Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Abschlussarbeit

André Behrens

Testen von Legacy Code in Open Source Projekten

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

André Behrens

Testen von Legacy Code in Open Source Projekten

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuende Prüferin: Frau Prof. Dr. Bettina Buth
Zweitgutachterin: Frau Prof. Dr. Ulrike Steffens

Abgegeben am 19.02.2016

André Behrens

Thema der Arbeit

Testen von Legacy Code in Open Source Projekten

Stichworte

Softwaretest, Legacy Code, Open Source Projekte, ews-java-api, Microsoft Exchange Server, Java

Kurzzusammenfassung

Thema dieser Arbeit ist es Teststrategien im Zusammenhang von Open Source Projekten und Legacy Code aufzuzeigen. Hier werden zunächst Grundlagen des Softwaretests mit Teststrategien in großen Unternehmen verglichen und deren Anwendbarkeit anhand von Analysen und Beispielen auf ein Projekt angewandt, welches große Bestandteile von Legacy Code aufweist.

André Behrens

Title of the paper

Testing Legacy Code in Open Source Projects

Keywords

Softwaretest, Legacy Code, Open Source Projects, ews-java-api, Microsoft Exchange Server, Java

Abstract

This thesis is about test strategies in big firms as well as them to be used in open source projects and will give some introduction on software testing basics in comparison with their applicability with strategies used in big companies. These strategies will then be applied on a single project that contains massive amounts of legacy code.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Viele Open Source Projekte stehen vor der Herausforderung unterschiedlichster Contributoren in dessen Rahmen schnelle, aussagekräftige und automatisierte Softwaretests wichtig sind, um für unterschiedliche Anpassungen einen Standard von Qualität und eine hinreichende Berücksichtigung der Anforderungen zu gewährleisten. Allerdings existieren einige Projekte, die von einer sehr niedrigen allgemeinen Testabdeckung charakterisiert werden.
Dies führt, wie im folgendem am Beispiel der ews-java-api dargestellt werden wird, oftmals zu Problemen, die zugrundeliegende Qualität von Softwareänderung auch im Hinblick auf Regressionstests einschätzen zu können. Diese Arbeit wird sich mit den Charakteristika und Folgen von großen Anteilen von Legacy Code in Softwareprojekten generell und im speziellen in der Verknüpfung mit Open-Source Projekten beschäftigen.
Zusätzlich zur allgemeinen Definition funktionaler als auch technischer Aspekte des Softwaretestens soll im Fokus dieser Arbeit stehen, Fallstricke in Bezug auf die Herausforderungen der Etablierung eines Testkonzepts empirisch zu erörtern und diese anhand von Best-Practices großer Softwarehäuser mit dem aktuellen Design der ews-java-api zu vergleichen.
Mittels einzelner praktischer Beispiele kann somit die derzeitige Architektur der API angepasst werden, um die Voraussetzungen für die Implementierung weiterer Tests zu schaffen und mögliche Problemstellungen von mittelgroßen - großen Projekten aufzuzeigen.
Im Rahmen der Bachelorarbeit wird der Autor ebenfalls einzelne Tools auswählen, die eine Implementierung von Test auf Ebene der Komponenten erleichtern und einen Test ermöglichen, ohne dabei auf die Ebene des Integrationstests vorzudringen. Des Weiteren werden empirisch Kriterien ermittelt, die für das Testen auf den einzelnen Teststufen angesetzt werden können und die Softwaretests gerade im Hinblick auf Qualitätsziele in mittelgroßen – großen Projekten unterstützen.
Hierbei wird ebenfalls der Problematik Rechnung getragen, dass entsprechende Anforderungen an das Beispielprojekt nur grundlegend vorhanden und somit nur in eingeschränkter Form durch Dokumentationen gestützt sind. Die Herausforderung wird sein, auch in diesem Bereich zuerst einzelne Anforderungen zu extrahieren, um anschließend die gewonnenen Informationen im Sinne einer Strategie zur Testfallerstellung nutzbar zu machen.
Gerade im Hinblick auf den zu erwartenden Aufwand steht das Softwareprojekt ews-java-api ebenfalls vor der Herausforderung, die anfallenden Arbeiten möglichst zielgerichtet einzubringen, um die verfügbaren Resourcen, welche zunehmend aus einem Team freiwilliger Entwickler

besteht, möglichst im Rahmen einzelner Milestones einzusetzen, um ebenfalls einer Überforderung vorzubeugen.

1.2 Arbeits- und Testumfeld

Die ews-java-api wird seit dem Jahr 2014 auf der Internetplatform github.com entwickelt. Als webbasierter Filehosting-Dienst für einzelne Softwareprojekte bietet github.com sowohl kostenlose öffentliche Repositories, als auch private und damit zu bezahlende Repositories an. Das für die ews-java-api genutzte Repository wird administrativ von einzelnen Personen von Microsoft Office Developern betreut.
Im Fokus des Projektes steht es den Kostenanteil für die Entwicklung der API auf Seiten Microsofts minimal zu halten. Vor diesem Hintergrund ist die Community angehalten, alle für die Entwicklung der API genutzten Bedarfsmittel möglichst von kostenlosen Anbietern einzuholen. Vertrieben wird die ews-java-api unter der MIT-Lizenz, was eine Anpassung der Software im jeweiligen Bedarfsfall durch andere Entwickler ermöglicht und ebenfalls den Einsatz in einem kommerziellen Umfeld erlaubt.

1.3 Zielsetzung

Zielsetzung dieser Arbeit ist es, mögliche Perspektiven für Projekte aufzuzeigen, die einen Großteil an Legacy Code aufweisen. Diese Projekte haben im Bereich der Anpassung ihres Quelltextes oftmals die Problematik, dass Änderungen nicht oder nur in unzureichendem Maße durch automatisierte oder manuell auszuführende Tests abgesichert werden. Dennoch bieten die vorhandenen Grundlagen zum Softwaretest und Beispiele aus der Praxis die Möglichkeit auch in Projekten, die derartige Charakteristika aufweisen Maßnahmen zu etablieren, die eine solche Entwicklung in positiver Art und Weise beeinflussen.
Anhand einer beispielhaften Anwendung im Bereich der ews-java-api oder anderweitiger Beispiele werden musterhafte Strategien zuerst auf ihre Anwendbarkeit generell beleuchtet und dann praktisch angewendet werden. Im Rahmen eines Ausblicks kann somit darüber resümiert werden, welche Strategien kurzfristig umgesetzt werden konnten und welche ebenfalls langfristig etabliert werden müssen. Dies vor allem vor dem Hintergrund, dass eine gezielte Anhebung der Testabdeckung der API nur mit sehr hohem Aufwand möglich ist.
Dennoch wird diese Arbeit Anknüpfungspunkte aufzeigen, in dessen Rahmen durch Priorisierung einzelner Bereiche der API und Auswertung von spezifischen Anwendungsfällen, Einstiegspunkte für einen Softwaretest angesteuert werden können. Hierbei wird deutlich, dass Testen grundsätzlich nur die Anwesenheit von Fehlerzuständen aufweist, nicht aber die Fehlerfreiheit eines Systems symbolisiert.

1.4 Abgrenzung

Die Anpassung der zur Veranschaulichung vorgesehenen API kann nur im Rahmen der Vorgaben erfolgen, die von Microsoft ebenfalls für die gleichartige ews-managed-api vorgehsehen ist.
Beide Softwareprodukte bieten die Möglichkeit, mittels ihrer Funktionalität einzelne Operationen auf Ebene des Microsoft Exchange-Servers durchzuführen und unterscheiden sich lediglich im Rahmen der zur Implementierung genutzten Programmiersprache.
Gerade im Hinblick auf entstehende Kosten, die sich sowohl monetär, als auch im zeitlichen Rahmen der Einplanung von Personentagen niederschlagen kann diese Arbeit nicht sämtliche erarbeiteten Vorschläge zur Erfüllung der gesetzten Qualitätsziele erfüllen. Die ausgewählten Strategien werden somit beispielhaft anhand einzelner Codeausschnitte zur Anwendung gebracht werden.
In einem ersten Schritt wird sich der Hauptteil dieser Arbeit auf Grundlage der Komponententests bewegen. Im Rahmen einer Meilensteinplanung werden hier zwar ebenfalls weitere Teststufen aufgeführt werden. Diese werden allerdings lediglich zur Veranschaulichung eines Ausblicks der Arbeit aufgeführt und werden somit nur durch den theoretischen Teil dieser Arbeit abgedeckt.

1.5 Gliederung der Arbeit

Die Arbeit wird mit einer Einführung in den Themenbereich allgemeiner Grundlagen des Testens beginnen. Dies dient vor allem als Grundlage für die spätere Anwendung und Erwähnung der beschriebenen Verfahren im Hinblick auf einen möglichen Einsatz im Rahmen der ews-java-api.
Hierzu werden zum einen einzelne Strategien in ausgewählten Unternehmen ausgewählt und vorgestellt werden, um im Nachgang deren Einsatzbarkeit auf die ews-java-api zu überprüfen.

2 Grundlagen des Softwaretests

In diesem Kapitel werden grundlegende Prinzipien des Softwaretests näher dargestellt. Somit wird zum einen die dieser Arbeit zugrundeliegende Definition des Begriffs „Legacy Code“ näher vorgestellt, aber auch auf generelle Vorgehensmodelle und Einordnungen des Begriffs Softwaretest eingegangen werden.

2.1 Definition Legacy Code

Unter der Definition Legacy Code werden umgangssprachlich und auch in Bezug auf die industrielle Wortverwendung Codefragmente verstanden, die in vielerlei Hinsicht schwer oder nur mit vielen Anstrengungen und unter Einsatz von hohem Aufwand anzupassen sind. Ebenfalls bringt Legacy Code oftmals die Schwierigkeit mit sich, dass Bestandteile des Codes durch eine Vielzahl an Änderungen schwer zu verstehen sind.
Der Zustand der Codebasis hat dabei allerdings oftmals keine direkte Verantwortlichkeit zu dem jeweiligen Entwickler, da der Code in Hinblick auf die oben genannte Definition, durch eine Vielzahl an Möglichkeiten zu Legacy Code degradiert werden kann. (Vgl. Feathers 2009) Im Hinblick auf seine Arbeit in Softwareprojekten beschreibt Michael Feathers die Bedeutung von Legacy Code derartig, dass man mit dem Begriff ebenfalls eine simplere Definition verbinden kann und definiert den Begriff als Code, der nicht getestet ist.
“But over years of working with teams, helping them get past serious code problems, I´ve arrived at different definition. To me, legacy code is simply code without tests.” (Feathers 2009, S. XVI)
Eine Kernaussage, die auch im nachfolgenden Teil dieser Arbeit als zugrundeliegende Definition herangezogen und ebenfalls im Rahmen der Arbeit als Definition für den Term angewendet werden wird.

2.2 Fehler / Mängel

Wie in allen von und durch Menschen beeinflussten Bereichen können auch in der Softwareentwicklung Situationen auftreten, die von Anforderungsgebern, dem Benutzer oder dem Entwickler selbst als fehlerbehaftet angesehen werden.
Hierzu formulierte bereits Marcus Tullius Cicero in der fünften seiner Philippischen Reden „Cuiusvis hominis est errare“ – „Jeder Mensch kann irren“. Eine solche Situation kann allerdings nur dann als Fehler definiert werden, wenn bereits im Vorfeld festegelegt wurde, wie sich eine als korrekt einzustufende, also nicht fehlerbehaftete Situation manifestieren soll.

Ein Fehler ist somit die Nichterfüllung einer festgelegten Anforderung, eine Abweichung zwischen dem Istverhalten (während der Ausführung des Tests oder des Betriebs festgestellt) und dem Sollverhalten (in der Spezifikation oder den Anforderungen festgelegt). (Spillner und Linz 2012, S. 7)

Zusätzlich zu einem solchen Fehlerfall kann in einer Software ebenfalls eine Situation eintreten, die als ein Mangel bezeichnet werden kann. Im Vergleich zu einem Fehler klassifiziert der Mangel allerdings die Situation, dass eine an die Software gestellte berechtigte Erwartung nicht, oder in unangemessener Art und Weise erfüllt wurde.
Ebenfalls kann ein Mangel dann entstehen, wenn die gleiche Ausprägung auf eine gestellte Anforderung übertragen werden kann. Beide ausgewiesenen Situationen können innerhalb eines Softwaresystems z.B. dann vorliegen, wenn diese seit der ersten Implementierung vorliegen, oder aber, wenn eine nachträgliche Änderung besagtes abweichendes Verhalten hervorruft. (Spillner und Linz 2012, S. 7)

2.3 Test

Um die in Abschnitt 2.2 beschriebenen Mängel und Fehler aufzudecken bedarf es einer Sammlung an Methodiken, die eine genaue Lokalisierung der fehlerbehafteten Situation im Softwareprodukt ermöglichen, da bei Auftreten des Fehlers generell nur dessen Existenz bekannt ist, dieser aber meist auch im Hinblick auf die Möglichkeit auftretender Folgefehler nicht genau durch bloße Betrachtung der Fehlerwirkung lokalisiert werden kann.
Das Lokalisieren und Beheben des Defekts ist hierbei Aufgabe des Softwareentwicklers und wird auch als Debugging (Fehlerbereinigung, Fehlerkorrektur) bezeichnet.
Die Behebung des Fehlerzustands führt im Idealfall zur Qualitätsverbesserung des Produktes (Spillner und Linz 2012, S. 9). Dieser Themenschwerpunkt bezüglich vollzogener Änderungen und neu entstehender Fehlersituationen im Softwareprodukt wird auch im Bereich des Legacy Codes eine hohe Bedeutung zukommen, thematisch wird dieser Zusammenhang in einem späteren Abschnitt erläutert werden.

Im generellen werden mit dem Testen mehrere Ziele verbunden, die in folgenden Zielvorstellungen zusammengefasst werden können:

- Ausführung des Programms mit dem Ziel, Fehlerwirkungen nachzuweisen
- Ausführung des Programms mit dem Ziel, die Qualität zu bestimmen
- Ausführung des Programms mit dem Ziel, Vertrauen in das Programm zu erhöhen
- Analysieren des Programms oder der Dokumente, um Fehlerwirkungen vorzubeugen

Hierbei ist ganz wichtig, dass generell durch das Testen keine Fehlerfreiheit nachgewiesen werden kann, denn selbst wenn alle ausgeführten Testfälle keinen einzigen Fehler mehr aufdecken, kann (außer in sehr kleinen Programmen) nicht mit völliger Sicherheit ausgeschlossen werden, dass es nicht zusätzliche Testfälle gibt, die weitere Fehlerwirkungen erzeugen und

somit weitere Fehlerzustände in der Software aufzeigen. (Spillner und Linz 2012, S. 9)

2.4 Test-Driven-Development

Das Prinzip von Test-Driven-Development (TDD) kann in einen Aktionsplan von 11 Phasen eingeteilt werden. Diese lassen sich jeweils in 3 größere Kategorien gruppieren. In „Beautiful Testing“ werden die Eigenschaften der jeweiligen Phase und auch dessen Urheber in übersichtlicher Weise dokumentiert.
Das Vorgehen läuft hierbei so ab, dass zuerst für z.B. ein neues Feature ein Testfall erstellt wird, dessen Ausführung fehlschlägt (Rot). Danach kann durch Reproduktion dessen, was durch eine Funktion oder Methode geleistet werden soll, ein Test funktionsfähig werden (Grün). Hierbei gilt die Maßgabe, dass jeweils der kleinstmöglichste Schritt zur Herstellung der Funktionsweise implementiert wird, auch wenn dieser in einer anderen Konstellation nicht funktionieren kann.
Ein simples Beispiel wäre hier die Implementierung einer Methode *increment()*, die keinen Parameter beinhaltet und jeweils einen Integer-Wert zurückgibt, der je nach Anzahl der Methodenaufrufe jeweils inkrementiert wird. Nach dem erstmaligen Aufruf der Methode gibt diese den Wert 1 zurück, nach dem zweiten Aufruf 2 etc..
Eine dem genannten Prinzip folgende Anwendung von TDD würde zuerst einen Test schreiben, der fehlschlägt, aber bereits die Methode aufruft und eine 1 erwartet. Weil die Methode noch nicht implementiert ist, gibt diese aber noch NULL zurück.
Nun würde der Testfall auf das zu erwartende Ergebnis angepasst werden. Folglich wird als Rückgabewert der Methode eine 1 als Integer zurückgegeben und der Test wird erfolgreich durchlaufen. Im Nachgang würde dieser Kreislauf wiederholt werden, bis das zu erwartende Ergebnis für alle zu testenden Bereiche erfolgreich angewendet werden kann. Der somit entstehende Kreislauf ist in folgender Darstellung zusammengefasst:

	Author	**Developer**
Red	1. Create Example	
	2. Execute Example	
		3. Execute Example
		4. Create unit test
		5. Execute unit test
Green		6. Write system code
		7. Execute unit test
		8. Execute example
	9. Execute Example	
Refactor		10. Refactor system code
		11. Execute Example and unit tests

Tabelle 1 Lifecycle TDD

(Riley 2010, S. 182–183)

Unter den Schritten 6 und 10 werden die in obiger Erklärung ausgewiesene iterative Vorgehensweise zur Implementierung des nächsten, einfachsten Schrittes, um den Test in seine Funktionsfähigkeit zu überführen, verstanden.
Wurde Schritt 11 durchlaufen, kann der Entwickler anhand der Vorgaben des Autors den nächsten Testfall ableiten, der den Test wieder in den Fehlerstatus überführt.

2.5 Der Testfall im Testprozess

Anschaulich beschreibt ein Testfall das Ausprobieren einer Software. Möchte man dies präzisieren, kommt man darauf, dass drei Teilschritte erfasst werden müssen: die Vorbedingungen, die Ausführung und die Nachbedingungen. (Kleuker 2013, S. 25)
Hierbei können wir die unterschiedlichen einzelnen Testfälle oder auch Testaktivitäten im Allgemeinen im Rahmen eines Testprozesses derartig darstellen, dass die drei genannten Abschnitte um die Aktivitäten Testplanung und Testabschlusses erweitert werden.
Somit ergibt sich also ein Ablauf, der für einen durchzuführenden Test als charakteristisch angesehen werden kann.

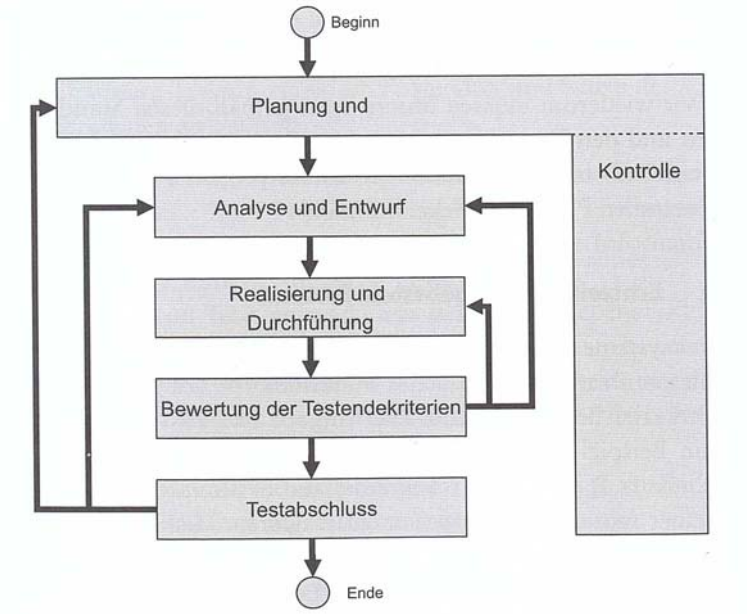

Abbildung 1: Grundstruktur des Testprozesses (Bath und McKay 2010)

In der obigen Abbildung kann sehr gut erkannt werden, dass einzelne Schritte des Testprozesses, die eine Testaktivität beinhalten, aufeinander aufbauend durchgeführt werden können.

Des Weiteren kann bei einer Bewertung der Ergebnisse auch eine Wiederholung des Tests mit zuvor gehender eventueller Anpassung ausgelöst werden.
Hierbei ist es ebenfalls sehr wichtig, die Ergebnisse des Testablaufs in der jeweiligen Stufe einer stetigen Kontrolle zu unterziehen, um eine Entscheidung

darüber fällen zu können, ob die jeweilige Stufe als abgeschlossen betrachtet werden kann. Um ebenfalls für den Test einen Abschluss bilden zu können, sind ebenfalls die Aufstellung von Testendekriterien sehr wichtig, da bei deren Erreichen der Abschluss eines Tests erfolgen kann.

2.6 Teststufen

In Softwareprojekten wird gewöhnlich in vier unterschiedliche Teststufen unterschieden. Diese sind zeitlich und sachlich in abgeschlossene Testabschnitte unterteilt. Die jeweilige Stufe spiegelt gewöhnlich den Implementierungsfortschritt des Projektes wieder. (Vgl. Roitzsch 2005)

2.6.1 Unit-Test / Komponententest

Unit-Tests sind bereits in den 1970er Jahren von Kent Beck als Methodik der Programmiersprache Smalltalk eingeführt worden, um dem Programmierer die Möglichkeit zu geben, in geschriebenen Modulen die zugrundeliegende Codequalität zu messen und diese zu verbessern. Gleichzeitig kann im Sinne einer Dokumentation das Verständnis für die Anforderungen an eine Methode geschärft warden. (Vgl. Osherove 2010)
Ein Unit-Test oder auch Komponententest beschreibt den auf der untersten Ebene der Teststufen befindliches Testszenario dessen Rahmenbedingungen vorgeben, dass jede Komponente einzeln und eigenständig getestet wird.
Dies geschieht vor allem auf der Grundlage, dass auftretende Fehlerwirkungen anderer, aus abhängigen Komponenten resultierenden Codefragmenten während des Tests ausgeschlossen werden sollen, um im Fehlerfall eine möglichst genaue Lokalisation des Fehlers zu erwirken.

2.6.2 Integrationstest

Im klassischen V-Modell schließt sich der Integrationstest an einen vorhergehenden Komponententest an und setzt voraus, dass die dem Integrationstest zugrundeliegenden Testobjekte bereits einen Komponententest durchlaufen haben.
Um die Aussagekraft des Integrationstests noch stärker zu gewichten sollten die im Komponententest aufgezeigten Fehler ebenfalls bereits behoben sein, da sonst ggf. Aspekte in der Integration aufgrund von Folgefehlern nicht eingehend getestet werden können.

Der Integrationstest beschreibt also nach dem Test der einzelnen Komponenten einen Test des Zusammenspiels der einzelnen Komponenten und der damit verbundenen Schnittstellen, eben der Integration der jeweilig einzelnen Komponenten. (Spillner und Linz 2012, S. 52–55)
Die Testziele des Integrationstests lassen sich also wie folgt konkretisieren:

- Eine Komponente übermittelt keine oder syntaktisch falsche Daten, sodass die empfangenen Komponenten nicht arbeiten kann oder abstürzt (funktionaler Fehler einer Komponente, inkompatible Schnittstellenformate, Protokollfehler).
- Die Kommunikation funktioniert, aber die beteiligten Komponenten interpretieren die übergebenen Daten unterschiedlich (funktionaler

Fehler einer Komponente, widersprüchliche oder fehlinterpretierte Spezifikationen).
- Die Daten werden richtig übergeben, aber zum falschen oder verspäteten Zeitpunkt (Timing-Problem) oder in zu kurzen Zeitintervallen (Durchsatz-, Kapazitäts- oder Lastproblem)

(Spillner und Linz 2012, S. 56)

2.6.3 Systemtest

Als nachfolgende aufbauende Stufe auf den Integrationstest erfolgt der Systemtest. Im Vergleich zu den darunterliegenden Teststufen bezieht sich der Systemtest allerdings nicht auf eine technische Spezifikationsprüfung, sondern auf einen Test, der aus Sicht des Kunden bzw. des späteren Anwenders durchgeführt wird.
Der Test ist deshalb im Regelfall notwendig, da viele Problemstellungen und Anforderungen erst unter einer Betrachtung des Gesamtsystems auffallen.
Als Basis dieser Teststufe können alle Dokumente und Informationsinhalte dienen, die eine Anwendung auf der Systemebene beschreiben.

Bei der Durchführung eines Systemtests sollte darauf geachtet werden, dass dieser in einer jeweiligen Testumgebung stattfindet, die der letztendlichen Produktivumgebung in Bezug auf z.B. Hardwareausstattung, Systemsoftware, Treiber, Fremdsystemen und Infrastrukturelementen möglichst nahekommt. Ebenfalls wird im Systemtest zur Prüfung der System- und Anwenderdokumentation auch eine Prüfung von Konfigurationseinstellungen speziell in Form von Last- und Performancetests durchgeführt.

Eine mögliche Problembehaftung, die aus der Definition des Systemtests entsteht, beschreibt den Zustand, wenn bezüglich der zugrundeliegenden Spezifikationen des Systems keine dokumentierten Anforderungen vorliegen und diese also nach dem jeweiligen Gefühl des testenden Anwenders oder der am Projekt beteiligten Personen entschieden werden.
Hieraus kann sich die Problematik ergeben, dass unterschiedliche Sachverhalte von einzelnen Personen jeweils unterschiedlich interpretiert werden und ebenfalls mit individuellen Meinungen verknüpft sind.
Der Tester erhält an dieser Stelle dann die Notwendigkeit, die für den Sachverhalt notwendigen Informationen nachträglich zusammenzutragen und mit den jeweiligen Stakeholdern in Kontakt zu treten.
(Spillner und Linz 2012, S. 60–64)

2.6.4 Abnahmetest

In allen den Abnahmetest zuvorkommenden Testaktivitäten liegt die Verantwortung des Tests in den Händen von den am Softwareprojekt direkt beteiligten Personen, wie Entwickler, Stakeholder oder sonstigen Projektbeteiligten.
Im Rahmen des Abnahmetests kann nun der direkte Endkunde bzw. Benutzer in den Testprozess mit einbezogen werden. Der Test erfolgt hierbei auf Basis von vertraglicher Akzeptanz, der jeweiligen Benutzerakzeptanz, Akzeptanz der Systembetreiber, sowie im Rahmen eines Feldtests.

Um im Rahmen des Abnahmetests auf Bestandteile der vertraglichen Akzeptanz zurückgreifen zu können, sollten im Vorfeld vertragliche Rahmenvereinbarungen getroffen worden sein, die eine möglichst dedizierte Aussage über den gewünschten Funktionsumfang der Anwendungssoftware ermöglichen.
Je nach Umfang der Risikoabhängigkeit kann die Form des Akzeptanztests in unterschiedlicher Gewichtung und Intensität stattfinden.
Musterhafte Beispiele für Abnahmetests unterschiedlicher Intensität könnten somit z.B. die Einführung eines Standardsoftwarepakets oder im Gegensatz dazu eine umfangreiche Individualsoftware mit Schnittstellen zu anderen Komponenten sein, die ebenfalls im Rahmen des Abnahmetests geprüft werden sollten.

Im Rahmen des Tests auf die Akzeptanz des Benutzers stehen die verschiedenen Anwendergruppen mit ihren unterschiedlichen Erwartungen an die Software im Vordergrund.
Hier geht es vor allem darum, ob das System von allen Benutzern zur Benutzung akzeptiert oder abgelehnt wird, was für eine spätere Produktionseinführung der Komponente erhebliche Nachteile mit sich bringen kann und es besteht ebenfalls die Möglichkeit durch Prototypen bereits frühzeitig Feedback einzuholen, um nicht erst vor dem Schritt der Einführung mögliche Diskrepanzen der Anwender in den Entwicklungsprozess aufzugreifen.

In Hinblick auf die Akzeptanz der Systembetreiber bietet der Abnahmetest die Möglichkeit ebenfalls ein Feedback der Systemadministratoren einzuholen und zu evaluieren, wie sich das System in der vorliegenden IT-Infrastruktur verhält. Ebenfalls ein möglicher Bestandteil des Abnahmetests ist die Durchführung eines Feldtests.
Dies kann vor allem dadurch begründet sein, dass die implementierte Komponente auf unterschiedlichsten Geräten eingesetzt wird und die Bereitstellung dieser im Rahmen einer jeweils isolierten Testumgebung als zu aufwendig oder kostenintensiv gilt.
Hier kann also durch Einbeziehung der Endanwender ein Alpha- oder Beta-Test durchgeführt werden, bei dem eine Vorabversion entweder im Rahmen eines Alpha-Tests beim Hersteller zum Test angeboten oder einem repräsentativen Anwenderkreis im Zuge des Beta-Tests zur Benutzung angeboten wird.

Um die Endanwender nicht durch eine Vielzahl an möglichen Instabilitäten zu überfordern, sollte darauf geachtet werden, im Vorfeld die bereits angesprochenen Artefakte des Abnahmetests in angemessener Art durchlaufen zu haben.
(Spillner und Linz 2012, S. 54–57)

2.7 Dynamisches Testen

Das dynamische Testen umschreibt den Testprozess, bei dem das entsprechende Testobjekt auf einem Rechner ausgeführt wird.
Zur systematischen Erstellung einzelner Tests gibt es eine Reihe von unterschiedlichen Verfahren, welche im Bereich des dynamischen Testens als nützliche Instrumente zur Anwendung kommen können.
Meist lassen sich diese Methodiken in zwei unterschiedliche Kategorien einteilen. Die Black- und White-Box Testverfahren.
Beide verfolgen in ihrer Zieldefinition die Erstellung von Tests und gelten deshalb auch als Testfallentwurfsverfahren.
(Spillner und Linz 2012, S. 112)

2.7.1 Black-Box Test

Im Bereich des Black-Box Testverfahrens ist über die interne Verarbeitungslogik des Testobjekts nichts bekannt. Es wird somit als Black-Box angesehen.
Dem Tester sind lediglich die Eingabedaten, das tatsächliche und das zu erwartende Ergebnis bekannt. In diesem Zusammenhang wird auch vom point of observation gesprochen, welcher beim Black-Box Testverfahren außerhalb des Testobjekts liegt.
Ebenfalls sind Beeinflussungen des Testobjekts nur über Vorbedingungen und die eigentlichen Eingabedaten möglich. Der sogenannte Point of Control liegt also ebenfalls außerhalb des jeweiligen Testobjekts.
Da die Verarbeitungslogik nicht bekannt ist, dienen zur Ermittlung der Testfälle meist formale Spezifikationen und Anforderungen an das System zur Gestaltung der zu erwartenden Ausgabewerte und zur Testfallerstellung.
Das Black-Box Testverfahren wird somit auch als funktionales oder spezifikationsorientiertes Testverfahren bezeichnet.
Black-Box Verfahren eignen sich für die Testfallerstellung meist auf einer höheren Ebene als White-Box Testverfahren.
(Spillner und Linz 2012, S. 112–113)

2.7.2 White-Box Test

Im Vergleich zum Black-Box Testverfahren liegt beim White-Box Testverfahren der point of observation innerhalb der zu testenden Anwendung. Somit wird also während der Testausführung der innere Programmablauf analysiert.
Ím Bereich des negativen Tests zum Beispiel kann der point of control ebenfalls innerhalb des Testobjektes liegen und somit die Ablauflogik des Tests während der Laufzeit beeinflussen.

Das Whitebox Testverfahren wird ebenfalls als strukturelles Testverfahren bezeichnet, da die Struktur des Testobjekts bei der Ermittlung der Testfälle mitberücksichtigt werden kann und ebenfalls die jeweilige Programmlogik zur Testfallerstellung herangezogen werden kann.
Einzuordnen sind Testfälle des White-Box Testverfahrens auf der untersten Ebene der jeweiligen Teststufen (Komponenten- und Integrationstests). (Spillner und Linz 2012, S. 112–113)

2.8 Refactoring

Auch das Refactoring hat in der täglichen Arbeit mit legacy Code eine weitreichende Bedeutung. Unter dem Schlagwort Refactoring versteht man den Prozess der Anpassung eines bereits bestehenden Systems in der Weise, dass das äußerliche Verhalten der Komponente oder des Codes nicht verändert wird. Wohl aber Verbesserungen oder Optimierungen an der internen Struktur der Codebasis durchgeführt werden.
Dies kann zum einen dann kritisch sein, wenn Codebestandteile als Legacy Code deshalb vorliegen, weil der Code nur sehr schwer verständlich ist.
Hier kann der Programmierer leicht in die Gefahr geraten, im Rahmen einer Optimierung neue Fehlerfälle zu etablieren.
Des Weiteren kann ein Refactoring dann problematisch sein, wenn durch fehlende Tests keine Aussage über die nachträgliche Funktion des Codes gemacht werden kann. Die angesprochene Evaluierung wäre z.B. durch die Ausführung von Regressionstests möglich.

2.9 Testautomatisierung

Die Testautomatisierung kann gerade im Hinblick auf Open-Source Projekte eine gewichtige Rolle einnehmen, da neben wiederkehrenden contributions grundlegende Testfälle wiederholt durchlaufen werden können.
Hierbei sollte in genereller Form evaluiert werden, welche Tests sich für die Inklusion in einen automatisierten Testprozess eignen, und welche Tests ggf. manuell durchgeführt werden müssen.
Heutige Continious Integration Server ermöglichen es Entwicklern mit relativ wenig Aufwand automatisierte Tests in den jeweiligen Releaseprozess oder Integrationsprozess einzubinden und somit z.B. nach jedem commit automatisiert durchzuführen.
Allerdings eignen sich nicht alle Tests dazu automatisiert durchgeführt zu werden, da dies damit verbunden sein kann, dass die Anpassung der Tests an eine sich schnell ändernde Software mit einem zeitlich hohen Aufwand und damit mit hohen Kosten verbunden sein kann.
Die Effizienz der Automatisierung muss somit bestimmt werden. „Es wird mit großer Wahrscheinlichkeit eine Zeitverschwendung sein, Software zu automatisieren, die sich schnell verändert, da die Wartung der Automatisierungsskripte teuer und zeitaufwendig sein wird.“
(Bath und McKay 2010, S. 343)

In der folgenden Grafik wird dargestellt, wie die Gesamtheit der Testfälle in unterschiedliche Kategorien unterteilt werden kann:

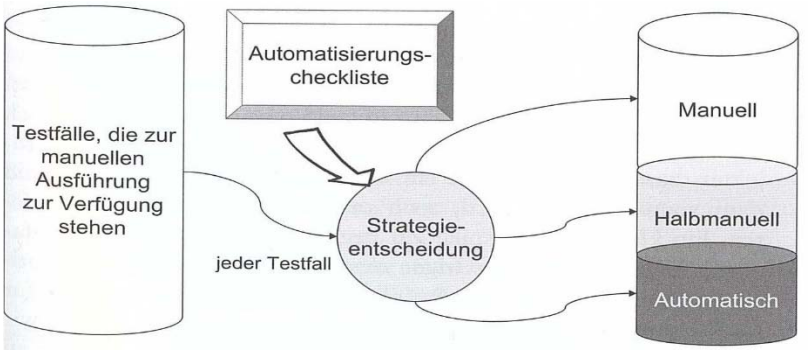

Abbildung 2: Auswahl von Testfällen zur Automatisierung (Bath und McKay 2010, S. 343)

In Bezug auf die Arbeit mit Legacy Code kann dies für etwaige Softwareprojekte bedeuten, dass die Überlegungen der Automatisierung bereits in der Definitionsphase der Test- und Qualitätsziele berücksichtigt werden sollte.
Hier spielt ebenfalls die Ausführungszeit des jeweiligen Tests eine erhebliche Rolle, um den Integrationsprozess und die Ausführungsdauer einzelner Test-Suiten nicht unnötig zu verlängern.

2.10 Benutzung von Test Doubles

Im Rahmen des Abschnitts Testautomatisierung wurden nun einige Prinzipien über die Einpflege von Testfällen in den Entwicklungsprozess dargestellt. Oftmals steht ein Entwickler jedoch vor der Herausforderung, dass in großen Softwareprojekten eine breite Masse an Objekten und Methoden bestehen, die in Ihrer Gesamtheit das System darstellen.
Hier kann es beim Testen von Vorteil sein, Abhängigkeiten aufgrund der Komplexität aufzubrechen und den Test einzelner Methoden auf einer modularen Basis fortzuführen.
Auch für die ews-java-api wird im späteren Verlauf dieser Arbeit darauf eingegangen werden, dass es eine Vielzahl an Stellen geben kann, an denen, um einen Modultest durchführen zu können, auf Techniken zurückgegriffen werden kann, die eine Zielerfüllung mit relativ einfachen Mitteln unterstützen. Existieren also in einem Projekt starke Abhängigkeiten zwischen einem Modul A und einem Modul B, können z.B. für einen Test im Modul A Verweise auf das Modul B unter Einbeziehung der im Nachgang definierten Objekte in ein eigens für den Test erstelltes Objekt umgeleitet werden.
Hier ist es dann z.B. möglich die korrekte Benutzung im Sinne eines Aufrufs der Methoden zu testen oder eigene Rückgabewerte einzelner Methoden zu deklarieren. Gerald Meszaros hat hierzu einige unterschiedliche Arten von Objekten charakterisiert, die im jeweiligen spezifischen Einsatzfall einen Test vereinfachen können und die für den Entwickler bereitstehenden Werkzeuge erweitern.

2.10.1 Stub Objekte

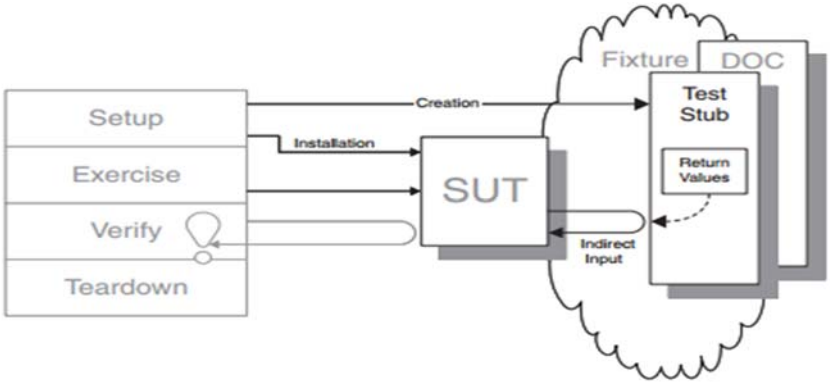

Figure 1: Test Stub zur Erzeugung eines indirekten Inputs (Meszaros 2012, S. 129)

Stub Objekte bieten als Kontrollpunkt die Möglichkeit im Rahmen von Fixtures das zu testende System mit indirektem Input zu versorgen.
Dies kann vor allem dann hilfreich sein, wenn die reale Komponente zu Testzwecken nicht verwendet werden kann oder die für den Test nötigen Inputs nur in besonderen oder schwer zu erzielenden Fällen gewährleistet werden. Das erzeugte Test Stub kann somit während der Ausführung des SUT die vorhergehend definierten Werte zurückgeben.
Die Benutzung von Test Stubs ist allerdings an die Bedingung gebunden, dass der Aufruf durch das SUT an die entsprechenden Stubs umgeleitet wird.
Als mögliche Gründe zur Nutzung eines Stubs kommen folgende Szenarien in Frage:

- die reale Komponente steht zum Zeitpunkt der Testerstellung noch nicht zur Verfügung
- die reale Komponente kann nicht dahingehend manipuliert werden, dass der erwünschte indirekte Input erzielt wird und nur ein wirklicher Softwarefehler kann die Reproduktion des Inputs bewirken.
- die reale Komponente kann manipuliert werden, die Manipulation ist aber sehr aufwändig.
- die Manipulation der realen Komponente hat weiterführende Seiteneffekte

(Meszaros 2012, S. 128–129)

2.10.2 Mock Objekte

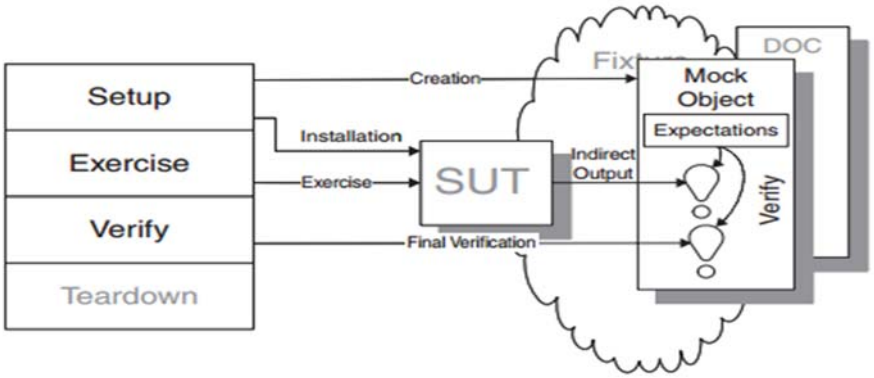

Figure 2: Verifizierung von Verhalten (Meszaros 2012, S. 544)

Die Benutzung von Mock Objekten ermöglicht die Verifikation von erwartetem Verhalten des zugrundeliegenden SUT. Dies kann zum einen dann nützlich sein, wenn das Verhalten des SUT stark durch den Kontext beeinflusst wird, in dessen Rahmen die Ausführung erfolgt.
Ein weiterer Grund könnte sein, dass ein erwartetes Verhalten des SUT verifiziert werden soll. (Meszaros 2012, S. 544–550)
In der Hinsicht kann die Einführung von Mock Objekten also eine gute Möglichkeit sein, den indirekten Output, wie in der obigen Abbildung dargestellt, des SUT abzugreifen und mit den erwarteten Werten des Testers zu verifizieren. Ein mögliches Einsatzszenario könnte z.B. ein SUT sein, welches erzeugte Daten an ein anderes Objekt weitergibt.
Hier kann im Rahmen eines Tests das entsprechende Verhalten validiert und die korrekte Datenverarbeitung getestet werden.

2.10.3 Fake Objekte

Fake Objekte sind Objekte, deren Inhalt grundlegend die gleiche Beschaffenheit aufweist, wie dies auch das in Produktion eingesetzte Objekt haben würde.
Das Fake Objekt bietet für die Qualitätskriterien des Tests allerdings die Möglichkeit, die Verarbeitungsgeschwindigkeit des Objektes zu erhöhen, indem auf Komplexität bei der Erstellung verzichtet wird. Der Einsatz eines Fake Objekts eignet sich aus diesem Grund nicht für den produktiven Einsatz.
Ein mögliches Szenario wäre anstatt der Verwendung einer persistenten Datenbank der Fake mittels eines nicht persistenten Speichers. (Fowler 2006)

2.10.4 Test Spy

Ein Test Spy weist in seiner Art und Weise grundlegend die gleichen Eigenschaften wie ein Stub Objekt auf. Mit dem Unterschied, dass etwaige indirekte Outputs des SUT vom Test Spy gesammelt und zur nachträglichen Auswertung durch den Test bereitgestellt werden.
Im Gegensatz zum Stub wird die Verifizierung also nicht explizit durch eine Logik des Fixtures übernommen, sondern in der Verifizierungsphase des Tests durchgeführt. (Meszaros 2012, S. 538–543)

2.10.5 Dummy Objekte

Ein Dummy Objekt sind Objekte, die im Rahmen des aktuellen Tests z.B. als Parameter für einen Konstruktoraufruf angewendet werden, für den eigentlichen Testfall als solches aber keine Rolle spielen.
Das Dummy Objekt verfügt also über keinerlei Daten und wird für die Testauswertung ebenfalls nicht berücksichtigt.
Es unterstützt den Verwender aber dahingehend, dass zugrundeliegende SUT in den richtigen Initialisierungszustand zu bringen, um einen Test durchführen zu können.
Die Voraussetzung für die Benutzung eines Test Dummys ist, dass die Initialisierung des Objekts in einfacher Art und Weise durchgeführt werden kann. „We create an instance of some object that can be instantiated easily and with no dependencies; we then pass that instance as the argument of the method of the SUT.“ (Meszaros 2012, S. 728)

3 Beschreibung ews-java-api

Grundlage für die weitere systematische Beschreibung und Analyse ist die ursprünglich von Microsoft entwickelte Exchange Webservice API (im Folgenden als ews-java-api bezeichnet).
Die API wird derzeitig auf Github entwickelt und einzelne Releases können mittels Sonatype Nexus z.B. per maven oder gradle in etwaige Projekte integriert werden. Zum Zeitpunkt der Erstellung dieser Arbeit ist die aktuelle Release-Version die API Version 2.0 und es haben insgesamt 209 Personen den Codestand per fork in ein privates repository überführt.
Innerhalb eines Tages wird die Internetseite der API auf Github von 30 bis 250 eindeutigen Benutzern besucht. Die ursprüngliche Codebasis der ews-java-api wurde von Microsoft im Juni 2014 bereitgestellt. (Github - ews-java-api commits 2014)

Zu diesem Zeitpunkt wurde das Projekt als 1:1 Portation der gleichbedeutenden ews-managed-api, also des entsprechenden Pendants, welches in C# entwickelt wurde, released. Zum Zeitpunkt der Überführung umfasste die ews-java-api keine Tests und jegliche Veränderungen wurden z.B. ohne Bereitstellung von Testfällen durchgeführt.
Um den Bereich der Qualitätssicherung auszubauen, wurde durch das Projektteam als erste Maßnahme beschlossen, dass bei jeder Änderung entsprechende Tests ergänzt werden müssen. Dies bezieht sich im Regelfall auf Komponententests.

Weitere Milestones und Ziele für die Zukunft werden in diesem Kapitel ebenso verdeutlicht werden, wie die Ausgangslage des Softwareprojekts zum gegenwärtigen Zeitpunkt.

3.1 Metriken

Die Einhaltung und Evaluierung von Metriken im Rahmen der CI-Builds ist in den meisten bekannten Projekten auf Open-Source Platformen wie github gängige Praxis.
Diese veranschaulichen zum einen z.B. mittels der Evaluierung des Codes mit Checkstyle, ob der neu erstellte oder auch bereits bestehende Code den beschriebenen Anforderungen im Hinblick auf Codekonventionen entspricht. Zum anderen ist es in diesem Zusammenhang aber ebenfalls möglich, potenzielle Schwachstellen oder Mängel in dem jeweiligen Code z.B. mittels statischer Codeanalyse durch Findbugs oder PMD offenzulegen.
Gerade in der Arbeit im Open Source Bereich kann die automatisierte statische Analyse von echtem Vorteil sein, da individuelle Contributions bereits vorab analysiert werden können.

Im Rahmen eines Reviews kann dann gezielter auf den eigentlichen Code eingegangen oder die Ergebnisse der Analyse mit zu einer Diskussion herangezogen werden.

Ebenfalls kann anhand der automatisch und durch den jeweiligen Continuous-Integration-Server erstellten Metriken ein Dokumentationsverlauf erkannt werden, anhand dessen ebenfalls Rückschlüsse auf den Testprozess abgeleitet werden können.
Die weiter oben genannten Tools können hierbei dabei unterstützen, gängige Programmierfehler auch ohne ein manuelles Review zu vermeiden und so häufigen Fehlerquellen vorzubeugen.
Die ausgewählten Tools stehen dem Projekt bereits zum aktuellen Zeitpunkt zur Verfügung. Natürlich können diese allerdings keine Aussage über die generelle Fehleranfälligkeit (im Sinne funktionaler Fehler) des Codes geben, da eine Evaluierung nur im Sinne einer statischen Analyse vollzogen wird.
In Bezug auf eine Transparenz ist aber die Möglichkeit zu gewährleisten, dass alle Projektteilnehmer zu jeder Zeit auf die evaluierten Ergebnisse zugreifen können, um in Ihrer Tätigkeit der Programmierung unterstützt zu werden.

Im Sinne objektiver Testmetriken können zur Überwachung und Erfolgskontrolle ebenfalls folgende Ansätze herangezogen werden:

- Fehlerbasierte Metriken: Anzahl gefundener Fehlerzustände bzw. Fehlermeldungen im jeweiligen Testobjekt pro Release
- Testfallbasierte Metriken: Anzahl der Testfälle in einem bestimmten Status (z.B. geplant, blockiert, fehlerbehaftet, fehlerfrei)
- Testobjektbasierte Metriken: Codeüberdeckung, Dialogüberdeckung usw.
- Kostenbasierte Metriken: aufgelaufene Testkosten, Kosten des nächsten Testzyklusses in Relation zum erwarteten Nutzen

(Spillner und Linz 2012, S. 194–195)
Die oben erwähnten Ansätze 1-3 werden aktuell im Rahmen eines kontinuierlichen builds bereits ermittelt, jedoch nur sporadisch und manuell ausgewertet. Da die API als Open-Source Projekt entwickelt wird, werden Kosten z.B. im Sinne einer Aufwandskalkulation nicht berücksichtigt und auch nicht festgehalten.

3.2 Github.com als Java-Entwicklungsplattform

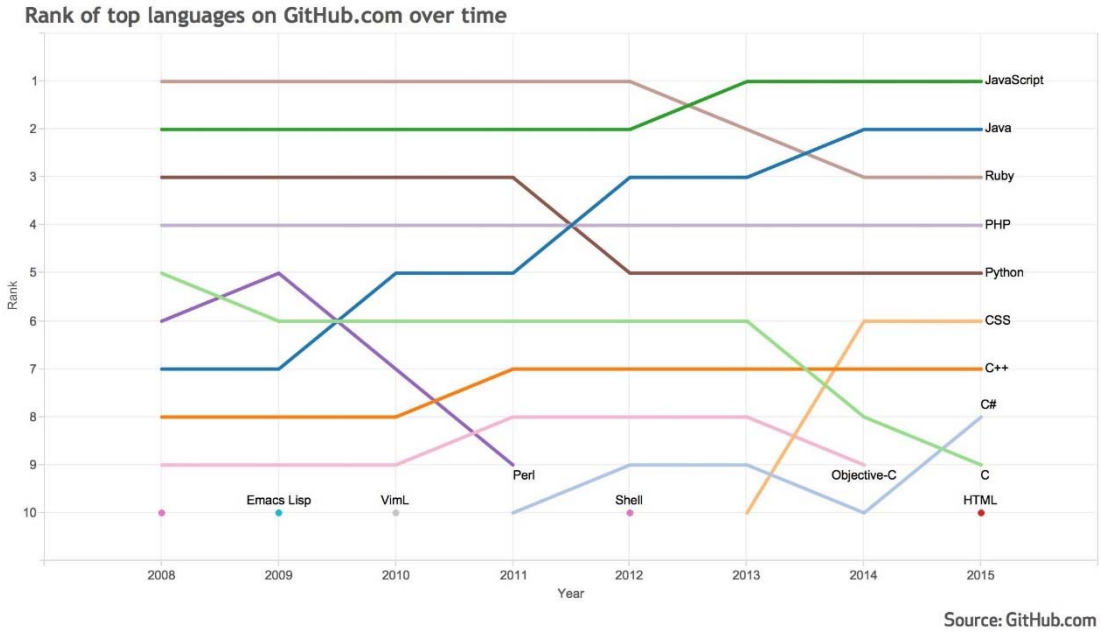

Abbildung 3: Rank of top languages on Github.com over time (github.com 2015)

Git und die damit verbundene Entwicklungsplatform github.com haben sich nicht zuletzt nach der Bekanntgabe der Auflösung von google-code zu einer der am meist benutzten Filehosting-Dienste für Software-Entwicklungsprojekte durchgesetzt.
Viele bekannte Projekte sind ebenfalls in diesem Umfeld vertreten und nutzen github zur Interaktion mit Contributoren und Benutzern der einzelnen Softwareprojekte. Hierzu zählen beispielhaft Bestandteile des Frameworks Spring oder Softwareprojekte der Apache Software Foundation, aber auch Programmiersprachen wie Groovy, scala, ruby oder julia sind hier vertreten. Tendenziell lässt sich aus einer Analyse der auf github verwendeten Programmiersprachen ebenfalls feststellen, dass die Zahl der in Java geschriebenen Projekte von Jahr zu Jahr ansteigend ist (siehe einleitend aufgeführte Grafik).
Wie aus der einleitend aufgeführten Grafik ersichtlich, bildet Java als Programmiersprache nach JavaScript einen erheblichen Anteil der auf github.com gehosteten Softwareprojekte, wobei die Tendenz im Laufe der Jahre als zunehmend beschrieben werden kann.

3.3 Verteilte Versionsverwaltung mit Git

Der folgende Abschnitt beschreibt einige Prinzipien, wie geleistete Entwicklungsarbeit am Code innerhalb eines Teams transparent und für alle zugänglich verwalten werden kann. Dies ebenfalls deshalb, weil in der Organisation größerer Softwareprojekte und in der Kollaboration von mehreren Entwicklern in einem Projektteam ein stetiger Zugriff auf die zu erweiternde Codebasis notwendig ist.
Die Verwaltung des Codes erfolgt hierbei über einzelne Repositories, die zum einen zentral (auf einem Server) und ebenfalls lokal zur Unterstützung der Arbeit des jeweiligen Entwicklers, auf dem Entwicklungsrechner vorliegen.

Zu den sehr verbreiteten Repräsentanten verteilter Versionsverwaltungen zählen Mercurial, Subversion und git. Hierbei sind die Kandidaten Subversion und Mercurial für die Arbeit mit dem webbasierten Filehosting-Dienst github.com zu vernachlässigen.

3.3.1 Git Branching Worflow

Im Bereich der Open-Source Entwicklung mit Git haben sich einige mögliche Modelle herausgestellt, die eine Handhabung von Releases und aktuellen Entwicklungen organisieren.
Ein möglicher Zyklus, der durch Atlassian im Rahmen einer übersichtlichen Darstellung zusammengefasst wurde sieht vor, dass pro jeweiligem Feature, welches losgelöst von der aktuellen Entwicklung weitergeführt werden kann ein eigener Branch erzeugt wird.

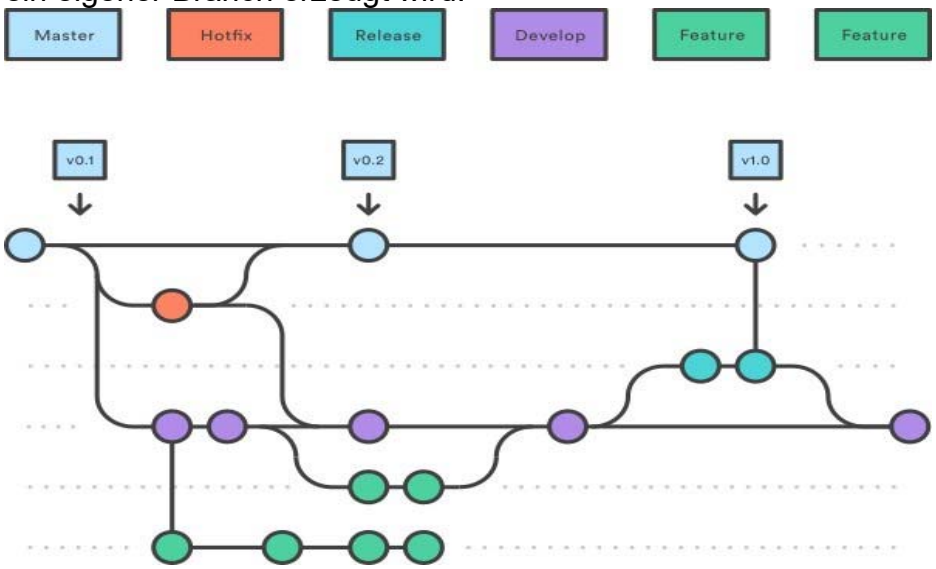

Abbildung 4: gitflow workflow complete (atlassian)

Zusätzlich zu den jeweiligen Features, die in der obigen Grafik aufgeführt sind, werden ebenfalls ein Master-Branch, ein hotfix branch, ein release branch und ein development branch gepflegt.
Jeder erzeugte branch veranschaulicht im Rahmen des Atlassian Git Branching Workflows eine eigene Rolle, die sich auf die einzelnen branches wie folgt anwenden lässt:

Master:
Auf dem master branch werden jeweils offizielle Release Versionen gemerget. Dies kann zum einen, wie in der Grafik veranschaulicht durch einen Hotfix der bereits releasten Version geschehen, oder aber durch eine Entwicklung auf dem Develop Branch.

Hotfix:
Mit Hilfe des hotfix branches können z.B. kritische Fehlerbehebungen direkt aus dem bereits releasten Stand behoben werden.
Dies kann zum einen dann notwendig sein, wenn eine Fehlerbehebung möglichst schnell ausgerollt werden soll und die Behebung nicht erst in ein

kommendes Release integriert werden soll. Der resultierende Zweig des Hotfixes wird direkt vom master geforked und anschließend wieder zurückgeführt.

Release:
Der release branch bietet die Möglichkeit Vorbereitungen für ein neues Release zu starten, auch wenn andere Projektteilnehmer noch an einem feature oder einer Fehlerbehebung arbeiten.
Somit können einzelne commits bei Bedarf in den release branch gemerged werden und nach erfolgter Ansammlung genügender Änderungen kann das Release per Überführung als neue Version in dem master branch erzeugt werden.

Develop:
Der develop branch dient neben den bereits vorgestellten branches dazu, einzelne Features an einer zentralen Stelle zu integrieren. Der develop branch dient ebenfalls dem Zweck, als ursprünglicher Zweig für neue feature branches zu fungieren.

Feature:
Jedes neue Feature sollte im Sinne einer Entkopplung von anderen Änderungen in einem neuen branch entwickelt werden.
Als Basis für ein neues Feature dient der develop branch. Nachdem ein Feature erfolgreich entwickelt wurde, sollten die jeweiligen Änderungen nicht direkt in den master branch integriert werden, sondern, um den Release Prozess zu unterstützen den Weg über den development branch -> release branch -> master branch in eine neue Version finden.

3.3.2 Favorisierter Workflow

Die ews-java-api nutzt zum Zeitpunkt der Erstellung dieser Arbeit eine abgewandelte Form des im Abschnitt „Git Branching Worflow“ vorgestellten Prinzips.
Dies ist zum einen darauf zurückzuführen, dass sich im fortschreitenden Projektverlauf gezeigt hat, dass einzelne contributor der API mit dem beschriebenen Verlauf nur unzulänglich bekannt sind und intuitiv feature branches direkt als pull request auf den master branch erstellt werden.
Zum anderen besteht das für die API zuständige Projektteam aktuell aus zwei Personen. Aufgrund dessen die Möglichkeit der Implementierung einzelner Features nur eingeschränkt gegeben ist, da zunehmend administrative Tätigkeiten in dem Projektteam angesiedelt sind.
So resultieren eine Vielzahl der geleisteten Features und Fehlerbehebungen direkt aus der Community und von Individualpersonen.
Ein weiterer Grund für die Abweichung ist, dass der für die direkte Projektleitung verantwortliche Vertreter von Microsoft den für die ews-java-api angewendeten Releasezyklus bereits im Rahmen anderer Projekte durchgeführt hat.

Somit wurde der bereits angesprochene Releasezyklus den in der Projektarbeit zugrundeliegenden Rahmenbedingungen angepasst und verfolgt folgende Ziele:

- Einfachheit / Verständlichkeit
- Intuitive Handhabung

Das Repository der ews-java-api verfügt über folgende branches, die eine gezielte Weiterentwicklung der API unterstützen sollen:

- master
- individueller branch für eine Release Version z.B. (master-2.x)
- feature

Im beschriebenen Szenario und aus den oben aufgeführten Gründen sind develop und master branch zusammengeführt worden.
Eine Entwicklung findet also direkt auf dem master branch statt. Sofern ein neues major release ausgerollt werden soll, wird hierfür ein neuer Branch erzeugt, der alle weiteren minor releases zu dieser Iteration beinhaltet.
Hierzu wurde bereits der branch master-2.x erstellt, der also zum einen den Stand der releasten Version 2.0 enthält und ebenfalls zukünftige Inkremente der Version enthalten wird.
Befüllt wird der genannte Branch durch Commits, die per pull-request direkt auf dem master erstellt wurden und nachträglich im Zuge der Erstellung einer neuen Version auf den master-x.x branch integriert und zu einer neuen Version gebündelt werden.
Damit entfällt also ein etwaiger release branch, da jeweilige Änderungen direkt und so lange auf dem master branch integriert werden, bis dieser genug neue features enthält, um eine neue Version zu erstellen.

3.3.3 Vor- und Nachteile aus der Praxis

Das unter 3.3.2 beschriebene Release Verfahren bringt gegenüber dem unter 3.3.1 vorgestellten in weiten Teilen umfänglicheren Vorgehen einige Vorteile mit sich, die dazu geführt haben, dass diese Variante durch eine Entscheidung im Projektteam für die ews-java-api verwendet wird.
Im Folgenden werden diese Vor- und Nachteile gegeneinander abgegrenzt.

Vorteile:

- Der favorisierte Workflow ist minimalistischer aufgebaut.
- Das Vorgehen hat sich bereits in anderen Projekten bewährt und ist somit erprobt.
- Fehleranfälligkeit bei der Erstellung von feature branches wird minimiert.
- Transparenz bleibt für einzelne Releases erhalten.
- Dokumentation des Vorgehens ist vereinfacht möglich.

Nachteile:

- Die Integration von Features ist auf ein Inkrement begrenzt.
- Das Szenario umfasst weniger Stufen und ist damit anfälliger für Fehler, die ggf. bei der Integration auf einen anderen Zweig aufgefallen wären.
- Weniger Transparenz in der Historie, da Bündelung der Releases auf einzelnen branches entfällt.
- Vorgehen entspricht nicht dem von vielen Projekten praktizierten Standard.

3.3.4 Semantic Versioning

Semantic Versioning gibt einzelnen Projekten die Möglichkeit, die Versionierung der publizierten Releases einem allgemeinen und verständlichen Standard zu unterziehen.
Hierbei steht im Vordergrund, dass die unter Semantic Versioning deklarierten Aspekte keine Grundlegenden Neuerungen bringen. Das erklärte Ziel ist vielmehr eine Transparenz über die jeweiligen Eigenschaften einzelner Releases zu schaffen. Zum Zeitpunkt der Erstellung dieser Arbeit liegt die Beschreibung des Sementic Versioning Prozesses in der Version 2.0.0 vor.

In der Umsetzung auf die ews-java-api wurde die Verbreitung von Releases unter dem Label Semantic Versioning ebenfalls deshalb eingeführt, da Benutzern die Möglichkeit gegeben werden sollte, sich über den jeweiligen Veränderungsgrad einer Version zu seinem Vorgänger zu informieren. Zusammengefasst lassen sich die Regeln des Semantic Versioning Prozesses wie folgt darstellen:
Gegeben sei eine Versionsbezeichung des Typs: *MAJOR.MINOR.PATCH (z.B. 2.0.0)*
(Tom Preston-Werner 2013)

Dabei gelten folgende Regeln für das Inkrementieren der Versionsnummer:

1. Erhöhen der *MAJOR* Version, wenn Änderungen durchgeführt wurden, in dessen Umfang die API inkompatibel zu einer Vorgängerversion ist.
2. Erhöhen der *MINOR* Version, wenn Funktionalität hinzugefügt wurde, die API aber noch rückwärtskompatibel einsetzbar ist.
3. Erhöhen der *PATCH* Version, wenn Fehler in dieser Version behoben wurden, die API aber noch rückwärtskompatibel einsetzbar ist.

(Tom Preston-Werner 2013)

3.4 Qualitätskriterien und Qualitätsziele der ews-java-api

In folgendem Kapitel werden die für die ews-java-api angesetzten Qualitätskriterien diskutiert. Hierzu wird die aktuelle Ausgangslange mit der auf der Vorstellung des Autors aufgebauten Zielvorstellung und den definierten Qualitätskriterien verglichen.

Der beschriebene Sachverhalt ist vor allem deshalb ein maßgebliches Ziel, da die bisherigen Kriterien zur Umsetzung einer angemessenen Abdeckung anhand von Testfällen und somit zur eigentlichen Qualitätssicherung nicht ausreichend beitragen konnten und somit erweitert werden müssen.
Dies wird im vornehmlichen Maße dann deutlich, wenn Änderungen an bestehenden Klassen der API durchgeführt werden und die zugehörigen Auswirkungen auf Teilbereiche der API nur durch eine ausgeprägte Analyse erhoben werden können, bzw. eventuell auch erst im Rahmen des produktiven Einsatzes aufgedeckt werden.

3.4.1 Ist-Zustand

Zum Zeitpunkt der Erstellung dieser Arbeit weist die ews-java-api einen sehr geringen Grad an Testabdeckung auf.
Ebenfalls sind zu diesem Zeitpunkt noch keine transparenten und offengelegten Kriterien über den Testprozess publiziert worden. Auch umfasst die C# Alternative der ews-java-api (im Folgenden als ews-managed-api bezeichnet) keine Tests und keine entsprechende Dokumentation einer Qualitätssicherung. Lediglich die Schnittstellen und Programmbausteine der ews-managed-api sind gut dokumentiert.
Da bei allen Anpassungen ebenfalls darauf geachtet wurde, die API nicht oder nur in begründeten Fällen dahingehend zu verändern, dass sich diese in der Verwendung der API unterscheidet, können anhand dieser Dokumentation Anforderungen an die Software an sich abgeleitet werden.
Hierbei können aber meist nur die funktionalen Anforderungen evaluiert werden, auf die dann im Nachgang, die bereits beschriebenen Blackbox-Verfahren bzw. die Testmethoden angewendet werden können.

Die nachfolgende Abbildung veranschaulicht hierbei, in welcher Form einzelne Bereiche bzw. Pakete der ews-java-api bereits durch ein Testverfahren abgedeckt sind und deren Inhalte zumindest durch einen durchlaufenden Test abgedeckt werden.
Die jeweiligen Analyseergebnisse wurden aus einem durch die Projektintegration mit der Webseite codecov.io erstellten Analyseergebnis entnommen.

Package	Statements	Hit	Missing	Partial	Branches	Coverage
Autodiscover	2073	104	1968	1	372	5,02 %
Core	9427	799	8552	76	885	8,48 %
Credential	91	6	85	0	7	6,59 %
Dns	55	5	50	0	7	9,09 %
Messaging	77	0	77	0	8	0,00 %
Misc	1490	216	1264	10	193	14,50 %
Notification	389	0	389	0	55	0,00 %
Property	5456	380	5015	61	1020	6,96 %
Search	635	0	635	0	62	0,00 %
Security	95	5	87	3	8	5,26 %
Sync	43	16	25	2	3	37,21 %
Util	600	600	0	0	5	100 %
Project total	**20431**	**2131**	**18147**	**153**	**2625**	**10,42** %

Tabelle 2 Ergebnis Testabdeckung codecov.io

In der obigen Grafik kann der aktuelle Projektfortschritt in Bezug auf die Testabdeckung abgelesen werden.
Im Gesamtprojekt liegt hier eine Überdeckungsrate von ca. 10% vor. Da die Abdeckung in den einzelnen Bereichen nicht durch eine nach speziellen Kriterien ausgerichtete strategische Planung vollzogen wurde, sondern nur darauf beruht, dass in diesen Bereichen neuere Änderungen durchgeführt worden sind, ist das angegebene Resultat vor allem mit der Einordnung der API in den Bereich des Legacy Codes verbunden.

Hierbei wird sowohl die Bedeutung des Legacy Codes in Bezug auf das Fehlen von Tests innerhalb eines bestimmten Bereichs deutlich, als auch die Bedeutung, dass unübersichtliche Codesegmente bei anstehenden Anforderungen verändert werden müssen. Dies besonders vor dem Hintergrund, dass der aktuell vorliegende Code neben den bereits manuell geleisteten Java-spezifischen Anpassungen (die aus der Transformation begründet sind) seinen Ursprung in der unter C# programmierten ews-managed-api hat. Da die Testabdeckung, wie aus Tabelle 2 abgelesen werden kann, in fast allen Bereichen der API sehr niedrig ist, müssen die Auswirkungen z.B. nach einer Änderung auf manuellem Wege ermittelt werden. Dies kann besonders bei größeren Änderungen nur in sehr geringem Maße gewährleistet werden.
Der geschilderte Sachverhalt zum Thema des aktuellen Status der API veranschaulicht, wie die API im Bereich der Überdeckungstests aufgestellt ist.

Für die Arbeit in der Community wurden ebenfalls grundlegenden Anforderungen an Contributions definiert, da in Bezug auf die Weiterentwicklung als Open-Source Projekt ebenfalls Anforderungen an beigesteuerten Source Code anfallen. Hier gelten auch im Hinblick auf die geschilderten manuellen Reviews folgende Kriterien:

- Der Code sollte für alle Reviewteilnehmer leicht verständlich sein.
- Neue Codebestandteile sollten durch Komponententests abgesichert sein.

- Die Funktionalität von Methoden und komplexen Segmenten sollte durch Kommentare und JavaDoc auch für spätere Änderungen dokumentiert worden sein.
- Weiterführende Features und Neuerungen sollten im Wiki ergänzt bzw. aufgeführt und beschrieben werden.
- Vor der Integration von neuem Code wurde von mindestens einem Core-Contributor eine Befürwortung signalisiert. Die Ernennung eines Core-Contributors erfolgt durch einen Vertreter von Microsoft.

Die vier zuerst aufgeführten Punkte dienen vor allem der Übersichtlichkeit der eingebrachten Features oder Issues.
Wohin gehend die Befürwortung von einer Person aus dem offiziellen Projektteam ebenfalls dem Aspekt Rechnung trägt, dass die Funktionsweise der API nicht durch eine spezielle Interessengruppe in unangebrachter Art und Weise mit Funktionen erweitert wird, die z.B. der Lösung eines individuellen Problems dienen oder ein Feature beinhalten, dass durch die globale Roadmap des Projektes nicht abgedeckt wird.
Hier geht es ebenfalls darum, die Qualität der Änderungen im Hinblick auf den Anwendernutzen grob einzuschätzen, um eine Priorisierung auf einzelne Releases zu erzielen.

3.4.2 Soll-Zustand

Ein maßgebliches Ziel, dass den Soll-Zustand des Projektes ews-java-api beschreibt ist, wie im Unterpunkt Ist-Zustand beschrieben, die Anwendung und Erhöhung einer Abdeckung durch kontrollflussorientierte Testverfahren.
Dies besonders vor dem Hintergrund eine höhere Qualität in Bezug auf die automatisierte Ausführung von Testfällen zu erzielen und ebenfalls mögliche (bereits einem Test überführte) Seiteneffekte, die durch eine Änderung hervorgerufen werden, aufzeigen zu können.
Um dieses Ziel zu erreichen sind kleinteiligere Meilensteine nötig, um jeweils sowohl einen nachhaltigen Qualitätseffekt und einen anhaltenden Mehrwert zu erzielen.

Ein weiteres Ziel ist es im Sinne einer build verification einen kritischen Anwendungspfad für die Software zu definieren. Dies würde es ermöglichen, die grundlegend wichtigen und meistgenutzten Bereiche der Anwendungslogik einem Test zu unterziehen, bevor das Artefakt entweder in Rahmen eines snapshot-release oder eines stable-release veröffentlicht wird. Dies würde den aktuellen Prozess wesentlich verschlanken, da eine Qualitätsprüfung im Rahmen einer manuellen Prüfung erfolgen muss.
Da sich das beschriebene Szenario in großen Teilen allerdings auf den Integrationstest bezieht, ist eine Einordnung dieser Verifizierung eines kritischen Anwendungspfades erst zu einem späteren Zeitpunkt eingeplant. Beiden Aspekten wird in Abschnitt 5 im Rahmen einer weiteren Auseinandersetzung Rechnung getragen.
Um dem genannten Schritt Rechnung zu tragen werden im Vorfeld einige Anpassungen an den zu testenden Klassen der API vorgenommen werden müssen. Hier ist es z.B. für die Bereitstellung einzelner Komponententests unerlässlich, dass Abhängigkeiten mit den in den Grundlagen vorgestellten

Mitteln für einen Test durch etwaige Dummy-, oder Mock-Objekte ersetzt werden. Andernfalls wäre im Rahmen eines Tests mit diversen abhängigen Komponenten der gekapselte Test nur eines Bestandteils der Komponente nicht möglich und der Test müsste in die Kategorie der Integrationstests eingeordnet werden.
Je nach Verzweigungsgrad der abhängigen Komponenten hätte dies sowohl Auswirkungen auf die Komplexität des Tests, als auch auf die Laufzeit des Build-Verfahrens, sofern der jeweilige Test in die Menge der automatisierten Tests integriert ist.

Neben der Aufstellung der einzelnen Meilensteine und dem Eintritt in deren Implementierungsphase, ist ein weiteres Ziel, die im vorherigen Abschnitt definierten Review Kriterien weiterführend zu automatisieren.
Das Ziel hierbei sollte es sein, den manuellen Reviewaufwand auf Aspekte einer guten Programmierung (in Bezug auf Verständlichkeit des Codes), Implementation von Tests und Besonderheiten in der Kommunikation mit dem Exchange-Server zu legen.
Die derzeitig anfallenden Reviews sehen aufgrund einer fehlenden Automatisierung in den jeweiligen Bereichen noch wiederholt Anmerkungen z.B. zum Styleguide, fehlenden License-Headern o.ä. vor, was mit einer zunehmenden Bindung an Kapazitäten einhergehend ist.

Die Grundlage für die Überführung der API in den Soll-Zustand bildet die Evaluierung der funktionalen Anforderungen, um vor Anwendung der gewonnenen Erkenntnisse eine klare Zielvorstellung über den notwendigen Funktionsumfang zu gewährleisten.
Dies kann ebenfalls für die Inklusion eines Smoke-Tests im Sinne eines kritischen Anwendungspfades sehr von Vorteil sein. Eine Einführung über die Schwierigkeiten und aus der jeweiligen Dokumentation entnommenen Anforderungen wird im folgenden Abschnitt aufgezeigt.

Neben der definierten Zielsetzung ist ein weiteres Ziel die aus großen Unternehmen (Abschnitt 4) bekannten Strategien anzuwenden. Hier erhofft sich der Autor dieser Arbeit zum einen die schnellere Erreichung der definierten Ziele und ebenfalls weitere für das Projekt positive Begleitfaktoren.

3.4.3 Anforderungsmanagement

Im Sinne des Anforderungsmanagements steht das Softwareprojekt ews-java-api vor der Problematik, dass Anforderungen an das System nur aus der für das C# Projekt erstellten Dokumentation abgeleitet werden müssen.
Die Ableitung der Anforderungen wird hierbei ebenfalls durch die unterschiedliche Umsetzung auf Quellcodeebene begleitet.
Dennoch sind an beide APIs grundlegend die gleichen Anforderungen zu stellen, auch wenn sich der jeweilige Implementierungsfortschritt und auch der Fortschritt im Hinblick auf die Qualitätssicherung beider Projekte zum Teil in eklatanter Weise unterscheidet.
Grundlegend konnten die aus der im Office Dev Center verfügbaren Dokumentation entnommenen Anforderungen in zwei Kategorien eingeteilt

werden. Diese bestehen zum einen aus Basisoperationen, die eine Grundfunktionalität der API veranschaulichen. Hierzu zählen z.B. die Handhabung von Ordnerstrukturen auf dem Server, die Suche nach gewissen Elementen und das Erstellen und Bearbeiten von E-Mails, Aufgaben und Terminen. In die zweite Kategorie fallen Anforderungen, die über die gerade erwähnten grundlegenden Operationen hinausgehen. Hierzu zählen z.B. die Suche nach erfolgten Konversationsverläufen, das Synchronisieren einer lokalen Datenquelle mit dem Exchange-Server oder das Auflösen von Kontaktgruppen in einzelne E-Mail-Adressen, die im Nachgang weiterverarbeitet werden können.

Da es sich bei den jeweiligen Operationen immer um eine Kommunikation mit einer entfernten Ressource (dem Exchange-Server) handelt, ist für sämtliche Operationen eine Verbindung zu der jeweiligen Serverinstanz notwendig. Zusätzlich wird für die Authentifizierung an den jeweiligen Server ein entsprechendes Token notwendig. Im einfachsten Fall kann dies eine Kombination aus Benutzername und Passwort sein.
Ebenfalls kann mit Benutzung der API auch eine Validierung der Authentifizierungsinformationen mittels des Azure AD-Authentifizierungssystems z.B. mit einer WS-Security-Authentifizierung erfolgen.
Hier sind allerdings entsprechende Kenntnisse über Verwendung und Generierung der entsprechenden Tokens notwendig. In der nachfolgenden Abbildung sind die extrahierten Anforderungen noch einmal innerhalb eines UML-Anwendungsfalldiagramms dargestellt:

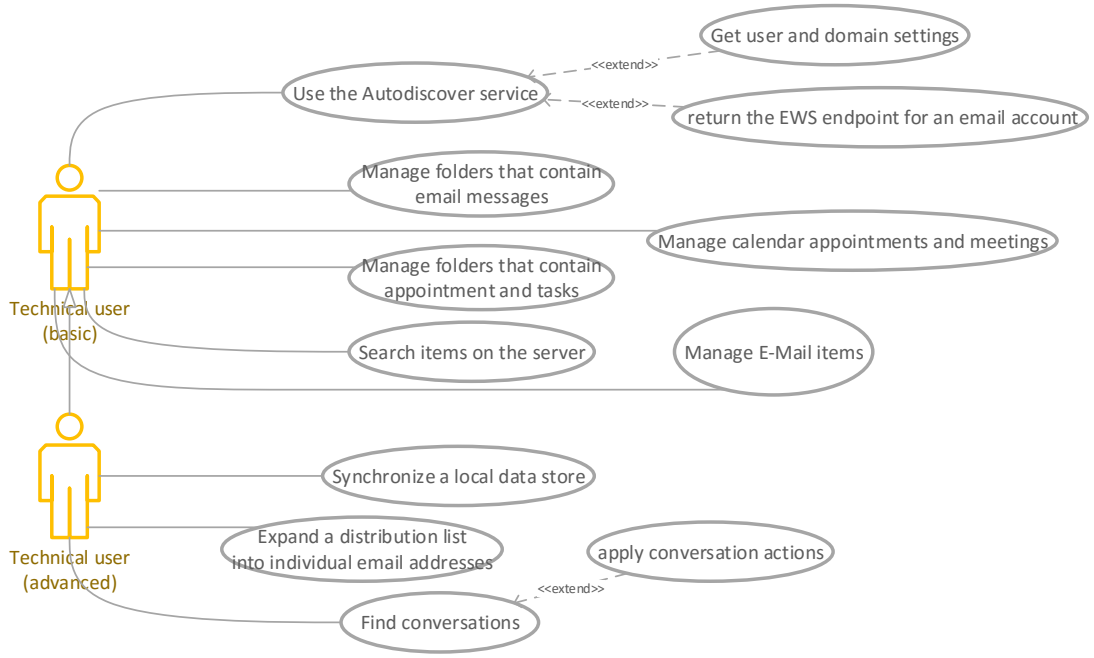

Abbildung 5: Anforderungen an die ews-java-api (Office Dev Center 2014)

Nach Maßgabe des aktuellen Entwicklungsstands der API werden alle Exchange-Server Versionen in den unter Basis angegebenen Anforderungen unterstützt. Hierbei sind Sonderfunktionen bezüglich der jeweiligen Exchange-

Version aber nur für die Versionen 2007 und 2010 mit den jeweiligen Service Packs implementiert. Hier gilt es ebenfalls, die pro Version gelisteten Funktionalitäten iterativ in zukünftige Versionen zu übernehmen.

Die in Abbildung 5 dargestellten Anforderungen beschreiben das System bezüglich der funktionalen Anforderungen, die zur Interaktion mit dem Exchange-Server notwendig sind.
Eine Übersicht, über vorhandene nicht-funktionalen Anforderungen konnte im Rahmen der Recherche in der Dokumentation nicht evaluiert werden.
Da die API aber zunehmend durch technische Benutzer, also Programmierer Verwendung finden wird, können die weiter unten dargestellten nicht-funktionale Anforderungen aus Sicht des Autors geltend gemacht werden.
Zur Erstellung der Anforderungen wurde ein aus der agilen Entwicklung bekanntes Muster zur Formulierung von Anforderungen in Form einer User-Story genutzt.
Die Struktur des Patterns spiegelt sich hierbei wie folgt wieder:

> As a <*role*>, I want <*goal/desire*> so that <*description of benefit*>
> (Connextra User Story 2001: ConnextraStoryCard)

- Als Softwareentwickler, der die ews-java-api innerhalb eines anderen Projektes nutzt, möchte ich eine einfache Erweiterbarkeit ohne grundlegende Veränderungen der Schnittstellen. Hiermit wird dem Aspekt Rechnung getragen, dass etwaige Änderungen, die für eine neue Generation von Exchange-Servern implementiert werden, möglichst keine Anpassungen an bereits bestehende Methoden in den öffentlichen Schnittstellen hervorrufen.
- Als Mitglied der Entwickler-Community der ews-java-api möchte ich eine hohe Wartbarkeit der Software, da für weitere Anpassungen ein leichtes Verständnis des bereits vorhandenen Codes sehr wichtig ist.

3.5 Design-Übersicht

Die unten dargestellte Hierarchie spiegelt die in der ews-java-api dargestellte Klassenhierarchie der einzelnen Exchange Objekte wieder.
Jede aufgeführte Klasse spiegelt die Repräsentation eines Exchange-Objektes wieder und erbt spezifische Methoden und Attribute entlang der Vererbungshierarchie.
In Bezug auf die Wiederverwendbarkeit können somit im Falle neuer Items, die mit einer neueren Exchange-Version eingeführt werden, entsprechende Erweiterungen implementiert werden.
Ebenfalls beinhalten die Objekte je nach Serverversion des in Verbindung stehenden Exchange-Servers, unterschiedliche Funktionalitäten.
Werden diese durch den angesteuerten Server nicht unterstützt, wird der Benutzer durch eigene Exceptions auf diesen Sachverhalt hingewiesen.

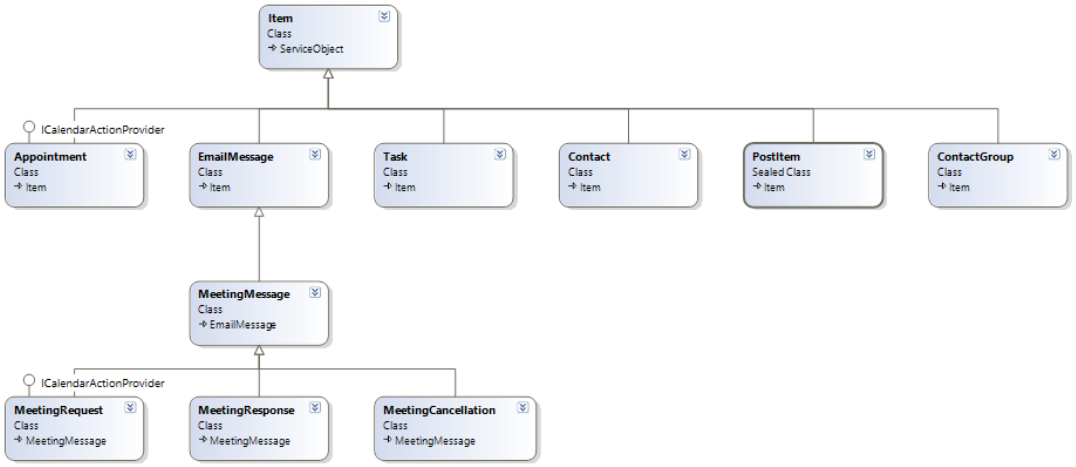

Abbildung 6 ItemHierarchy (OfficeDev/ews-java-api)

Zusätzlich zu den eingangs erwähnten Items, die Operationen auf dem entfernten Exchange-Server auslösen können, existieren einzelne Klassen, die eine jeweilige Ordnerstruktur zur Ablage einzelner Elemente auf dem Server ermöglichen. Hierbei kann ebenfalls auf einzelne Ordner zugegriffen werden oder über den Inhalt des jeweiligen Ordners iteriert werden.
Eine clientseitige Anwendung kann somit folgende Operationen für Ordner und Objekte auf dem Exchange-Server ausführen: create, update, delete, copy, find, get.
Die der API zugrundeliegenden Klassen, die eine soeben beschriebene Operation auf den jeweilig entfernt liegenden Ordnern und Ordnerstrukturen ermöglichen, werden in Abbildung 7 dargestellt.

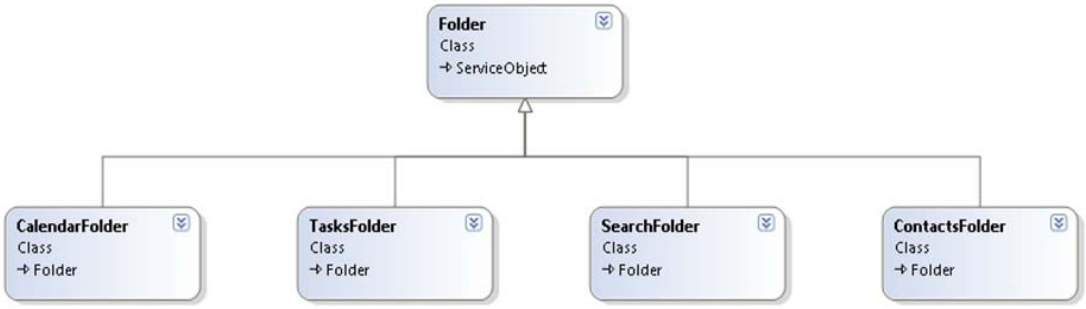

Abbildung 7 FolderHierarchy (OfficeDev/ews-java-api)

3.6 Testinfrastruktur und Qualitätssicherung

Zur Qualitätssicherung werden in der Weiterentwicklung der ews-java-api diverse Ansätze verfolgt.
Hier geht es primär darum, etwaige Contributions, die von Personen mit divergenten Kenntnisständen in der genutzten Programmiersprache oder der Funktionsweise der API an sich getätigt werden, unter einem einheitlichen Dach zu vereinigen. Da die API aufgrund des abgebildeten Bereichs zunehmend im gewerblichen Umfeld zum Einsatz kommt, gilt es ebenfalls die Gesamtsicht über die Funktionalität und mögliche Fehlersituationen bzw. Anfälligkeiten nicht aus dem Auge zu verlieren.
Um die beschriebenen Ziele einhalten zu können werden unterschiedliche Tools eingesetzt, auf die in diesem Kapitel näher eingegangen werden soll und die den Ablauf der Qualitätsprüfung eines Features oder einer Fehlerbehandlung unterstützen.
Dennoch findet ein Großteil der Qualitätssicherung zum Zeitpunkt der Erstellung dieser Arbeit mittels Kommunikation im Team statt. Die hierbei angestrebten Ziele und unterstützenden Maßnahmen werden ebenfalls in diesem Kapitel behandelt werden.

3.6.1 Eingesetzte Tools (SaaS)

Zur Gewährleistung und einfacheren Handhabung der Qualitätssicherung des Projektverlaufs werden einzelne Tools im Sinne einer Bereitstellung als SaaS (Software as a Service) eingesetzt, die eine transparente Darstellung des jeweiligen Status vor und nach einer Änderung darstellen. Grundlage hierfür bildet die Einbettung eines Continuous Integration Servers, der im vorliegenden Fall der ews-java-api ebenfalls die Funktion einer kontinuierlichen Installation (Continuous Delivery) übernimmt. Nachdem ein Softwareartefakt die unterschiedlichen Testautomatisierungsstufen und die CI durchlaufen hat, werden erfolgreich gebaute Artefakte ebenfalls als Snapshot Version in das entsprechende Repository geladen und stehen der Community zur weiteren Verwendung in einem nicht produktiven Umfeld zur Verfügung.
Zur Auswahl im Bereich Continuous Integration standen unterschiedliche Angebote. So z.B. AppVeyor, TeamCity oder TravisCI. Die folgende Auflistung zeigt einige für die ews-java-api wichtige Anforderungen, die an einen Contiuous Integration Server gestellt wurden.

Name	Gated-Commits	Nutzungskosten	Einfach konfigurierbar	Verbreitungsgrad z.B. für Support
TeamCity	Ja	Ja	Ja (nicht per yaml Datei)	Kommerzieller Bereich
AppVeyor	Nein	Nein	Ja	Für .NET Projekte (Einbettung der NuGet Paketverwaltung)
Travis CI	Nein	Nein	Ja	Stark verbreitet auf github.com für Java Projekte

Tabelle 3 Ergebnis Auswertung SaaS

Neben der Anforderung, dass eine einfache github Integration unterstützt wird, was bei allen Anbietern der Fall war, spielten für die Entscheidung die Kriterien, ob ein Gated-Commit Feature vorhanden ist, die Nutzungskosten, Konfigurierbarkeit des Servers, sowie der Verbreitungsgrad der jeweiligen Komponente eine entscheidende Rolle.
Dies besonders auch vor dem Hintergrund, dass im Problemfall ebenfalls eine Lösung durch die Community erfolgen kann. Hier hatten sich sowohl der Anbieter AppVeyor, als auch TravisCI hervorgetan, da die Konfiguration per yaml Datei erfolgen kann und die Konfiguration somit durch jedes Teammitglied eingesehen und ebenfalls eine Änderung im Rahmen der genutzten Versionsverwaltung erfolgen kann.
AppVeyor bietet hierbei Vorzüge für die Nutzung von .NET Projekten und TravisCI ist als CI-Server zunehmend zu einem Standard in der Verbindung mit github.com geworden. Beide Server stehen für Open-Source Projekte kostenlos zur Verfügung, was ebenfalls einen Unterschied zu TeamCity symbolisiert, da eine Nutzung hier mit Kosten verbunden ist, bzw. ein explizites Sponsoring von JetBrains erfolgen muss, damit eine kostenfreie Nutzung ermöglicht werden kann.
Allerdings ist TeamCity der einzige zur Auswahl stehende CI-Server, der eine direkte Integration zwischen Entwicklungsumgebung und dem Build-Server ermöglicht.
Nach Abwägung aller evaluierten Anforderungen wurde dennoch der Anbieter TravisCI ausgewählt, da hier zum einen der Kostenfaktor berücksichtigt wurde und die Integration von Projekten auf github.com und TravisCI nach subjektivem Empfinden des Projektteams in den meisten Java-Projekten auf github.com anzutreffen ist. Dies ermöglicht es, im Problemfall ebenfalls auf Rückmeldungen aus der Community zurückzugreifen.

Ebenfalls durch etwaige Toolunterstützung wird der Build-Prozess der API mittels maven betrieben. Die Integration erlaubt es neben einer automatisierten Testausführung ebenfalls weitere zur Qualitätssicherung bereits erwähnte Tools in den Standard-Lebenszyklus eines Builds zu integrieren und somit den Grundgedanken Convention over Configuration beim Build-Prozess der API zu gewährleisten. Zum Zeitpunkt der Erstellung dieser Arbeit ist in diesem Zusammenhang ebenfalls eine Refaktorisierung bezüglich der Umstellung des Build-Managements zu Gradle geplant. Dies besonders vor dem Hintergrund, dass in diesem Zusammenhang eine einfachere Handhabung der Projektfiles ermöglicht wird.
Ebenfalls erhofft sich das Projektteam durch die Umstellung mehr Möglichkeiten, den Prozess auch in Hinblick auf eine parallele Ausführung weiter beschleunigen zu können.

Das gerade erwähnte Toolpaket wird ebenfalls durch den Einsatz einer Continuous Coverage Analyse erweitert.
Auch hier existieren unterschiedliche Anbieter dieser Lösungen, welche bei einer Auswahl des passenden Anbieters in Erwägung gezogen wurden. Im Fokus stand auch in diesem Zusammenhang eine einfache Einbettung, sowie die Möglichkeit transparent die jeweiligen Analyseergebnisse darstellen

zu können. Mögliche Vertreter, die eine Softwarelösung bereitstellen und in den Pool, der zur Auswahl stehenden Lösungen aufgenommen wurden, sind codecov.io, coveralls.io und sonarqube.org.
Das Angebot von sonarqube wäre in diesem Zusammenhang das weitest gehende, da neben einer Coverage Analyse ebenfalls weitere Metriken analysiert werden können. Allerdings kann sonarqube kostenlos nur auf einem eigenen Server betrieben werden und gehostete Varianten sind nicht ohne entsprechende Aufwendungen verfügbar.
Die anderen beiden genannten Vertreter unterschieden sich in ihrer Bedienung und Installation nur marginal. Dennoch bietet codecov.io den Vorzug, dass bei Einstellung eines entsprechenden pull requests auf das Projekt ein automatisierter Kommentar erzeugt wird, der bereits über die jeweilige Änderungsrate der Testabdeckung informiert.
Da dies ebenfalls dazu genutzt werden kann, im Zuge von Änderungen ebenfalls für eine entsprechende Testabdeckung zu sorgen und gleichermaßen transparent den Änderungsverlauf darzustellen, wurde codecov.io als SaaS zur Ermittlung des Überdeckungsgrads ausgewählt.
Im Zusammenhang mit der Erstellung der Metriken werden hier folgende Analysen ausgewertet:

- Anweisungsüberdeckung
- Zweigüberdeckung

3.6.2 Systemübersicht

Im Rahmen der automatisierten Integration werden einzelne im Build-Lifecycle verankerte Maßnahmen der Qualitätssicherung durchgeführt.
Genutzt wird hierzu der Continuous Integration Server TravisCI. Der Server weist laut Angaben des Anbieters folgende Spezifikationen auf:

Bezeichnung	Eigenschaft(en)
Virtualisierungsart	Docker Virtualisierung (Container-Based Infrastructure)
CPU / RAM	2 dedicated cores / 4 GB
Servertyp	Amazon Elastic Compute Cloud (EC2)
Operating System	Linux
Vorinstallierte JDKs	Oracle JDK 7 (oraclejdk7) OpenJDK 7 (openjdk7) OpenJDK 6 (openjdk6) Oracle JDK 8 (oraclejdk8)

Tabelle 4 TravisCI Systemübersicht

(Mathias Meyer 2014)

Zusätzlich zu den oben angegebenen Spezifikationen ist das build Skript so konfiguriert, dass zur Beschleunigung der Skriptausführung ein Caching der projektbezogenen abhängigen dependencies per maven erfolgt.
Diese werden jeweils vor dem Bild aus einem Archiv geladen und bei eventuellen Änderungen nach dem erfolgreichen build wieder in das Archiv zurückgesichert.
Um Seiteneffekte zu vermeiden, werden die zwischengespeicherten Sicherungen durch ein Mitglied des Projektteams in unregelmäßigen Abständen gelöscht und bei einem erneuten build werden automatisch neue Sicherungen angelegt.Das Skript ist ebenfalls so konfiguriert, dass eine Ausführung der Tests in unterschiedlichen, voneinander getrennten Umgebungen vollzogen wird.
Somit werden Tests auch aus Gründen der Unterstützung älterer Java-Versionen mit unterschiedlichen JDKs ausgeführt:

- Oraclejdk8
- Oraclejdk7
- Openjdk6

3.6.3 Manuelle Prüfung

Ein Großteil der Qualitätskontrolle einer Contribution erfolgt im Rahmen der ews-java-api manuell. Das Vorgehen wird hierbei von der Plattform github.com weiterführend unterstützt, da sog. pull requests, die eine Fehlerbehebung oder die Implementierung eines neuen Features erhalten, über die Plattformen einem Codereview unterzogen werden können.
Dies dient zum einen der Transparenz einem sich ständig ändernden Projektteams und zum anderen der Einbeziehung möglichst vieler Akteure und Steakholder an den jeweiligen Implementierungen.
Da die API im Open-Source Bereich entwickelt wird, sind somit sämtliche interessierten Personen ebenfalls eingeladen, den eingereichten Code von

anderen Personen im Hinblick auf eine qualitativ hochwertige Umsetzung zu überprüfen und ggf. durch eigene Beiträge, sei es durch die Verbesserung eines pull requests auf Codeebene oder durch die Anbringung von Hinweisen als Kommentar zu vervollständigen.
Dieses Vorgehen hat sich auch dadurch als praktikabel erwiesen, da sich die eingereichten pull requests über einen längeren Zeitraum verteilen und somit die Möglichkeit besteht, contributions vor Ihrer Übernahme in das nächste Snapshot Release über einen Zeitraum transparent im Sinne eines „Schwarzen Brettes“ anzukündigen.
Hinweise, die auf diese Art angebracht wurden, können sich in folgende Kriterien unterteilen lassen:

- Vorschläge zu Programmierfehlern (sofern nicht bereits durch autom. QS abgedeckt)
    - Inhaltliche Fehler
    - Besonderheiten / Hacks bezogen auf die jeweilige Exchange Version
    - Gängige Programmierfehler in JAVA
- Vorschläge zum angewendeten Styleguide (sofern nicht bereits durch autom. QS abgedeckt)
    - Formatierung des Codes
    - Namenskonventionen
    - Anbringen des Lizenzheaders (MIT-Lizenz) in neuen Dateien
- Vorschläge zur Anbringung weiterer oder Erweiterung der Testszenarios
- Anmerkungen zur Einbettung in bereits bestehende Projektanforderungen
- Alternative Umsetzungsstrategien und Workarounds
- Sonstiges

3.6.4 Automatisierte Prüfung

Neben der vorab dargestellten manuellen Überprüfung werden automatisierte Tests durch einen Integrationsserver ebenfalls mit jedem Commit auf den einzelnen Branches des Repositories durchgeführt.
Dies ist besonders dann sinnvoll, wenn einzelne Features oder Fehlerbehebungen in einen neuen Entwicklungszweig integriert werden sollen, die dort vorliegende Software aber einen anderen Entwicklungsstand hat als in dem Ursprung der Integration. Ebenfalls erfolgt eine automatisierte Prüfung wenn eine neue Version zur Auslieferung erstellt wird.
Dies dient zum einen der doppelten Absicherung, ebenfalls soll so verhindert werden, dass funktionsunfähige Software durch ein Versehen aus dem jeweiligen Repository mit in die Software eingebaut wird. Hier erfolgt ebenfalls eine Toolunterstützung, die verhindert, dass Änderungen, die nicht committet wurden in ein fertiges Artefakt überführt werden.

Neben der dynamischen Codeanalyse wird durch eine automatisierte Überprüfung ebenfalls eine statische Codeanalyse durchgeführt, welche Bedingungen der normierten Programmierrichtlinien des Softwareprojektes überprüfen.

3.6.5 Transparente Ergebnisdarstellung

Da die in den Abschnitten zuvor genannten Prüfungen ebenfalls zur Qualitätssicherung beitragen sollen, ist eine nachhaltige und transparente Ergebnisdarstellung der jeweiligen Tests zwingend erforderlich.
Zu diesem Zweck werden die gesammelten Ergebnisse automatisch zur Darstellung in ein HTML-Format überführt.

Zum gegenwärtigen Zeitpunkt können die ausgewerteten Informationen allerdings nach Ausführung der Tests nur auf dem lokalen Rechner eingesehen werden. Eine zentrale Sammlung der Testergebnisse, anhand derer ebenfalls ein Verlauf der Ausprägungen erkannt werden kann ist noch nicht möglich. Im Sinne der transparenten Darstellung der Ergebnisse aber vorgesehen. Die jeweiligen Testergebnisse können aber in textueller Form über den CI-Server anhand des jeweiligen Builds eingesehen und auch in die Vergangenheit zurückverfolgt werden.

3.6.6 Testdaten

Da die Bereitstellung von Testdaten mit zunehmender Komplexität der Abhängigkeiten eines Tests und des jeweiligen Testobjekts sehr aufwändig sein kann, wurde zur Durchführung der in dieser Arbeit verwendeten Tests jeweils ein Test mit Produktionsdaten eines zur Verfügung stehenden Exchange-Servers durchgeführt.
Dies kann sich zur echten Qualitätskontrolle eines Systems allerdings als problematisch erweisen, da im vorliegenden Fall ein ursächliches Problem ebenfalls in einer fehlerhaften Konfiguration eines Exchange-Servers oder aber in einer mit falschen Informationen gefüllten Antwort des Exchange-Servers verbunden sein kann.
Im Praxiseinsatz birgt der Test mit Produktionsdaten ebenfalls Risiken, da z.B. Belange des Datenschutzes berücksichtigt werden müssen, die meist eine Anonymisierung der Daten erzwingen.

3.6.7 Testerstellungskriterien

Um erfolgreich innerhalb eines Projektes testen zu können wurde im Rahmen der ews-java-api für die zu erstellenden Tests eine gleiche Paketstruktur angewendet wie bereits aus dem Umgang mit den zu testenden Klassen bekannt. Dies ist besonders vor dem Hintergrund zu sehen, dass die Testklassen zu einem Testobjekt möglichst schnell wiedergefunden werden können. Um die Qualität eines einzelnen Tests beurteilen zu können, sollte bei der Testerstellung auf folgenden Kriterien geachtet werden, die eine möglichst genaue Fehlerreproduktion ermöglichen:

- Wiederholbarkeit des Tests und Produktion von deterministischen Ergebnissen
- Schlichtheit von Tests und Ausrichtung auf ein bestimmtes Problem
- Unabhängigkeit von Tests und deren Isoliertheit gegenüber anderen Tests und deren Ergebnissen

(Resig und Bibeault 2014, S. 44)

4 Teststrategien in großen Unternehmen

Im folgenden Abschnitt werden Strategien zur Umsetzung von Testansätzen in größeren Unternehmen näher erörtert. Hierzu wurden exemplarisch Teststrategien, best practice Ansätze und genutzte Tools der Unternehmen Microsoft und Google anhand einer Literaturrecherche evaluiert und gegenübergestellt.
Hieraus können sich im nachfolgenden Kapitel Ansätze ergeben, welche Bereiche hier ebenfalls auf die ews-java-api angewendet werden können, um die ebenfalls erstellten kurzfristigen, mittelfristigen und langfristigen Ziele aufzustellen.
Es werden Strategien präsentiert, die dem Autor als besonders geeignet zur Übertragung auf die Anwendung in Zusammenhang mit der ews-java-api und den Einsatz im Bereich Legacy Code erscheinen.
Ebenfalls wird erwähnt werden, warum die jeweilige Teststrategie von Nutzen sein kann und in einer späteren Betrachtung werden die einzelnen Strategien auch Anwendung in Bezug auf die ews-java-api finden und es wird erörtert, wie und mit welchen Schwierigkeiten die Implementierung erfolgt ist.

4.1 Teststrategien bei Google

Google bietet als Unternehmen eine breite Palette an unterschiedlichen Softwarelösungen an. Um diese ebenfalls einer Qualitätssicherung zu unterziehen, verwendet Google eine Reihe unterschiedlichster Strategien, die sich ebenfalls auf andere Projekte übertragen lassen.
Im späteren Verlauf wird ebenfalls anhand einer Vergleichsmatrix erörtert, welche der Strategien eine besondere Eignung finden können. Dabei soll dieser Abschnitt eine Auswahl gängiger Paradigmen vorstellen und diese ebenfalls erläutern.

4.1.1 Anforderungen an Tests zur Laufzeit

Gerade im Hinblick auf Systeme, in denen mehrere Tests unter Umständen sogar parallel ausgeführt werden sollen, ergeben sich Anforderungen, die an eine korrekte Testausführung gestellt werden und dafür sorgen, dass die erzielten Ergebnisse ebenfalls als deterministisch angesehen werden können. Hierunter fallen folgende Grundvoraussetzungen:

- Jeder Test muss in sich abgeschlossen und unabhängig gegenüber anderen Tests und deren Ausführungsreihenfolge sein.
- Die Laufzeitumgebung darf durch Tests nicht verändert oder temporär beeinflusst werden.

Um das beschriebene Verhalten zu überprüfen haben Entwickler bei Google z.B. die Möglichkeit, per Anforderung darüber zu entscheiden, ob die Ausführungsreihenfolge bewusst variiert werden soll. Hierbei sollte gewährleistet sein, dass der Status „any order“ auch dazu führen kann, dass eine parallele Testausführung erfolgen kann. Dies kann z.B. dann problematisch werden, wenn Tests einen exklusiven Portzugriff auf dem System benötigen, ein Ordner oder eine Datei auf dem System unter demselben Pfad erzeugt werden, oder aber eine gleichnamige Datenbanktabelle Verwendung findet. So z.B. wenn die Zugriffe durch konkurrierende Operationen (z.B. Löschen der Tabelle und gleichzeitiger Schreibversuch, durch einen anderen Prozess/Test) eine gegenseitige Blockade oder ein fehlerhaftes Testresultat verursachen.
(Whittaker et al. 2012, S. 48–49)

4.1.2 Aufteilung der Tests in einzelne Kategorien

Gerade die Vorzüge der automatisierten Testausführung z.B. im Rahmen eines Continuous Integration Servers verknüpfen die Ausführung einzelner Tests mit der Notwendigkeit diese in einer unterschiedlichen Häufigkeit und im Rahmen eines variierenden zeitlichen Umfangs und auf einem unterschiedlichen Isolationslevel auszuführen.
Hier spielen ebenfalls die unterschiedlichen, bereits aus den Grundlagen bekannten Teststufen eine signifikante Rolle, welche Google prinzipiell ohne die bereits bekannten Bezeichnungen Unit Test, Integrationstest, Systemtest etc. in einzelne Kategorien unterteilt hat.
Zusätzlich zu den bereits bekannten Spezifikationen fügt Google im Rahmen der Definition der einzelnen Segmente allerdings weitere Voraussetzungen für die entsprechende Einordnung hinzu.

Getreu dem KISS Prinzip („Keep it simple [and] stupid“) erfolgt die Unterteilung einzelner Tests in die Kategorien „kleine Tests“, „mittlere Tests“ und „große Tests“. Je nach Kategorie unterscheiden sich hierbei sowohl das Isolationslevel zu anderen Komponenten, die Ausführungsgeschwindigkeit und die mit einer Einteilung verbundenen Ziele, Stärken und Schwächen.
Die einzelnen Definitionen lassen sich hierbei wie folgt in Kernziele ableiten und wurden vornehmlich zu dem Zweck eingeführt, dass Tests im Bereich der continuous integration in eine gängige Normung unterteilt werden können und somit ebenfalls je Einstufung Aussagen über die Laufzeitumgebung, abhängige Komponenten oder die Ausführungsgeschwindigkeit erwogen werden können.
Ebenfalls bietet die Einordnung von Tests innerhalb der einzelnen Kategorien die Möglichkeit, durch Einhaltung der jeweiligen Spezifikation, eine schrittweise Überführung z.B. von bereits bestehenden Tests, in die jeweils angestrebte Kategorie, durchzuführen.

Kleine Tests:

- haben eine sehr kurze Laufzeit (weniger als 100 ms) und schlagen bei Überschreitung von 1 Minute automatisch fehl.
- erzeugen klareren Code, da erforderliche Abhängigkeiten innerhalb des Tests gemockt werden und somit Interfaces zwischen den Systemen als Spezifikationen bekannt sind.
- geben schnelles Feedback und können oft ausgeführt werden, z.B. im Rahmen von Test-Driven-Development.
- haben einen eindeutigen Zielerfüllungsgrad und sind deshalb speziell auf eine mögliche Problemsituation zugeschnitten.
- Subsysteme können manchmal schwierig durch mocks ersetzt werden
- Mock- und Fake-Objekte können auf einem veralteten Stand beruhen

Mittlere Tests:

- haben eine relativ kurze Laufzeit und können deshalb regelmäßig ausgeführt werden (weniger als 1 Sekunde) und schlagen bei einer Überschreitung von 5 Minuten automatisch fehl.
- bieten dem Entwickler die Möglichkeit, Tests schrittweise in die Rubrik der kleinen Tests zu überführen.
- können in der Entwicklungsumgebung der jeweiligen Programmierer ausgeführt werden.
- ggf. sind diese nichtdeterministisch, da externe Systeme und abhängige Systeme und Schnittstellen ggf. nicht vollständig gemockt wurden.

Große Tests:

- haben eine Laufzeit, die mit „so schnell wie möglich“ betitelt wird.
- ggf. sind diese nichtdeterministisch, da externe Systeme und abhängige Systeme und Schnittstellen nicht gemockt wurden.
- testen die wesentlichen Bestandteile der Applikation und das Verhalten von externen Subsystemen
- aufgrund der Vielzahl der Abhängigkeiten kann es ggf. schwierig sein, den Fehlerzustand zu finden.
- können im Bereich des Datensetups sehr komplex sein.

Eine hohe Praxisrelevanz haben in diesem Zusammenhang ebenfalls die für die jeweilige Einteilung in eine Kategorie angesetzten Abhängigkeitskriterien, ohne deren Einhaltung eine wie oben angegebene Verschlankung auch in Bezug auf die Ausführungsgeschwindigkeit von kleinen Tests wohl nicht möglich wäre.

Resource	*Large*	*Medium*	*Small*
Network Service	Yes	Localhost	Mocked
Database	Yes	Yes	Mocked
File System Access	Yes	Yes	Mocked
Access to User-Facing Systems	Yes	Discouraged	Mocked
Invoke Syscalls	Yes	Discouraged	No
Multiple Threads	Yes	Yes	Discouraged
Sleep Statements	Yes	Yes	No
System properties	Yes	Yes	No

Tabelle 5 Benötigte Ressourcen nach Testgröße
(Whittaker et al. 2012, S. 44–48)

4.1.3 Erstellung eines Testkonzepts in 10 Minuten

Bei der Erstellung eines Testkonzepts spielen häufig zeitliche Faktoren eine Rolle. Ebenfalls muss das Konzept ggf. im weiteren Projektverlauf oftmals überarbeitet und angepasst werden. Geschieht dies nicht, droht die Dokumentation auch aus Gründen des Verlustes ihrer Aktualität nicht mehr aussagekräftig zu sein.
Ebenfalls kann eine Problemstellung sein, den richtigen Umfang für die Dokumentation zu finden, um genügend, aber dennoch nicht überflüssige Informationen bereitzustellen.
Im Rahmen eines Feldversuchs wurde durch James Whittaker ein Team damit beauftragt, das Testkonzept für ein Software Artefakt in der Zeit von 10 Minuten zu erstellen. Das entsprechende Projektteam war bei der Erstellung ebenfalls angehalten, die Dokumentation auf die wesentlichen Inhalte zu beschränken. Zwar wurde von keinem der Teams die Zeitvorgabe von 10 Minuten für die Erstellung des Testkonzepts eingehalten, allerdings konnte nach einer weiteren Ausdehnung um 20 Minuten ca. 80 % des Gesamtergebnisses erreicht werden. (Whittaker et al. 2012, S. 103–104)

Das im Rahmen dieses Feldversuchs erzielte Gesamtergebnis muss hierbei ebenfalls am ausgewiesenen zeitlichen Rahmen gemessen werden und gibt in erster Linie eine Aussage darüber, dass auch durch eine gezielte Teamarbeit in Verbindung mit z.B. Brainstorming bereits aussagekräftige, valide Ergebnisse erzielt werden können und könnte auch durch die Aussage „to put the right people in the right place“ versinnbildlicht werden.

4.2 Teststrategien bei Microsoft

Auch Microsoft bietet als Softwarehersteller unterschiedlichste Produktlösungen an und hat auch im Bereich der Projektplanung einige Unternehmensexpertise vorzuweisen.
Der folgende Abschnitt soll einige der im Hause Microsoft angewendeten Strategien aufzeigen und beschreiben.

4.2.1 Risikoanalyse anhand von Code Komplexität nach dem Paretoprinzip

Italian economist Vilfredo Pareto created a formula to describe the irregular distribution of wealth in his country, observing that 80 percent of the wealth belonged to 20 percent of the people. Many people believe that the Pareto principle (the 80:20 rule) applies as well to software projects.
Applied generally, the Pareto principle states that in many measurements, 80 percent of the results come from 20 percent auf the causes. When applied to software, the Pareto principle can mean that 80 percent of the users will use 20 percent of the functionality, that 80 percent of the bugs will be in 20 percent of the product, or that 80 percent of the execution time occurs in 20 percent of the code. (Page et al. 2009, S. 145–146)

In Anwendung auf die ews-java-api und die Risikoanalyse bedeutet dies herauszufinden, welche Teile der Software die 20 Prozent der genutzten Funktionen beinhalten, um hier an prominenter Stelle auch im Bereich des Legacy Codes einen Ansatzpunkt zu haben, an dem einzelne Teststrategien angewendet werden können. Page et al. Deklariert hierbei ebenfalls die Gefahr, dass bei Anwendung dieses Prinzips jedoch außer Acht gelassen wird, dass ebenfalls Kunden existieren, die Funktionalitäten nutzen, die nicht innerhalb dieses Rahmens abgedeckt werden.
The other side of the risk in this approach is that this approach relies on accurately determining where the majority of of the testing effort should be and misses the fact that there will be customers who will use the features and code paths that fall outside of the 20 percent range. (Page et al. 2009, S. 146)

4.2.2 Das Meilensteinmodel

Projektspezifische Meilensteine in der Softwareentwicklung zu definieren wird auch im Hinblick auf die Anwendung agiler Methoden und Prozesse in der Informatik und in der Projektarbeit in diversen Unternehmen bereits angewandt.
Interessant ist in diesem Zusammenhang die Verknüpfung von Meilensteinen über die fachlichen Vorgaben hinaus zu einer Definition von Qualitätskriterien, die ebenfalls einen Meilenstein bilden. In diesem Bereich hat Microsoft eigene Exit Kriterien definiert, in dessen Rahmen die Einführung von Qualitätsmeilensteinen betrieben werden.
Je nach Reifegrad des Projektes können somit Projekte einen höheren Meilenstein erreichen und sich damit in unterschiedlichen Qualitätsstufen befinden, die durch vorhandene Kriterien definiert werden. Vordefinierte Kriterien der einzelnen Stufen beinhalten meist folgende Ausprägungen:

1. Umsetzungsstand der Anforderungen
2. Ziele aus dem Bereich Testabdeckung oder Umsetzungsstatus einzelner Tests
3. Umsetzungsstand von als wichtig definierten Fehlern (z.B. Blocker)
4. Anwendung nichtfunktionaler Kriterien (z.B. Performance- oder Stresstests)

Der Erfüllungsgrad der einzelnen Kriterien wird hierbei jeweils höher, wenn in einen nächsthöheren Meilenstein gewechselt wird, bis das Projekt letztlich den notwendigen Status zur Publizierung eines Releases aufweist. Ein weiterer Vorteil dieses iterativen Verfahrens kann es ebenfalls sein, dass sich das Team im Rahmen der Erfüllung der jeweiligen Kriterien über den jeweiligen Projektstatus austauschen muss, um möglichen Problemen vorzubeugen und ebenfalls die Bewertung eingehender Fehler für die Zielerfüllung geschärft wird. (Page et al. 2009, S. 45–48)

4.2.3 Das Pestizid Paradoxon

Zur Veranschaulichung des Pestizid Paradoxons nutzt Allan Page in seinem Werk über den Testprozess bei Microsoft ein Gleichnis in Bezug auf die Gartenarbeit und den Einsatz von Pestiziden gegen die Verbreitung von Schnecken. Hierin wird veranschaulicht, dass Bier, Asche, Eierschalen und Salz gegen einen möglichen Schneckenbefall im heimischen Garten eingesetzt werden können.
Eine Maßnahme alleine ist jedoch meistens nicht ausreichend, um alle Schneckenarten abzuwehren. Ähnlich dem Einsatz im Garten stehen auch in Bezug auf den Softwaretest unterschiedliche Maßnahmen zur Verfügung und erfahrene Tester würden hierbei bestätigen, dass eine einzelne Maßnahme nicht gegen alle Fehlertypen Wirkung zeigen kann.
Dieses Dilemma ist auch als das Pestizid Paradoxon bekannt und wurde durch Beizers´s First Law abgeleitet: “Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual“. Es müssen also mehrere Methoden angewendet werden, um die Effektivität des Tests zu steigern und eine ausreichende Wirkung zu erzielen. (Page et al. 2009, S. 77)

4.2.4 Der „Happy path“ sollte immer funktionieren

Eine weitere Beschreibung bezieht sich auf den so bezeichneten „Happy Path“. Dieser wurde durch die Einführung eines neuen Features beschrieben, welches durch etwaige Tester getestet werden sollte.
Leider veranschaulicht die Erzählung, dass kurz vor Ende der Implementierung des Features durch den Softwareentwickler eine weitere Änderung erstellt wurde, welche auch den einfachen Test der Funktion unmöglich machte.
In diesem Zusammenhang wird von einem Standardtest mit simplen Eingaben gesprochen, welcher vor Änderungen getestet werden sollte, damit die Funktion des zugrundeliegenden Testobjektes noch gegeben bleibt. Dies vor allem vor dem Hintergrund, dass durch die jeweiligen Testabteilungen ein Fehlverhalten meist zuerst in der Konfiguration erwartet wird. (Page et al. 2009, S. 69–70)

Die Anspielung hierbei auf einen Regressionstest ist unübersehbar, da somit durch einen entsprechenden Test der grundlegenden Funktionalität die Fehleranfälligkeit in diesem Bereich reduziert werden kann.
Dies kann zum einen Kosten einsparen und ebenfalls Frustration ersparen, da ein untestbares Testobjekt zur Übergabe in z.B. eine andere Abteilung oder einen anderen Entwickler meist nicht gerne gesehen wird.

5 Anwendung einzelner Strategien zur Qualitätsverbesserung

5.1 Priorisierung einzelner Klassen der API

Der folgende Abschnitt soll eine Einordnung darüber bringen, an welchen Stellen der API ein Einstieg in den Test am besten erfolgen kann.
Dies wird auf der Grundlage einer zyklomatischen Komplexitätsanalyse für die einzelnen Klassen der API durchgeführt.

Die zyklomatische Komplexitätsanalyse dient als Metrik, um die linear unabhängigen Pfade durch einen zugrundeliegenden Programmcode zu ermitteln. Mathematisch wird die zyklomatische Komplexität ebenfalls als Ausprägung des Kontrollflussgraphen dargestellt.
Somit hätte also ein einfaches Codesegment ohne eine Menge, an den Kontrollfluss des Programms manipulierenden Anweisungen, eine Komplexität von 1. Mit einer einfachen If-Anweisung würde die Komplexität auf 2 ansteigen. Hierbei würde jeweils die Ausführung unter der Bedingung, dass die If-Anweisung wahr wäre und einmal, dass die If-Anweisung falsch wäre, berücksichtigt. (Laird und Brennan 2010, S. 58–62)

In Tabelle 6 werden die anhand einer Analyse ermittelten Ergebnisse, der pro Klasse akkumulierten zyklomatischen Komplexität dargestellt.
Das Analyseergebnis wurde aus Übersichtsgründen auf die sieben Klassen mit der höchsten Komplexität reduziert und mit dem IDE plugin MetricsReloaded durchgeführt.

Der berechnete Wert wird jeweils in der Spalte WMC (Weighted Methods per Class) dargestellt und dient dazu über die jeweilige Klasse eine mögliche Aussage ihrer Komplexitäten zu liefern.
Hier kann im Rahmen einer Priorisierung beurteilt werden, ob die entstehende hohe Korrelation von Komplexität ebenfalls für die Priorisierung innerhalb des Tests Nutzen finden kann.

Package (microsoft.exchange.webservices.data.XXX)	Class	WMC
Core	ExchangeService	226
complex	RulePredicates	146
Core	EwsUtilities	139
Autodiscover	AutodiscoverService	135
Core	EwsXmlReader	127
autodiscover.configuration.outlook	OutlookProtocol	105
core.service.item	Appointment	100

Tabelle 6 Analyseergebnis WMC nach Klassen

Wie in der oben aufgeführten Tabelle ersichtlich, hat die Klasse ExchangeService die mit Abstand höchste Komplexität.
Dies mag zunächst verwundern. Betrachtet man jedoch das der API zugrundeliegende Design einmal genauer (siehe Abbildung 9), fällt auf, dass hier ein Großteil der zu erwarteten Logik gebündelt ist. Somit wurden viele, die Geschäftslogik der API betreffende Prozesse innerhalb dieser Klasse programmiert.
Dies könnte ebenfalls eine Andeutung des im Kapitel 4 vorgestellten Paretoprinzips sein, da eine derartige Komplexität in einer der zentralsten Klassen der API ebenfalls auf eine hohe Fehleranfälligkeit dieses Bereichs hindeuten kann.
Die entstehende Problematik wird ebenfalls durch den Umstand bestärkt, dass für entsprechende Änderungen innerhalb dieser Klasse nur sehr wenig Tests im Sinne eines Regressionstests vorhanden sind.

5.2 Definition von Meilensteinen zum Soll-Zustand

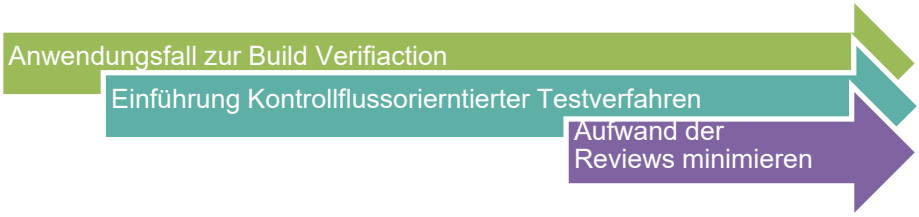

Abbildung 8 Meilensteine zum Soll-Zustand

Im Abbildung 8 können die jeweils groben Meilensteine abgelesen werden, die eingangs in dem Soll-Zustand beschrieben worden sind. Die einzelnen übergeordneten Meilensteine lassen sich hierbei wie folgt in einzelne kleinteiligere Aufgaben zerlegen:

Anwendungsfall zur Build Verfifiaction:

1. Ermittlung eines möglichen Szenarios
2. Einbindung der ggf. benötigten neuen Tools/Frameworks
3. Absicherung des Szenarios durch einzelne Unit-Tests
4. Automatisierte Ausführung der jeweiligen Tests
5. Absicherung des Szenarios durch einzelne Integrationstests
6. Automatisierte Ausführung der jeweiligen Tests

Einführung/Ausbau kontrollflussorientierter Testverfahren

1. Kategorisierung einzelner Klassen zum Definieren eines Startpunkts
2. Erhöhung der Testabdeckung durch Einführung weiterer Unit-Tests
3. Anweisungsüberdeckung von 100% bei Änderungen und Neuimplementierungen
4. Sinnvolle Pfad- und Zweigüberdeckung bei Änderungen und Neuimplementierungen
5. Sukzessive Erhöhung des Überdeckungsgrades für bereits bestehende Klassen
6. Einordnung der Tests anhand der Ausführungszeit und benötigten Ressourcen zur Automatisierung

Aufwand der manuellen Reviews minimieren

1. Problemsituationen identifizieren
2. Suchen von geeigneten Automatisierungsmethodiken
3. Anwendung neuer Tools zur automatisierten Abdeckung

5.3 Ermittlung eines kritischen Anwendungsfalls zur Build Verfification

Da die ews-java-api eine Vielzahl von Anwendungsfällen im Rahmen der abgebildeten Logik abdeckt, ist für die im Rahmen der exemplarischen Auswertung notwendigen Anpassungen eine Auswahl eines typischen Anwendungsfalls von Vorteil, der ebenfalls im Sinne eines „Happy Paths“ angesehen werden kann (siehe Abschnitt 4.2.4).
Der im folgenden Abschnitt dargestellte Anwendungsfall kann also ebenfalls als kritischer Pfad durch die Anwendung und somit zur Identifizierung eines erfolgreichen Build dienen. Da zum gegenwärtigen Zeitpunkt keine Evaluierung oder Befragung des Anwenderkreises stattgefunden hat, geht diese Arbeit davon aus, dass die Möglichkeit eine E-Mail über die API zu versenden einen wesentlichen Bestandteil zur Funktionsfähigkeit der API beiträgt. Dies deckt sich ebenfalls mit einer Nutzung des Funktionsumfangs eines deutschen Versicherungsunternehmens in dessen Projekten die API zum Einsatz kommt. Im Sinne einer Build Verification kann also der ausgewiesene Anwendungsfall zur Veranschaulichung der jeweiligen Teststrategien genutzt werden. In der folgenden Abbildung wird das aktuelle Design der API für die zur Erstellung einer E-Mail benötigten Klassen dargestellt.

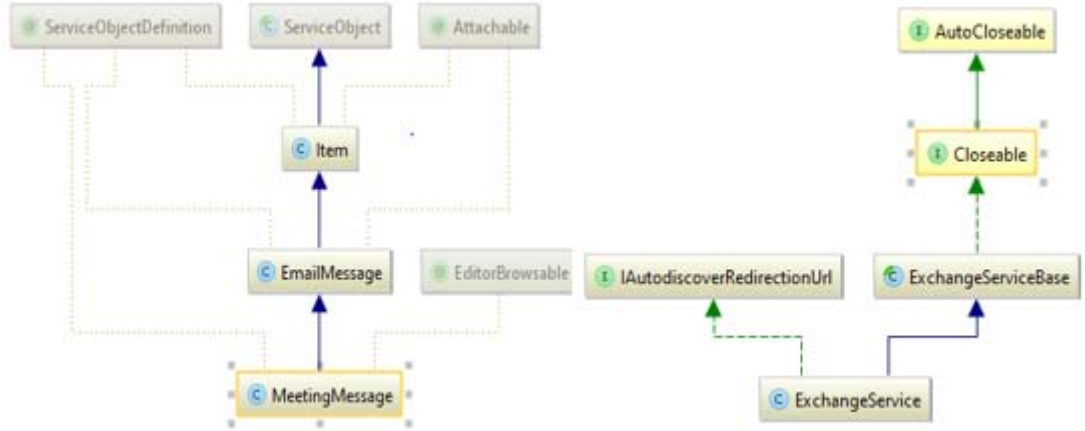

Abbildung 9 Übersicht abhängige Klassen des ExchangeService

Es wird zum einen eine Instanz des ExchangeServices benötigt. Hier werden Informationen zum Verhalten des Loggings, benötigte Authentifizierungsdaten und die jeweilige Exchange-Url hinterlegt. Des Weiteren werden die E-Mail-Daten wie im folgenden Beispiel beschrieben gefüllt.

```
// Erzeugen des ExchangeServices
ExchangeService service = new ExchangeService(ExchangeVersion.Exchange2010_SP1);
service.setTraceEnabled(true);
service.setUrl("<URL>");
ExchangeCredentials credentials = new WebCredentials("USERNAME", "PASSWORD", "DOMAIN");
service.setCredentials(credentials);

// Erzeugen und Absenden einer E-Mail
EmailMessage msg= new EmailMessage(service);
msg.setSubject("Hello world!");
msg.setBody(MessageBody.getMessageBodyFromText("Sent using the EWS Java API."));
msg.getToRecipients().add("someone@contoso.com");
msg.send();
```

5.4 Kriterien zur Auswahl von Strategien

In Kapitel 4 dieser Arbeit wurden einzelne Strategien aus großen, beispielhaften Unternehmen genannt. Die Auswahl der einzelnen Strategien erfolgte hierbei aufgrund der zu erwartenden Anwendbarkeit innerhalb der ews-java-api und wird in Abschnitt 5.5 zur Anwendung gebracht.
Die Strategien können allerdings ebenfalls einen möglichen Mehrwert auch in Bezug auf andere Projekte darstellen, die große Teile an Legacy Code aufweisen, oder bei denen z.B. Anknüpfpunkte für die Erstellung möglicher Tests gefunden werden sollen. Die Einschätzung der jeweiligen Strategien erfolgte hierbei auf der subjektiven Bewertung des Autors anhand der folgenden Kriterien:

1. (Kriterium 1) Anwendbarkeit innerhalb des Kontextes der ews-java-api
2. (Kriterium 2) Allgemeingültige Anwendbarkeit
3. (Kriterium 3) Messbarkeit des Nutzens
4. (Kriterium 4) Anwendbarkeit der Maßnahmen in zeitlicher Hinsicht (kurz-, mittel- und langfristig)

Maßnahme	*Kriterium 1*	*Kriterium 2*	*Kriterium 3*	*Kriterium 4*
Google 1	Hoch	Hoch	Hoch	Mittelfristig
Google 2	Hoch	Mittel	Mittel/Hoch	Mittel- od. Langfristig
Google 3	Mittel	Mittel/Hoch	Hoch	Kurzfristig
Microsoft 1	Hoch	Hoch	Hoch	Kurzfristig
Microsoft 2	Hoch	Mittel/Hoch	Hoch	Kurzfristig
Microsoft 3	Mittel	Hoch	Mittel	Langfristig
Microsoft 4	Mittel	Hoch	Mittel/Hoch	Kurz- od. Mittelfristig

Tabelle 7 Anwendbarkeit der Maßnahmen

Wie in der oben aufgeführten Tabelle veranschaulicht, lassen sich die Maßnahmen anhand der Kriterien in einzelne Kategorien einordnen.
Hier wären zum einen die Maßnahmen, die sich z.B. für einen kurzfristigen Einsatz eignen.
Als Vertreter lässt sich hierbei die von Google beschriebene Testfallerstellung in 10 Minuten deklarieren, da hier innerhalb von kurzer Zeit ein zu erwartendes und ebenfalls messbares Ergebnis erzielt werden kann.
Die Eignung für die ews-java-api wurde als mittelmäßig bewertet, da für die API kein entsprechendes Team existiert, dass sich über einen kurzfristigen Zeitraum zu einem Brainstorming zusammensetzen kann.
Eine mögliche Umgehungslösung wäre z.B. die Abstimmung und Konzeption mittels kollaboritiver Tools (z.B. JIRA etc.) durchzuführen.

Ebenfalls in Bezug auf die Umsetzungsdauer als kurzfristig eingeordnet sind die Microsoft Strategien zur Ermittlung der Risikoanalyse (MS-Strategie 1), dass aufgestellte Meilensteinmodell (MS-Strategie 2) und die Ermittlung des so bezeichneten „Happy Paths“ (MS-Strategie 4).

Die Strategien 1 und 2 lassen sich hierbei in ihrer Gewichtung der Anwendbarkeit in Bezug auf die ews-java-api und ebenfalls andere Projekte gleichermaßen kategorisieren, da bei beiden Strategien ein gut dokumentierbares und ebenfalls unabhängiges Ergebnis erzielt werden kann, welches auch für die ews-java-api z.B. im Rahmen der Risikoanalyse oder zur Bestimmung weiterer Meilensteine einen erheblichen Nutzen für die zukünftige Projektarbeit bieten kann.
In Bezug auf Strategie 4 lässt sich sagen, dass je nach Grad der Automatisierung eine unterschiedliche Priorisierung dieser Strategie erfolgen sollte, da zum Ausgangspunkt der Strategie die kontinuierliche Prüfung des kritischen Anwendungsfalls gehört. Dieser kann entweder manuell oder automatisiert erfolgen und steuert somit ebenfalls zur Build-Verfification bei. Ebenfalls je nach Art der Anwendung kann die Ermittlung des „Happy Paths“ eventuell als schwierig angesehen werden, da Anwendungen existieren können, dessen Nutzerkreis zu seinem Anwendungsverhalten ggf. nicht befragt werden kann. Hier müssten entsprechende Anforderungen mittels Aussage des möglichen Fachbereichs oder anhand der Dokumentation entnommen werden. Dies kann unter Umständen zu einer zeitlichen Verlängerung dieser Maßnahme führen.

Auch zur mittelfristigen Anwendung lassen sich einzelne Strategien klassifizieren. So z.B. die Stellung von Anforderungen an die jeweiligen Tests zur Laufzeit (G-Strategie 1) oder die Aufteilung einzelner Tests in jeweilige, die Ausführungszeit betreffende Kategorien (G-Strategie 2).
Je nach Anzahl der bereits existieren Tests, muss hierbei mit einem unterschiedlichen Maß an zeitlichen Aufwänden gerechnet werden.
Aus diesem Grund sollte eine Anwendbarkeit auf die bereits implementierten Tests ggf. im Projekt geprüft werden.
Für die ews-java-api lässt sich allerdings verdeutlichen, dass aufgrund der Testsituation eine hohe Eignung dieser Strategien ausgewiesen werden kann.
Bei einer großen Vielzahl an Tests wäre ggf. auch möglich, die Anwendung der Kriterien als ersten Schritt nur auf neue oder zu wartende Tests zu beziehen.
Auch wäre eine Kombination mit anderen Strategien z.B. in Bezug auf die Risikoanalyse denkbar, in dessen Rahmen die Menge, der zu bearbeitenden Tests reduziert werden kann.

Im Bereich der langfristig anzuwendenden Strategien erscheint in der aktuellen Auswertung je nach Art des Projektes die Anwendung des Pestizit Paradoxons als sinnvoll (MS-Strategie 3).
Die jeweilige Eignung kann natürlich in Bezug auf den Projektfortschritt neu klassifiziert werden und sollte je nach Projekt individuell erfolgen.
Unter der Maßgabe, dass ein Projekt über sehr wenige, unterschiedliche Maßnahmen zur Qualitätssicherung verfügt, sind ggf. hohe Aufwand und andere Strategien nötig, um das Maßnahmenpaket zur besseren Absicherung des Projektes, zu erweitern.

Aus diesem Grund wurde ebenfalls die Gewichtung für das ews-java-api Projekt als mittelmäßig priorisiert. Hier stehen zwar bereits einzelne Maßnahmenpakete zur Verfügung und können ebenfalls angewendet werden, jedoch ist eine Ausweitung der Maßnahmen auch mit wesentlichen zeitlichen Kosten verbunden, weshalb die Anwendung dieser Strategie eher als sukzessive eingeordnet werden kann.
Der jeweilige Qualitätsgewinn kann hierbei nicht unbedingt an einzelnen Faktoren gemessen werden, wird sich aber bei richtiger und qualifizierter Anwendung in der allgemeinen Produktqualität wiederspiegeln und kann somit einen erheblichen Faktor in der Qualitätssicherung bilden.

5.5 Exemplarische Anwendung der ausgewählten Strategien

Einzelne Strategien, die im vorherigen Abschnitt dargstellt wurden, sind bereits an anderer Stelle dieser Arbeit zur Anwendung gekommen. So z.B. die Risikoanalyse nach dem Paretoprinzip, die zum Aufzeigen eines möglichen Ansatzpunktes der eingeführten Teststrategien zur Anwendung gekommen ist. Ebenfalls wurde im Rahmen eines musterhaften Anwendungsfalls ein möglicher „Happy Path“ durch die API veranschaulicht.
Da das Projekt ews-java-api zum gegenwärtigen Zeitpunkt über so wenig Tests verfügt, dass die exemplarische Anwendung einzelner Teststrategien nur in manchen Fällen durchgeführt werden kann, werden zur Veranschaulichung der jeweiligen Strategie ebenfalls exemplarische Beispiele herangezogen.

5.5.1 Strategie 1: Anforderungen an Tests zur Laufzeit

Die erste den Beschreibungen von Google entnommene Strategie beschäftigt sich mit einer generellen Einhaltung von Qualitätsvorgaben bezüglich des Schreibens von Tests.
Um eine Abgeschlossenheit der Tests zu gewährleisten, muss auf eventuelle Abhängigkeiten zwischen den Tests und deren Ausführungsreihenfolge verzichtet werden.
Ebenfalls wird die Unabhängigkeit der Laufzeitumgebung angesprochen. Folgendes Szenario auf Ebene eines Unittests würde also gegen beide aufgeführten Regeln verstoßen:

```
@Test
public void testWriteSomeStringToFile() throws IOException {
  final String filePath = "/test.file";
  final String toBeWritten = "Lorem ipsum";
  Files.write(Paths.get(filePath), toBeWritten.getBytes());
  Assert.assertTrue(Paths.get(filePath).toFile().exists());
}
@Test
public void testReadExpectedStringFromFile() throws IOException {
  final String filePath = "/test.file";
  final String toBeWritten = "Lorem ipsum";
  byte[] bytes = Files.readAllBytes(Paths.get(filePath));
  Assert.assertEquals(toBeWritten,new String(bytes));
}
```

Die beiden Tests verstoßen vor allem deshalb gegen die definierten Qualitätskriterien, da zum einen die Laufzeitumgebung verändert wird, was ebenfalls erhebliche Konsequenzen auf die parallele Ausführung der Tests hat.
Ebenfalls ist für die korrekte Ausführung der Tests die Ausführungsreihenfolge `testWriteSomeStringToFile()`, `testReadExpectedStringFromFile()` nötig, da andernfalls die entsprechende Datei zum Einlesen nicht vorhanden ist.

5.5.2 Strategie 2: Aufteilung der Tests in einzelne Kategorien

Die von Google ebenfalls eingeführte Strategie der Einteilung einzelner Tests anhand von im Vorfeld definierten Kriterien verfolgt unter anderem den psychologischen Aspekt der simpleren Darstellung.
Hierbei rückt in den Vordergrund, dass sich ein Entwickler im Vorfeld über benötigte Ressourcen eines Tests und deren Bedeutung für die jeweilige Ausführungsdauer weiterführende Gedanken macht.

Zusätzlich kann ebenfalls kontrolliert werden, ob ein Entwickler die unter Strategie 1 genannten Kriterien eingehalten hat, da z.B. die Ausführungsreihenfolge im Rahmen eines Tests bewusst verändert werden kann. Hierbei ist ebenfalls wichtig die, im Bereich der Grundlagen definierte Einstufung und Anwendung von Test Doubles verinnerlicht zu haben, da mit Hilfe der Stubs einzelne Bereiche mit vorhergehend definierten Werten versorgt werden können.
Dies wäre z.B. im Bereich der ews-java-api dann nötig, wenn Methoden, die einen Webrequest an den Exchange-Server auslösen, um beispielsweise E-Mails in Form einer Response vom Server zu transferieren.
Die vorliegende Antwort würde hier in XML-Kodiert sein und von der API zur Erstellung der benötigten Klassen mittels eines Parsers umgewandelt werden.
Im Rahmen von Mocks wäre es ebenfalls möglich, die Funktionalität in Bezug auf den garantierten Aufruf einer Methode zu überprüfen.

Im folgenden Beispiel ist jeweils ein Beispiel für die Benutzung eines Mocks und eines Stubs mit dem Framework mockito aufgeführt.

```
@Test
public void testMockAndStub() {
  // mocking
  List<String> mockedList = mock(List.class);
  mockedList.add("one");
  verify(mockedList).add("one");

  // stubbing
  List<String> stubbedList = mock(List.class);
  when(stubbedList.get(0)).thenReturn("stubbing");
  Assert.assertEquals("stubbing", stubbedList.get(0));
}
```

Ein weiterer Grund für die von Google erfolgte Einteilung der Tests in einzelne Kategorien liegt in dem Ziel, die Ausführungsdauer der automatisierten Testausführung möglichst gering zu halten und ebenfalls eine Transparenz darüber zu schaffen, welche Tests sinnvollerweises automatisiert ausgeführt werden und welche Tests einer manuellen Ausführung unterliegen.

5.5.3 Strategie 3: Erstellung eines Testkonzepts in 10 Minuten

Die Erstellung und Abstimmung eines weiterführenden Testkonzepts außerhalb der in dieser Arbeit bereits festgehaltenen Spezifikation in punkto Umfang, Qualitätskriterien und Vorgehensweise ist ein noch zur weiteren Ausführung festgehaltener Punkt im Rahmen der Qualitätssicherung.
Da es sich bezüglich der ews-java-api um ein globales Projektteam handelt, das sich aus den Beteiligten unterschiedlichster Nationen und Herkunft zusammensetzt, werden hier allerdings weitere Events angesetzt, die einer Konzepterstellung, die durch das Projektteam getragen wird unterstützend zur Seite stehen.
Zum Zeitpunkt der Erstellung dieser Arbeit wird die Planung eines Events zur Steigerung der implementierten Qualitätsmerkmale durch die Projektleitung der ews-java-api analysiert.
Das hierbei hervorzuhebende Ziel ist es, die Anzahl der am Projekt beteiligten Personen nachhaltig zu vergrößern und durch Planungsmaßnahmen im Team zu einer gemeinsamen und transparenten Vereinbarung über die im Bereich der Qualitätssicherung zu erbringenden Leistungen zu kommen.
Als möglicher Ansatzpunkt kann hier ebenfalls die nachträgliche Erstellung eines Testkonzepts sein.

5.5.4 Strategie 4: Risikoanalyse anhand von Code Komplexität nach dem Paretoprinzip

Eine beispielhafte Risikoanalyse nach dem Paretoprinzip kann den Abschnitt Priorisierung einzelner Klassen der API entnommen werden. Hier wurden anhand der zyklomatischen Komplexität Annahmen über das Potenzial von Fehleranfälligkeiten getroffen. Die in Tabelle 6 dargestellte Komplexität von 226 WMC ist hierbei über alle Methoden der Klasse summiert worden. Betrachtet man das der API zugrundeliegende Design mit dem menschlichen Auge, wird hierbei die durch die Komplexitätsanalyse hervorgehobenen Ergebnisse unterstützt.
Hieraus ergeben sich einzelne Probleme, da Änderungen nicht durch jeweilige Tests abgesichert werden. Somit besteht also die Möglichkeit, dass eine nachträgliche Änderung eine bereits behobene Fehlerursache wieder hervorruft. Langfristig kann hieraus das Ziel abgelesen werden, auch wie im Bereich der Meilensteine beschrieben, eine entsprechende Testabdeckung zu etablieren und Änderungen an der API nur dann zu akzeptieren, wenn die Änderungen durch Tests zumindest grundlegend abgesichert sind.

5.5.5 Strategie 5: Das Meilensteinmodel

Wie bereits im Abschnitt Definition von Meilensteinen zum Soll-Zustand deklariert, konnten unter Anwendung der Strategie einzelne Meilensteine extrahiert und mit konkreten Aufgaben verbunden werden, die innerhalb eines zeitlichen Rahmens zur Umsetzung kommen können.

Da die Umsetzung einer jeweiligen Maßnahme in Bezug auf die ews-java-api auf etwaige Contributions in diesem Bereich angewiesen ist, weil das Projekt über keine hauptamtlichen Entwickler verfügt, sollte der zeitliche Rahmen hier offengehalten werden.

5.5.6 Strategie 6: Das Pestizid Paradoxon

Bezieht man die Anwendung des Pestizid Paradoxons auf die Anhebung der Softwarequalität im Allgemeinen umfasst die ews-java-api bereits einige Mittel, die zur Qualitätssicherung eingesetzt werden. In diesem Zusammenhang existieren automatisierte Build-Verfahren, die nach jeder Änderung auf dem zentralen Repository unter anderem die Software zusammenstellen, die vorhandenen Test-Suites ausführen und dem Entwickler einen Statusbericht erzeugen. Um die gesammelten Informationen neben der Darstellung über den Build-Server nachhaltig an zentraler Stelle festzuhalten, kann das Build Skript z.B. derartig erweitert, dass die Information nach einem erfolgreichen Build auf der Webseite der ews-java-api veröffentlicht wird.
Die nachträglich aufgeführte Erweiterung des Skripts würde diese Änderung durchführen.

```
cd "$HOME/OfficeDev"
echo "[GH-PAGES] Cloning travis-user fork"
git clone -q --progress -b gh-pages --single-branch
https://${GITHUB_TOKEN}@github.com/$TRAVIS_REPO_SLUG.git gh-pages && cd gh-pages
git remote add upstream https://github.com/$TRAVIS_REPO_SLUG.git
echo "[GH-PAGES] Updating origin if needed"
git fetch -q --progress upstream gh-pages
git merge -q --progress upstream/gh-pages gh-pages
git push -fq --progress origin gh-pages

echo "[GH-PAGES] Applying local changes"
git rm -rf --ignore-unmatch "./docs/snapshots/$TRAVIS_BRANCH"
mkdir -m777 -v -p "./docs/snapshots/$TRAVIS_BRANCH"

# copy all the builded stuff into the snapshot dir
cp -Rf "$HOME/$TRAVIS_REPO_SLUG/target/site/*" "./docs/snapshots/$TRAVIS_BRANCH"

git add -A
echo "[GH-PAGES] Committing changes to local repository"
git commit --author "travis-ci <travis@travis-ci.org>" -m "[TRAVIS]Deploy mvn site
[ci skip] @ $TRAVIS_BUILD_NUMBER"
echo "[GH-PAGES] Pushing changes to origin gh-pages"
git push -fq --progress origin gh-pages
```

Im Sinne weiterer Maßnahmenpakete kann, wie im Abschnitt Definition von Meilensteinen zum Soll-Zustand beschrieben, auf noch weiterführende Testmaßnahmen zurückgegriffen werden. Denkbar wäre hier z.B. ein konkreter Integrationstest, der die API auch gegen einen konkreten Server testet. Ebenfalls könnten im Rahmen eines Last- und Performance Tests Bereiche der API auf mögliche Erfüllung nichtfunktionaler Anforderungen überprüft werden.

5.5.7 Strategie 7: Der „Happy path“ sollte immer funktionieren

Im Rahmen der Analyse wurde für die ews-java-api der Anwendungsfall eines Mailversands ausgewiesen. Um die Funktionalität dieses Pfades auf der Ebene des Unit-Tests absichern zu können sind einige Anpassungen am gegenwärtigen Design der API notwendig.
So wäre es für den Test als solches interessant, ob die Erstellung einer E-Mail und das abschließende Versenden ebenfalls in einem potenziellen Request an den Exchange-Server resultiert.
Da für den Versand von einer E-Mail Instanzen sowohl vom Typ ExchangeService, als auch von einer EmailMessage notwendig sind, sollte bei der Erstellung des Stubs zum Test der jeweiligen Funktionalität eine Instanz der zur weiteren Verarbeitung genutzten Requests mitgeliefert werden können. Hier könnte ebenfalls im Sinne einer Verifizierung mit Mocks der Aufruf der für den Versand notwendigen Methoden getestet werden, ohne dass hierbei ein Integrationstest unter Einbeziehung externer Systeme geschaffen werden muss.

6 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden neben Grundlagen zum Softwaretest und einer Vorstellung der ews-java-api ebenfalls Maßnahmen zur Qualitätssicherung in großen Unternehmen vorgestellt.
Diese Maßnahmen wurden in Bezug auf Ihrer Anwendbarkeit auf die API und andere Projekte und Projektteams durch den Verfasser dieser Arbeit analysiert und bewertet. Ebenfalls wurden die jeweiligen Strategien auch auf Basis der zugrundeliegenden API und unter Einbeziehung allgemeiner Beispiele in ihrer Anwendung erläutert.
Gerade in Hinblick auf eine extrem niedrige Testabdeckung, wie mit der ews-java-api verbunden, konnten somit einzelne Maßnahmen und Tools skizziert werden, die die eingangs definierten Qualitätsmerkmale der API unterstützen und nachhaltig verbessern können.

Es wurde ebenfalls veranschaulicht, dass eine komplette Umsetzung der einzelnen Strategien auch in Bezug auf entstehende Kosten problematisch sein kann, weshalb Möglichkeiten aufgezeigt wurden, Anknüpfungspunkte zur Anhebung der Testabdeckung zu finden.
Dies kann im Nachlauf dann problematisch sein, wenn die Kriterien nur unzureichend durch das stetig wechselnde Projektteam getragen werden.
Aus diesem Grund ist eine stetige Transparenz und Nachhaltigkeit bei der Umsetzung der Ziele besonders in einer Anfangsphase notwendig.
Hierbei kann die automatisierte Testausführung einen erheblichen Anteil leisten, indem z.B. Unittests und auch Tests in Bezug auf die Build Verification automatisiert ausgeführt werden.

Um mögliche Ergebnisse eines Builds schnell darstellen zu können wurde ebenfalls der Nutzen, Tests in unterschiedliche Kategorien einzuteilen veranschaulicht. Hierbei steht ebenfalls im Fokus, die benötigten Ressourcen eines Tests zu evaluieren, da diese eine Ausführung erheblich verlangsamen können.

Im Rahmen der zur Auswahl stehenden Tools wurde gezeigt, dass trotz der Vorgabe, keine Kosten zu verursachen, diverse Maßnahmenpakete umgesetzt werden können und die Unterstützung für Open-Source Projekte grundlegend gegeben ist. Auch wenn dies bedeutet, dass das Projektteam ggf. mehr Zeit zur Behebung einzelner Fehler aufbringen muss, da Support nur durch die Community erfolgen kann und ebenfalls ein zeitlicher Rahmen zur geeigneten Evaluierung einer Open-Source lösung aufgebracht werden muss.
Im Sinne eines Einsatzes dieser Strategie in Unternehmen sind hier unbedingt

gängige Unternehmensrichtlinien zu betrachten, da nicht alle Unternehmen einen Einsatz von Open-Source Software (ohne vorhergehende Prüfung und Genehmigung) erlauben.

Für die zukünftige Projektarbeit und Weiterführung, der in dieser Arbeit deklarierten Ziele, ist ebenfalls ein Community-Event geplant, in dessen Umfang der Grad der Testabdeckung nachhaltig durch gezielte Contributions erhöht wird.
Hierzu sind jedoch noch weitere Absprachen mit dem Microsoft Team notwendig, da ein solches Event auch auf weitere, von Microsoft erstellte Projekte angewendet werden kann. Wichtig ist in diesem Zusammenhang ebenfalls, den Aufwand und die Maßnahmen transparent an das Team zu kommunizieren, damit etwaige Aufgaben durch interessierte Projektteilnehmer bearbeitet werden können. Die vorgestellten Maßnahmen, den Anknüpfungspunkt bei Änderungen zu suchen oder Pakete anhand ihrer Komplexität zu gewichten, können hierbei ebenfalls unterstützen.

Bezüglich der Definition von Legacy Code konnte ebenfalls erkannt werden, dass trotz einer Vielzahl an vorhanden LoC (Lines of Code) und vorhandenen Klassen, die eine unterschiedliche Komplexität aufweisen, Maßnahmen zur Verfügung stehen, die Qualitätssicherung auch in solchen Projekten, wie der ews-java-api, gewährleisten können.

Im Sinne einer nachhaltigen Qualitätssicherung, auch im Hinblick auf mögliche Anpassungen an die Software im späteren Projektverlauf sollten geeignete Methodiken der Qualitätssicherung und des Softwaretests möglichst frühzeitig in das Projekt eingebunden werden, um nachträgliche Aufwände möglichst zu vermeiden. Denn obwohl geeignete Maßnahmen auch im Nachhinein auf das Projekt angewendet werden können, ist eine Einbindung und Konzipierung geeigneter Tools und Mathodiken zu einem frühen Projektverlauf ratsam, da diese mit wesentlich höherem Aufwand und somit Kosten verbunden sind.

7 Glossar

Contribution	
Pull-request	Pull requests let you tell others about changes you've pushed to a repository on GitHub. Once a pull request is sent, interested parties can review the set of changes, discuss potential modifications, and even push follow-up commits if necessary. (Using pull requests - User Documentation 2016)
Commit	git-commit - Record changes to the git repository. (Git - git-commit Documentation)
Branch	A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process discussed in Git Basics, the first module of this series. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project. (Git - Branches 2015)
Impediments	
Styleguide	Leitfaden für Fragen des Stils (Duden | Style-guide | Rechtschreibung, Bedeutung, Definition, Herkunft 2016)

8 Literaturverzeichnis

atlassian: gitflow-workflow. Maintenance Branches. Hg. v. https://www.atlassian.com. Online verfügbar unter https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow, zuletzt geprüft am 19.02.2016.

Bath, Graham; McKay, Judy (2010): Praxiswissen Softwaretest - Test Analyst und Technical Test Analyst. Aus- und Weiterbildung zum Certified Tester - Advanced Level nach ISTQB-Standard. 1. Aufl. Heidelberg: dpunkt-Verl.

Connextra User Story 2001: ConnextraStoryCard. Online verfügbar unter http://agilecoach.typepad.com/photos/connextra_user_story_2001/connextrastorycard.html, zuletzt geprüft am 19.02.2016.

Duden | Style-guide | Rechtschreibung, Bedeutung, Definition, Herkunft (2016). Online verfügbar unter http://www.duden.de/rechtschreibung/Styleguide, zuletzt aktualisiert am 25.01.2016, zuletzt geprüft am 19.02.2016.

Feathers, Michael C. (2009): Working effectively with legacy code. 10. print. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference (Robert C. Martin series).

Fowler, Martin (2006): TestDouble. Online verfügbar unter http://www.martinfowler.com/bliki/TestDouble.html, zuletzt aktualisiert am 11.11.2015, zuletzt geprüft am 19.02.2016.

Git - Branches (2015). Online verfügbar unter https://www.atlassian.com/git/tutorials/using-branches/, zuletzt aktualisiert am 30.10.2015, zuletzt geprüft am 19.02.2016.

Git - git-commit Documentation. Online verfügbar unter https://git-scm.com/docs/git-commit, zuletzt geprüft am 19.02.2016.

Github - ews-java-api commits (2014). Online verfügbar unter https://github.com/OfficeDev/ews-java-api/commit/45d1e90f100011ccdfccda9a5ab2d4aab58ca0b9, zuletzt geprüft am 19.02.2016.

github.com (2015): Github - Where software is built. Hg. v. github.com. Online verfügbar unter http://www.github.com, zuletzt geprüft am 19.02.2016.

Kleuker, Stephan (2013): Qualitätssicherung durch Softwaretests. Vorgehensweisen und Werkzeuge zum Test von Java-Programmen. Wiesbaden: Springer. Online verfügbar unter http://dx.doi.org/10.1007/978-3-8348-2068-6.

Laird, Linda M.; Brennan, M. Carol (2010): Software measurement and estimation. A practical approach. Hoboken, N.J: Wiley-Interscience (Quantitative software engineering series). Online verfügbar unter http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=158192.

Mathias Meyer (2014): The Travis CI Blog: Faster Builds with Container-Based Infrastructure and Docker. Hg. v. https://blog.travis-ci.com. Online verfügbar unter https://blog.travis-ci.com/2014-12-17-faster-builds-with-container-based-infrastructure/, zuletzt aktualisiert am 17.12.2015, zuletzt geprüft am 19.02.2016.

Meszaros, Gerard (2012): XUnit test patterns. Refactoring test code. 5. printing. Upper Saddle River, NJ: Addison-Wesley (The Addison-Wesley signature series).

Office Dev Center, Microsoft (Hg.) (2014): Explore the EWS Managed API 2.0. Online verfügbar unter https://msdn.microsoft.com/en-us/library/office/dd633710(v=exchg.80).aspx, zuletzt aktualisiert am 13.02.2014, zuletzt geprüft am 19.02.2016.

Osherove, Roy (2010): The art of Unit Testing. [lesbare, wartbare und zuverlässige Tests entwickeln ; Stubs, Mock-Objekte und automatisierte Frameworks ; Einsatz von.NET-Tools inkl. NUnit, Rhino Mocks und Typemock Isolator]. Dt. Ausg., 1. Aufl. Heidelberg: mitp.

Page, Alan; Johnston, Ken; Rollison, Bj (2009): How we test software at Microsoft. Redmond, Wash.: Microsoft (Best practices). Online verfügbar unter http://gbv.eblib.com/patron/FullRecord.aspx?p=488766.

Resig, John; Bibeault, Bear (2014): JavaScript. Geheimnisse eines JavaScript Ninjas. Heidelberg: Verlagsgruppe Hüthig Jehle Rehm (mitp Professional). Online verfügbar unter http://gbv.eblib.com/patron/FullRecord.aspx?p=1638966.

Riley, Tim (Hg.) (2010): Beautiful testing. 1. ed. Sebastopol, Calif: O'Reilly (Theory in practice). Online verfügbar unter http://proquest.tech.safaribooksonline.de/9780596806934.

Roitzsch, Erich H. Peter (2005): Analytische Softwarequalitätssicherung in Theorie und Praxis. [der Weg zur Software mit hoher Qualität durch statisches Prüfen, dynamisches Testen, formales Beweisen]. Münster: Verl.-Haus Monsenstein und Vannerdat (Edition Octopus).

Spillner, Andreas; Linz, Tilo (2012): Basiswissen Softwaretest. Aus- und Weiterbildung zum Certified Tester ; Foundation Level nach ISTQB-Standard. 5. überarb. und aktualisierte Aufl. Heidelberg: dpunkt Verl. (Safari Tech Books Online). Online verfügbar unter http://site.ebrary.com/lib/hamburg/Doc?id=10717979.

Tom Preston-Werner (2013): Semantic Versioning. Version 2.0.0. Online verfügbar unter http://semver.org/spec/v2.0.0.html, zuletzt aktualisiert am 18.06.2013, zuletzt geprüft am 19.02.2016.

Using pull requests - User Documentation (2016). Online verfügbar unter https://help.github.com/articles/using-pull-requests/, zuletzt aktualisiert am 24.01.2016, zuletzt geprüft am 19.02.2016.

Whittaker, James A.; Arbon, James; Carollo, Jeff (2012): How Google tests software. S.l.: Addison-Wesley Professional. Online verfügbar unter http://proquest.tech.safaribooksonline.de/9780132851572.

9 Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

*Hamburg, den _19.02.2016_* _____