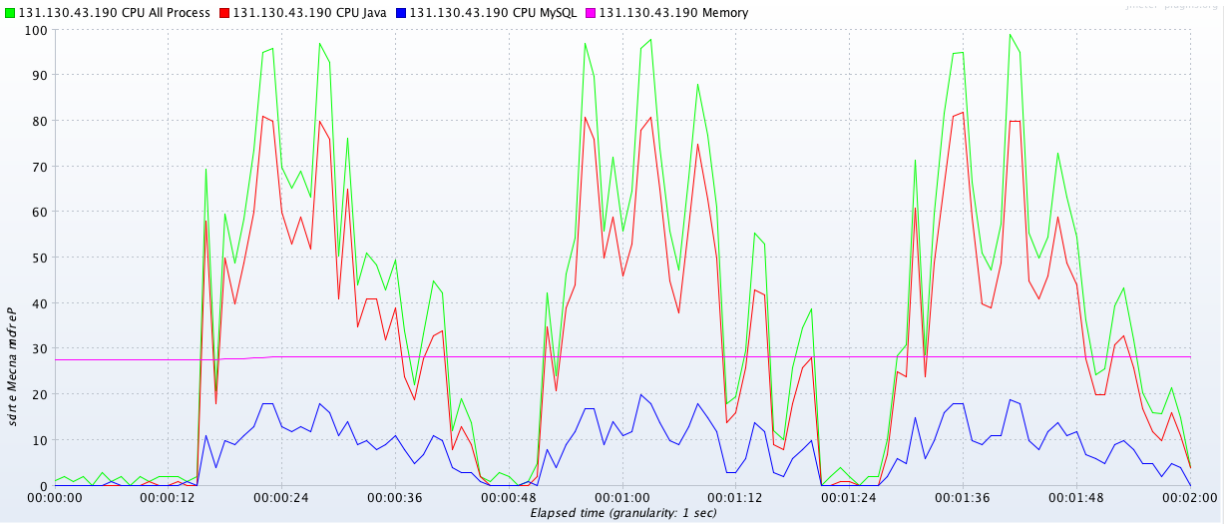

Abbildung 16: Stresstest: Angemeldete User - CPU und Speicher

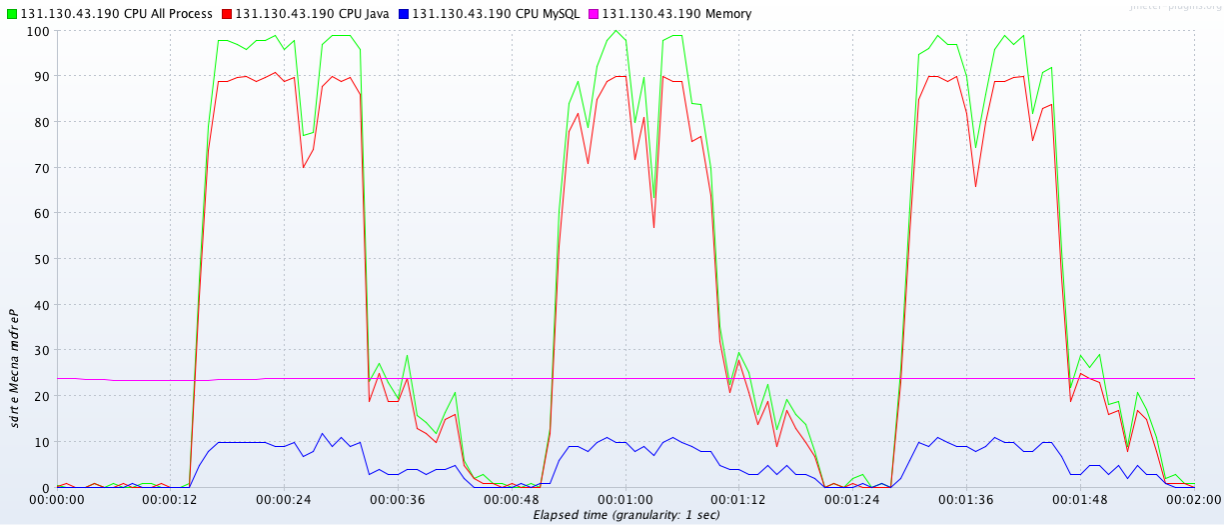

Abbildung 17: Stresstest: Administrator - CPU und Speicher

Die Graphen sind wie folgt aufgebaut, die x-Achse gibt die abgelaufene Zeit an, die y-Achse die Auslastung des Servers in Prozent. Die grüne Linie steht für die Auslastung CPU-Gesamt, die rote Linie für die Auslastung CPU-Java (CPU Verbrauch der Microservices), die blaue Linie steht für die Auslastung CPU-MySQL und die rosa Linie für den Speicherverbrauch.

Der Speicherverbrauch ist konstant gleich bei ca. 23 Prozent. Dies wird daran liegen, dass dem Tomcat Server und der MySQL Datenbank in den Konfigurationen eine maximale Speicher-Nutzungsgrenze vorgegeben wird.

Des Weiteren ist zu erkennen, dass die Auslastung CPU-Gesamt (Summe aus CPU-Java und CPU-MySQL) an die 100 Prozent Marke geht, was dann als Konsequenz zu den Responsefehlern führt, da der Server nicht mehr mit allen Anfragen umgehen kann.

Betrachtet man das Muster der Graphen, erkennt man in der Abbildung 15 das relativ gleichmäßige Surfverhalten der nicht angemeldeten User. Bei Abbildung 16 und Abbildung 17 hingegen, erkennt man in den an- und abschwellenden Kurven das Login und Logout Verhalten der User. Die Spitzen dazwischen spiegeln die eigentlichen Aktivitäten der angemeldeten User/Administratoren wider.

Label	Anz. der Proben	Durchschnitt	Min	Max	% Fehler
/psm/projects/rupert	231	58	21	332	0,00%
/psm/projects/bpfm/info	246	288	45	1035	0,00%
/psm/about	258	37	16	264	0,00%
/psm/	240	126	39	400	0,00%
/psm/projects/rupert/info	221	314	48	979	0,00%
/psm/events	259	128	31	9990	0,77%
/psm/events/modellierung/info	244	326	48	10096	0,41%
/psm/projects/ec/info	234	303	47	998	0,00%
/psm/projects/istar/info	221	325	51	989	0,00%
/psm/projects/cv	250	63	19	538	0,00%
/psm/events/summerschool/info	232	273	46	880	0,00%
/psm/projects/som	225	61	21	289	0,00%
/psm/projects/istar	234	59	19	253	0,00%
/psm/projects/semfis	240	58	19	212	0,00%
/psm/projects/ec	233	105	19	10010	0,43%
/psm/projects/som/info	247	285	49	979	0,00%
/psm/projects/diba/download	223	279	46	1081	0,00%
/psm/projects/som/download	260	292	47	1203	0,00%
/psm/projects	238	80	49	308	0,00%
/psm/projects/bpfm	241	94	22	9096	0,41%
/psm/projects/semfis/info	258	314	49	1029	0,00%
/psm/projects/cv/info	269	296	46	887	0,00%
/psm/projects/cv/download	232	336	49	9283	0,43%
/psm/projects/muviemot/download	231	344	46	10099	0,43%
/psm/services	197	56	33	309	0,00%
Gesamt	5964	197	16	10099	0,12%

Abbildung 18: Stresstest: Anonyme User - Antwortzeit und Fehlverhalten

Label	Anz. der Proben	Durchschnitt	Min	Max	% Fehler
casLogin	773	82	33	200	0,00%
LoginForm	740	37	19	432	0,00%
/psm/projects/ec	75	329	71	1904	0,00%
/psm/projects	90	82	49	202	0,00%
/psm/events/summerschool/info	83	237	47	1332	0,00%
/psm/	108	108	37	512	0,00%
/psm/projects/som/download	72	290	49	1726	0,00%
/psm/projects/ec/info	95	320	51	1509	0,00%
/psm/projects/semfis	95	363	78	2618	0,00%
/psm/events/modellierung/info	77	277	56	1910	0,00%
/psm/projects/cv/download	95	335	49	1908	0,00%
/psm/projects/cv	106	410	76	2101	0,00%
/psm/projects/istar	84	421	78	2397	0,00%
/psm/projects/semfis/info	79	307	47	1495	0,00%
/psm/projects/rupert	99	543	79	2435	0,00%
/psm/projects/som	97	391	81	1962	0,00%
/psm/projects/istar/info	68	295	48	1791	0,00%
/psm/projects/cv/info	85	343	49	1508	0,00%
/psm/projects/som/info	98	316	51	2397	0,00%
/psm/about	92	27	18	136	0,00%
/psm/projects/rupert/info	100	360	53	1884	0,00%
/psm/projects/muviemot/download	84	294	54	1718	0,00%
/psm/projects/bpfm	91	426	76	2395	0,00%
/psm/projects/diba/download	81	385	50	1476	0,00%
/psm/projects/bpfm/info	75	347	50	1768	0,00%
/psm/events	105	45	31	148	0,00%
/psm/services	83	45	32	160	0,00%
3 /psm/logout	737	1093	96	159940	1,22%
4 /psm/caslogout	718	1795	53	159825	1,39%
Gesamt	5185	546	18	159940	0,37%

Abbildung 19: Stresstest: Angemeldete User - Antwortzeit und Fehlverhalten

Label	Anz. der Proben	Durchschnitt	Min	Max	% Fehler
casLogin	650	55	32	199	0,00%
LoginForm	641	27	18	77	0,00%
5 /psm/projects/semfis/info/admin?view=home	1758	492	100	8428	0,06%
3 /psm/logout	585	121	92	336	0,00%
4 /psm/caslogout	585	74	57	120	0,00%
Gesamt	4219	245	18	8428	0,02%

Abbildung 20: Stresstest: Administrator - Antwortzeit und Fehlverhalten

Die Abbildungen 18 bis 20 verdeutlichen den Zusammenhang zwischen dem Auftreten eines Fehlers und der sehr hohen Antwortzeit. Das kommt daher, dass der Server von Seiten der CPU so aus- beziehungsweise überlastet ist, dass er die Anfragen nicht mehr verarbeiten kann und es zu Timeouts kommt.

Die Besonderheit bei dem Stresstest: Angemeldete User (Abbildung 19), ist das Verhalten des CAS Servers beim Ausloggen. Dieser stellt das Nadelöhr des Systems dar. Vergleicht man auch hier die Antwortzeit, so erkennt man, dass alle anderen Dienste noch relativ schnell antworten.

Lasttest
Die einzelnen Lasttests der 3 Testserien wurden jeweils für 30 Minuten durchgeführt, um ein aussagekräftiges Testergebnis zu erhalten.

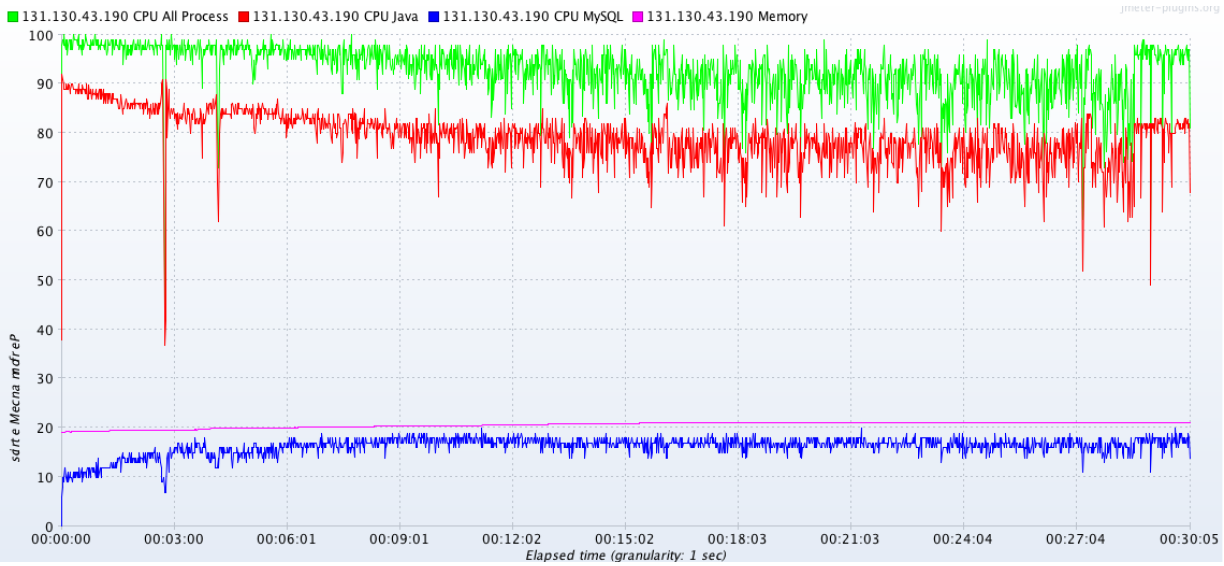

Abbildung 21: Lasttest: CPU und Speicher

Wie in Abbildung 21 zu sehen, kommt der Server gut mit der angelegten Last zurecht. Lediglich am Anfang, beim initialen Hochfahren der einzelnen Threads, kommt die CPU Auslastung in die Nähe der 100 Prozent Marke. Danach pendelt sich die Last ca. im 95 Prozent Bereich ein, mit einem Speicherverbrauch von ca. 20 Prozent.

Label	Anz. der Proben	Durchschn...	Min	Max	% Fehler
35 /psm/	13418	75	23	14670	0,01%
55 /psm/projects	13417	50	27	711	0,00%
casLogin	10550	39	15	286	0,00%
LoginForm	10550	59	19	721	0,00%
224 /psm/projects/istar/info	2702	199	48	621	0,00%
86 /psm/projects/som/download	2634	195	45	598	0,00%
/psm/projects/som	643	245	76	611	0,00%
/psm/projects/bpfm/info	625	194	45	539	0,00%
190 /psm/projects/ec/download	2830	198	48	638	0,00%
5 /psm/projects/semfis/info/admin?view=home	16308	62	32	1281	0,00%
/psm/projects/cv/download	577	199	51	642	0,00%
284 /psm/projects/bpfm/info	2640	203	46	545	0,00%
3 /psm/logout	10548	140	72	14664	0,01%
4 /psm/caslogout	10546	96	36	514	0,00%
252 /psm/projects/cv/download	2593	200	46	628	0,00%
254 /psm/projects/cv/info	2713	195	42	768	0,00%
/psm/projects/cv	587	250	79	714	0,00%
223 /psm/projects/istar	2657	41	17	288	0,00%
94 /psm/about	2758	24	14	311	0,00%
/psm/projects/muviemot/download	605	215	52	584	0,00%
/psm/projects/diba/download	631	197	45	508	0,00%
160 /psm/projects/muviemot/info	2590	217	49	652	0,00%
120 /psm/events/summerschool/info	2665	180	42	510	0,00%
/psm/projects/semfis/info	646	207	49	562	0,00%
/psm/projects/rupert	618	250	77	607	0,00%
/psm/projects/som/download	616	205	48	587	0,00%
/psm/about	571	25	16	167	0,00%
159 /psm/projects/muviemot	2640	42	18	291	0,00%
283 /psm/projects/bpfm	2737	41	17	216	0,00%
99 /psm/events	2700	26	14	284	0,00%
/psm/events/modellierung/info	639	181	48	463	0,00%
128 /psm/services	2702	26	14	262	0,00%
/psm/events	621	32	15	256	0,00%
/psm/events/summerschool/info	586	181	46	500	0,00%
91 /psm/projects/som/info	2687	193	44	567	0,00%
/psm/projects/istar/info	612	205	48	502	0,00%
/psm/projects/semfis	611	243	78	626	0,00%
/psm/	647	80	25	288	0,00%
/psm/projects/som/info	603	206	49	584	0,00%
/psm/projects/cv/info	622	207	48	576	0,00%
/psm/projects	627	61	32	478	0,00%
/psm/projects/ec/info	590	208	47	556	0,00%
/psm/projects/bpfm	657	251	81	594	0,00%
/psm/projects/istar	607	248	75	620	0,00%
/psm/projects/rupert/info	589	203	55	522	0,00%
/psm/services	600	32	15	171	0,00%
/psm/projects/ec	612	244	73	555	0,00%
Gesamt	140927	102	14	14670	0,00%

Abbildung 22: Lasttest: Antwortzeit und Fehlverhalten

Der Lasttest läuft sehr stabil, wie man aus der Abbildung 22 erkennen kann. Es gab lediglich minimale Fehler beim Logout (von 10.548 Proben gab es bei 0,01 Prozent der Durchläufe einen Fehler, was in absoluten Zahlen bedeutet, dass ungefähr 1 Probe fehlerhaft war). Schaut man sich unter der Spalte "Max", den Peak der Antwortzeit von ca. 14.664 Millisekunden an und vergleicht diesen mit dem Durchschnittswert von 140 Millisekunden, so liegt es nahe, dass die fehlerhafte Probe einem kurzfristigen Netzwerkproblem geschuldet ist.

5 Related Work

Bei meiner Recherche zu diesem Paper sind mir zwei Arten von Quellen aufgefallen. Zum einen die allgemein gehaltenen Werke wie zum Beispiel [25]. Diese Art von Quellen liefern einen guten allgemeinen Überblick für das Testen an sich und gehen zumeist auf die technischen Hintergründe und deren Umsetzung von prominenten Testarten wie zum Beispiel Unittest ein, nur am Rande auf Lasttest und kaum im Zusammenhang mit verteilten Systemen wie Microservices.

Die andere Art der Quellen sind spezialisierte Werke, wie es auch dieses Paper darstellt. Allerdings gibt es meines Erachtens keine Werke, welche sich der Fragestellung Lasttest verteilter Microservices im Java Spring Umfeld annehmen. Ein Werk [47] geht überhaupt nur in diese Richtung, spezialisiert sich aber auf Teilaspekte, speziell auf das generelle Testen der einzelnen Komponenten von verteilten Software Systemen. Mein Paper geht darüber hinaus viel detaillierter auf Lasttest von Microservices ein, nicht auf das Testen der einzelnen Komponenten, sondern des gesamten Systems.

6 Conclusion

Die Bachelorarbeit verdeutlicht im einleitenden Theorieteil, wie wichtig es ist, die Software zu testen. Dies sollte den gesamten Entwicklungszyklus begleitend geschehen. Als Resultat erhält man eine Software, welche im Gegensatz zu nicht, beziehungsweise ungenügend getesteter Software, eine signifikant höhere Qualität aufweist, was sich wiederum positiv auf den Geschäftserfolg auswirkt.

Die konkreten Testergebnisse der Fallstudie zeigen auf, dass das Gesamtsystem der Microservices doch sehr performant ist. Immerhin läuft der Server mit einer 2 GHz CPU und 4 GB Arbeitspeicher, was im Vergleich zu aktuellen Serversystemen, mit Octa-Core Mehrkernprozessoren und >32 GB Arbeitsspeicher, doch klein dimensioniert ist.

Sollen mehr User auf die Microservices gleichzeitig zugreifen können, so kann die Performance relativ einfach erhöht werden, indem die Server Hardware, wie oben beschrieben, verbessert und zusätzlich die Ressourceneinstellungen des Tomcat und MySQL Servers angepasst werden. Zusätzlich können auch die einzelnen Microservices, jeweils auf einem eigenen Server, ausgegliedert werden. Hier muss, als potenzielles Nadelöhr, auf die Netzwerkbandbreite der Verbindung untereinander geachtet werden.

Des Weiteren zeigte sich deutlich, dass der CAS Server ein begrenzendes Element, gerade beim Logout Verhalten, für die Anwendung darstellt.

Der CAS Server, speziell der Umgang mit dem dynamischen Ticketsystem, stellte auch beim Testen eine sehr große Herausforderung dar. Apache JMeter kann nicht direkt mit den dynamisch vergebenen Tickets umgehen, bietet aber mit dem Regular Expression Extractor und dem BeanShell PostProcessor zwei mächtige Werkzeuge zur Lösung des Problems.
Zusammenfassend ist zu sagen, dass der Lasttest ein sehr interessantes Instrument ist, um Aussagen über die Anwendung zu treffen. Dies sind nicht nur direkte Aussagen, sondern auch Aussagen über Zusammenhänge einzelner Komponenten. Das Tool der Wahl, in diesem Fall Apache JMeter, ist sehr flexibel, bietet sehr viele Möglichkeiten zur individuellen Umsetzung des Lasttest und ist dabei noch durch Plugins erweiterbar. Des Weiteren wäre es sehr interessant zu untersuchen, wie sich die Anwendung verhält, wenn sich die Server Hardware und die Software, wie oben beschrieben, verändert hat. Gegebenenfalls lässt sich daran erkennen, ob die Anwendung bezüglich steigender Nutzerzahlen skalierbar ist.

Literatur

1. http://www.manager-magazin.de/unternehmen/it/kolumne-bohnet-software-ist-zu-wichtig-um-sie-techies-zu-ueberlassen-a-1011663.html (Abgerufen: 14/09/2015 14:00 Uhr)
2. https://www.qz-online.de/news/uebersicht/nachrichten/risikomanagement-bessere-software-qualitaet-durch-fruehe-tests-986473.html (Abgerufen: 14/09/2015 14:00 Uhr)
3. http://www.heise.de/developer/artikel/Anleitung-fuer-professionelles-Software-Testmanagement-227244.html (Abgerufen: 14/09/2015 14:00 Uhr)
4. Franz, K.: Handbuch zum Testen von Web- und Mobile-Apps : Testverfahren, Werkzeuge, Praxistipps. Berlin, Heidelberg : Springer Berlin Heidelberg, 2 edn. (2015)
5. Hoffmann, D.W.: Software-Qualität. Springer Berlin-Heidelberg (2008)
6. Kleuker, S.: Qualitätssicherung durch Softwaretests : Vorgehensweisen und Werkzeuge zum Test von Java-Programmen. Wiesbaden : Springer Fachmedien Wiesbaden (2013)
7. Pilorget, L.: Testen von Informationssystemen : Integriertes und prozessorientiertes Testen. Vieweg+Teubner Verlag : Wiesbaden (2012)
8. http://junit.org/ (Abgerufen: 12/09/2015 08:00 Uhr)
9. http://cunit.sourceforge.net/ (Abgerufen: 12/09/2015 08:00 Uhr)
10. http://cppunit.sourceforge.net/doc/cvs/index.html (Abgerufen: 12/09/2015 08:00 Uhr)
11. http://www.jsunit.net/ (Abgerufen: 12/09/2015 08:00 Uhr)
12. http://www.nunit.org/ (Abgerufen: 12/09/2015 08:00 Uhr)
13. https://phpunit.de/ (Abgerufen: 12/09/2015 08:00 Uhr)
14. http://www.soapui.org/ (Abgerufen: 12/09/2015 08:00 Uhr)
15. http://www8.hp.com/de/de/software-solutions/loadrunner-load-testing/index.html (Abgerufen: 12/09/2015 08:00 Uhr)
16. http://jmeter.apache.org/ (Abgerufen: 12/09/2015 08:00 Uhr)
17. http://www.seleniumhq.org/ (Abgerufen: 12/09/2015 08:00 Uhr)
18. http://www.intelsol.de/aktivesbr-end-to-end-monitoring/ (Abgerufen: 12/09/2015 08:00 Uhr)
19. http://www.openvas.org/index.de.html (Abgerufen: 12/09/2015 08:00 Uhr)
20. https://codiscope.com/product/secureassist/ (Abgerufen: 12/09/2015 08:00 Uhr)
21. http://www8.hp.com/de/de/software-solutions/webinspect-dynamic-analysis-dast/index.html (Abgerufen: 12/09/2015 08:00 Uhr)
22. http://httpd.apache.org/docs/2.2/programs/ab.html (Abgerufen: 12/09/2015 08:00 Uhr)
23. http://www.red-gate.com/de/products/dotnet-development/ants-performance-profiler/ (Abgerufen: 12/09/2015 08:00 Uhr)
24. DIN40041: Zuverlässigkeit (1990-12)
25. Avci, S.: Evaluieren von automatisierten Tests bei Web-Applikationen. Mag.-arb., Technische Universität Wien (2010)
26. https://developer.ibm.com/bluemix/wp-content/uploads/sites/20/2015/01/microvsmono.png (Abgerufen: 17/09/2015 14:00 Uhr)
27. https://jaxenter.de/microservices-agilitaet-mit-architektur-skalieren-19499 (Abgerufen: 19/09/2015 12:00 Uhr)
28. Conway, M.E.: How do committees invent? Datamation 14(5), 28–31 (April 1968)
29. Schmalenbach, C.: Performancemanagement für serviceorientierte JAVA-Anwendungen : Werkzeug- und Methodenunterstützung im Spannungsfeld von Entwicklung und Betrieb. Berlin, Heidelberg : Springer-Verlag Berlin Heidelberg (2007)
30. http://desmoj.sourceforge.net/home.html (Abgerufen: 12/09/2015 08:00 Uhr)
31. http://grinder.sourceforge.net/ (Abgerufen: 12/09/2015 08:00 Uhr)
32. http://www.clarkware.com/software/JUnitPerf.html (Abgerufen: 12/09/2015 08:00 Uhr)
33. http://ejp.sourceforge.net/ (Abgerufen: 12/09/2015 08:00 Uhr)
34. https://code.google.com/a/eclipselabs.org/p/simpleprofiler/ (Abgerufen: 12/09/2015 08:00 Uhr)
35. http://perfinsp.sourceforge.net/jprof.html (Abgerufen: 12/09/2015 08:00 Uhr)
36. http://sourceforge.net/projects/eclipsecolorer/ (Abgerufen: 12/09/2015 08:00 Uhr)
37. https://github.com/p6spy/p6spy (Abgerufen: 12/09/2015 08:00 Uhr)
38. http://sourceforge.net/projects/jstress/ (Abgerufen: 12/09/2015 08:00 Uhr)
39. http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/index.html (Abgerufen: 12/09/2015 08:00 Uhr)
40. http://www.appperfect.com/products/load-test.html (Abgerufen: 12/09/2015 08:00 Uhr)
41. http://java.quest.com/ (Abgerufen: 12/09/2015 08:00 Uhr)
42. http://www.omilab.org/web/guest/about (Abgerufen: 10/08/2015 08:00 Uhr)
43. Miron, E.T.: Open model laboratory booklet. http://www.omilab.org/web/user/booklet/ (2014)

44. Karagiannis, D., Grossmann, W., Höfferer, P.: Open Model Initiative - A Feasibility Study. Austrian Federal Ministry for Transport, Innovation and Technology, http://www.openmodels.at/publications (2007)
45. http://www.liferay.com/
46. http://www.nngroup.com/articles/website-response-times/ (Abgerufen: 29/08/2015 01:00 Uhr)
47. Thillen, F.: Isolated testing of software components in distributed software systems. Dipl.-arb., Technische Universität Wien (2012)

Eigenständigkeitserklärung

"Hiermit gebe ich die Versicherung ab, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Publikationen entnommen sind, sind als solche kenntlich gemacht. Die Arbeit wurde in gleicher oder ähnlicher Form weder im In- noch im Ausland (einer Beurteilerin/ einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt."

Wien, den 10.10.2015