

MASTERARBEIT

zur Erlangung des akademischen Grades
„Master of Science in Engineering“
im Studiengang Software Engineering

Eigenschaften von Automatisierungslösungen für funktionale Systemtests von React Webapplikationen am Beispiel des selbst entwickelten Frameworks WDIO-WORKFLO

Ausgeführt von: Florian Hilbinger, BSc
Personenkennzeichen: 1610299001

1. Begutachterin: Mag. Maria-Therese Teichmann
2. Begutachter: Ing. Gerhard Gruber

Elsbach, 01.05.2018

Eidesstattliche Erklärung

„Ich, als Autor / als Autorin und Urheber / Urheberin der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. Urheberrechtsgesetz idgF sowie Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt und Gedankengut jeglicher Art aus fremden sowie selbst verfassten Quellen zur Gänze zitiert habe. Ich bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Ort, Datum

Unterschrift

Kurzfassung

Funktionale Systemtests überprüfen die Einhaltung der Kundenanforderungen und leisten somit einen essentiellen Beitrag in der Qualitätssicherung von Softwareentwicklung. Jedoch bedürfen sowohl deren Vorbereitung als auch deren Ausführung eines beträchtlichen Aufwandes, was gerade kleinere Projekte und Firmen mit sehr limitierten Ressourcen vor Probleme stellen kann. Um Kosten zu reduzieren, kann die Ausführung funktionaler Systemtests automatisiert werden. Doch selbst die Erstellung und Wartung automatisierter funktionaler Systemtests ist immer noch recht aufwändig, was diese zu einer Langzeitinvestition macht. Eine Lösung, die die mit funktionalen Systemtests verbundenen Kosten senkt und somit auch die Zeit, innerhalb welcher sich diese amortisieren, verkürzt, ist daher sehr gefragt.

Im Rahmen dieser Arbeit werden die spezifischen Charakteristika automatischer funktionaler Systemtests für Webapplikationen in einer NodeJS Umgebung sowie deren Integration in das Qualitätsmanagement eines Softwareentwicklungsprojekts untersucht. Die erlangten Erkenntnisse fließen als Anforderungen in die Entwicklung eines Open-Source Frameworks mit dem Namen „WDIO-WORKFLO“ ein. Anhand dessen soll gezeigt werden, dass eine Testautomatisierungslösung, welche genau auf die besonderen Eigenschaften funktionaler Systemtests zugeschnitten ist, den Aufwand, der für die Erstellung und Wartung automatisierter funktionaler Systemtests sowie für die Auswertung deren Resultate benötigt wird, reduziert und zugleich die Qualität der Testfälle, als auch der Anforderungen, die durch diese überprüft werden, steigert.

Um die potenzielle Kostensenkung und Qualitätssteigerung zu beurteilen, werden die resultierende Architektur und Implementierung des Frameworks in allen Einzelheiten betrachtet und anhand dessen das Ausmaß bestimmt, in welchem das Framework die zuvor recherchierten, Framework-spezifischen Anforderungen umsetzt. Zudem wird durch den Einsatz des Frameworks in einer konkreten, React-basierten Webapplikation für Angebotsverwaltung und -erstellung, sowie innerhalb eines Fallbeispiels, die praktische Anwendung des Frameworks veranschaulicht.

Abschließend werden aufgrund empirischer Erkenntnisse Empfehlungen für die optimale Art der Durchführung automatisierter funktionaler Systemtests mithilfe des entwickelten Frameworks abgegeben.

Schlagwörter: Testautomatisierungsframework, Funktionale Systemtests, Selenium

Abstract

Functional system tests validate the compliance with customer requirements and therefore make an important contribution to the quality assurance of software development. However, their preparation and execution requires a considerable effort and might pose problems for smaller projects and companies that have very limited resources. In order to reduce costs, the execution of functional system tests can be automated. Yet, automated functional system tests are still both expensive to create and to maintain, making them a long-term investment. A solution that can lower the expenses associated with functional system tests and shorten the time until their automation pays off is therefore much sought-after.

This thesis examines the specific characteristics of automated functional system tests for web applications in a NodeJS environment and their integration into the quality management process of a software development project. The insights serve as requirements for a self-developed framework named “WDIO-WORKFLO”, aiming to prove that a test automation solution tailored to the very features of functional system tests reduces the effort required for creating and maintaining automated functional system tests and for evaluating their results while at the same time increasing the quality of both the test cases and the requirements they validate.

In order to judge the potential cost reductions and quality increases, the framework’s resulting architecture and implementation are observed in full detail and assessed against the previously researched, framework-specific requirements. Furthermore, the thesis illustrates the framework’s practical application through its deployment in a concrete React web application for proposal management and creation and in a case example.

Finally, built on empirical findings, this thesis makes recommendations on how to optimally use the developed framework for executing automated functional system tests.

Keywords: Test Automation Framework, Functional System Tests, Selenium

Danksagung

Der Autor möchte sich an dieser Stelle bei der Firma DocuMatrix und deren technischem Betreuer, Herrn Ing. Gerhard Gruber, für die Möglichkeit bedanken, das im Zuge dieser Arbeit geschaffene Testframework im Rahmen der funktionalen Systemtests einer Webanwendung für Angebotserstellung und -verwaltung einsetzen und auch während der Arbeitszeit weiterentwickeln zu können.

Zudem ergeht ein großer Dank an Frau Mag. Maria-Therese Teichmann und Herrn Dr. Martin Hasitschka für die Vermittlung einer breiten und fundierten Wissensbasis in deren Vorlesungen zu den Bereichen Softwaretest, Anforderungsgestaltung, Application Lifecycle Management, Softwarequalitätsmanagement und Softwarearchitektur.

Frau Mag. Teichmann sei zudem für die stets unterstützende Betreuung und hilfreiche Anregungen beim Verfassen dieser Arbeit ganz besonders gedankt.

Schließlich gilt ein spezieller Dank des Autors allen EntwicklerInnen der Open Source Community, und insbesondere jenen der Tools Selenium, Jasmine, WebdriverIO und Allure, ohne deren Leistungen das im Rahmen dieser Masterarbeit entwickelte Framework nie hätte existieren können.

Inhaltsverzeichnis

1	Einführung	9
1.1	Grundlagen funktionaler Systemtests	9
1.1.1	Einordnung und Ziele funktionaler Systemtests	9
1.1.2	Testfallentwurfsverfahren.....	13
1.2	Automatisierung funktionaler Systemtests	15
1.2.1	Motivation und Risiken automatisierter Testausführung	15
1.2.2	Vereinbarkeit mit Test Driven Development	18
1.2.3	Tools für funktionale Systemtests von Webapplikationen.....	19
1.3	Signifikanz automatisierter funktionaler Systemtests	21
1.4	Ziele und Zielgruppen dieser Arbeit	24
1.5	Vorarbeiten	26
2	Konzeption, Implementierung und Evaluierung des Frameworks	29
2.1	Methodischer Überblick	29
2.2	Analyse der erforderlichen Anforderungen.....	30
2.2.1	Eigenschaften von JavaScript und React-Webapplikationen	31
2.2.2	Grundzüge des Softwaretests.....	38
2.2.3	Merkmale funktionaler Systemtests	42
2.2.4	Allgemeine Anforderungen an Testwerkzeuge.....	45
2.2.5	Besonderheiten von Automatisierungslösungen	52
2.2.6	Gestaltung von Anforderungen	57
2.2.7	Traceability	65
2.2.8	Ansprüche an Reports	68
2.2.9	Design eines Software Frameworks.....	71
2.2.10	Softwarequalität gemäß ISO/IEC 25010:2011	81
2.3	Framework-Entwurf	85
2.3.1	Designentscheidungen und technische Bestandteile	85
2.3.2	Systemgrenzen und Zusammenspiel der Bestandteile	89
2.3.3	Architektur der Framework-Bausteine.....	92
2.4	Framework-Bausteine und Tool Customization.....	97
2.4.1	Specs	97
2.4.2	Manual Results.....	99
2.4.3	Testcases	100

2.4.4	Steps	102
2.4.5	Page Objects	104
2.4.6	Lists	121
2.4.7	Utility Functions	122
2.4.8	Reporters	123
2.4.9	Customization von WebDriverIO und dessen Dependencies	133
2.4.10	Commandline Interface	136
2.4.11	Konfiguration	141
2.4.12	Ordnerstruktur	142
2.5	Evaluierung des Frameworks	143
2.5.1	Evaluierungsstrategie und Deployment-Umgebung	143
2.5.2	Evaluierung des Bereichs JavaScript und React-Webapplikationen	145
2.5.3	Evaluierung des Bereichs Grundzüge des Softwaretests	148
2.5.4	Evaluierung des Bereichs Funktionale Systemtests	150
2.5.5	Evaluierung des Bereichs Testwerkzeuge	154
2.5.6	Evaluierung des Bereichs Testautomatisierungslösungen	158
2.5.7	Evaluierung des Bereichs Gestaltung von Anforderungen	161
2.5.8	Evaluierung des Bereichs Traceability	166
2.5.9	Evaluierung des Bereichs Reports	168
2.5.10	Evaluierung des Bereichs Framework Design	170
2.5.11	Evaluierung der Softwarequalität gemäß ISO/IEC 25010	177
2.5.12	Auswirkungen auf die Kosten der Testautomatisierung	185
3	Beurteilung und Schlussbetrachtung	191
3.1	Diskussion der Resultate der Framework-Evaluierung	191
3.1.1	Diskussion der Resultate im Testfallentwurf	192
3.1.2	Diskussion der Resultate in der Testfallentwicklung	192
3.1.3	Diskussion der Resultate im Bereich Testausführung	194
3.1.4	Diskussion der Resultate im Bereich Testanalyse	196
3.1.5	Diskussion des Einrichtens der Testumgebung	196
3.1.6	Unzureichend geklärte Fragestellungen	197
3.2	Empirische Empfehlungen	198
3.3	Resümee und Erfüllung der Aufgabenstellung	202
3.4	Ausblick und Schlussfolgerung	205

Literaturverzeichnis	208
Abbildungsverzeichnis	212
Tabellenverzeichnis	214

1 Einführung

1.1 Grundlagen funktionaler Systemtests

1.1.1 Einordnung und Ziele funktionaler Systemtests

Die Entwicklung von Software findet gemäß dem von Spillner et al. [1, pp. 40-41] beschriebenen allgemeinen V-Modell in mehreren Stufen statt, wie sich anhand von Abbildung 1 erkennen lässt:

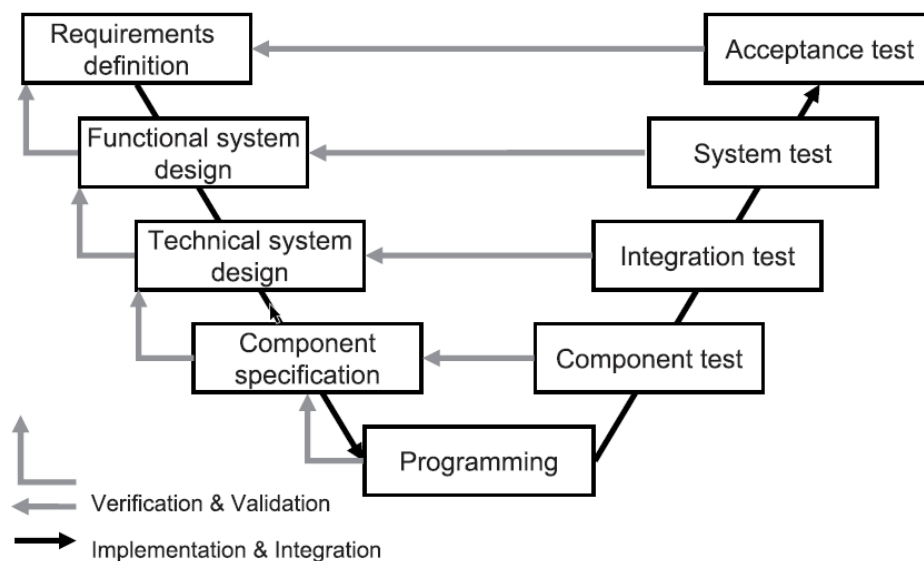


Abbildung 1: Allgemeines V-Modell [1, p. 40]

Zudem unterscheidet das allgemeine V-Modell in Entwicklungsaktivitäten auf der linken und Testaktivitäten auf der rechten Seite, wobei auf jeder Stufe des V-Modells der Entwicklungsaktivität eine entsprechende Testaktivität gegenübersteht.

Die Aktivitäten der Entwicklung werden auch als Implementierung bezeichnet. Ihre Aufgabe ist es, auf Basis der Anforderungen, die in der Sprache der Problemdomäne und damit auf der höchstmöglichen Abstraktionsebene formuliert sind, eine technische Lösung des beschriebenen Problems zu erarbeiten. Dafür werden in den unterschiedlichen Stufen des Entwicklungsprozesses zunehmend konkretere Phasenergebnisse erzeugt, bis schließlich der Quellcode als techniknächste Abbildung des ursprünglichen Problems erreicht ist.

Bei den Aktivitäten des Tests hingegen verhält es sich genau umgekehrt. So konzentriert sich die erste und techniknächste Teststufe auf die Korrektheit des Programmcodes isolierter Komponenten, während die zuletzt durchgeführten Akzeptanztests die Abnahme der Anforderungen in natürlicher Sprache umfassen. Da sich mit zunehmender Teststufe der Fokus der Betrachtung von einzelnen Bestandteilen hin zu immer größeren, zusammengefassten Einheiten verschiebt, spricht man hierbei von Integration.

Systemtests überprüfen als zweithöchste Stufe im V-Modell, ob das vollständig integrierte Gesamtsystem die spezifizierten funktionalen und nicht-funktionalen Anforderungen erfüllt. Somit liegt ein Fokus der Systemtests Sommerville [2, p. 219] zufolge darauf, sicherzustellen, dass sämtliche integrierte Komponenten miteinander kompatibel sind, korrekt miteinander agieren und die richtigen Daten zur richtigen Zeit über die Grenzen der Teilsysteme hinweg übertragen. Spillner et al. [1, p. 60] schreiben Systemtests darüber hinaus die Aufgabe zu, Fehler aufgrund unvollständiger, inkorrekt, inkonsistenter oder sogar fehlender Anforderungen aufzudecken.

Jede Stufe im allgemeinen V-Modell nimmt Überprüfungen auf zwei verschiedene Arten vor: Validierungen und Verifizierungen. Da das V-Modell sowohl funktionale als auch nicht-funktionale Systemtests auf einer Stufe zusammenfasst, sich diese Arbeit jedoch grundsätzlich auf funktionale Systemtests beschränkt, wird zur Erklärung von Validierung und Verifizierung eine etwas abgewandelte Form des allgemeinen V-Modells betrachtet, welche von Meyers et al [3, pp. 114-118] bevorzugt und detailliert beschrieben wird. Dieses abgewandelte V-Modell wird in Abbildung 2 dargestellt:

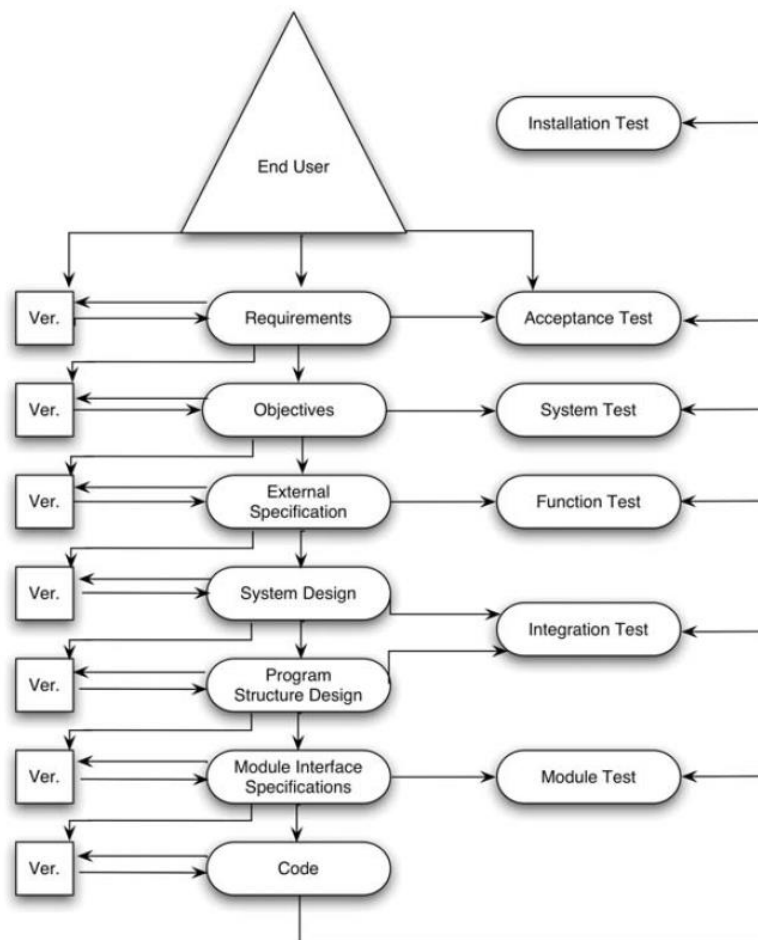


Abbildung 2: Abgewandelte Form des V-Modells von Meyers et al. (Quelle: [3, p. 117])

Im Unterschied zum allgemeinen V-Modell werden in der abgewandelten Variante in Abbildung 2 zusätzlich Funktionstests als eigene Stufe unterhalb der Systemtests eingeführt. Die Testbasis stellen im Falle der Akzeptanztests die Anforderungen, im Falle der Systemtests die Ziele und im Falle der Funktionstests die externe Spezifikation der Software dar. Meyers et al. [3, p. 115] beschreiben diese drei Spezifikationstypen wie folgt:

- Anforderungen definieren, warum ein Programm benötigt wird. Sie werden oft im Rahmen von Anwendungsfällen und User Stories formuliert. Im Fokus stehen hierbei die Bedürfnisse der unterschiedlichen Rollen von AnwenderInnen eines Programms.
- Ziele spezifizieren, was ein Program tun soll und wie gut es dies tun soll (hierbei spielen nicht-funktionale Aspekte wie Ladezeiten oder Sicherstandards eine große Rolle). Meist beschreiben Ziele auch die unterschiedlichen Stakeholder, die an deren Erfüllung Interesse haben, welche Einschränkungen dabei zu berücksichtigen sind und wie groß die erwarteten Vorteile durch die Zielerreichung ausfallen sollen.
- Externe Spezifikationen geben die exakte Repräsentation eines Programms seinen BenutzerInnen gegenüber vor – sie beschreiben dessen äußeres Erscheinungsbild. Der Name „externe Spezifikation“ leitet sich daraus her, dass Funktionstests ein Produkt als Blackbox betrachten und dieses nur über dessen externe Schnittstellen betesten. Häufig steht hierbei die graphische Benutzeroberfläche im Mittelpunkt, da diese oftmals die einzige Interaktionsmöglichkeit zwischen AnwenderInnen und System darstellt.

Nimmt man eine Software für die Erstellung und Verwaltung von geschäftlichen Angeboten als Beispiel, könnte eine Anforderung lauten, dass die BenutzerInnen mit der Rolle „Solution Designer“, welche viel Zeit mit der Erstellung und Bearbeitung von Angeboten verbringen, aufgrund der großen Anzahl an vorhandenen Angeboten das Bedürfnis haben, diese nach bestimmten Parametern zu durchsuchen und sich die Ergebnisse in einer Übersicht anzeigen lassen zu können.

Als Ziel könnte definiert werden, dass die Suche nach Angeboten über deren Namen und deren Angebotsnummer unterstützt werden soll und, dass die Ergebnisse nach maximal drei Sekunden angezeigt werden müssen. Die Abteilungsleiter erwarten sich durch die Suchmöglichkeit eine Zeitersparnis von zumindest 20 Sekunden pro bearbeitetem Angebot. In der externen Spezifikation könnte schließlich definiert werden, dass die Übersicht der gefundenen Angebote in Form einer Tabelle mit den Spalten Angebotsname, Angebotsnummer und Kundenname dargestellt werden soll und, dass oberhalb der Tabelle ein Eingabefeld zur Verfügung stehen muss, in welches die BenutzerInnen die gesuchten Begriffe eintragen und durch Drücken der Enter-Taste die Suche starten können.

Nachdem die externe Spezifikation als Testbasis von Funktionstests genauer betrachtet wurde, erfolgt nun wie bereits angesprochen eine Erklärung von Validierung und Verifikation.

Meyers et al. [3, p. 115] weisen darauf hin, dass die unterschiedlichen Stufen des Softwareentwicklungsprozesses eine Menge an Kommunikation, Verständnis und Übersetzung von Informationen in unterschiedliche Abstraktionsebenen beinhalten. Die meisten Fehler in Software stammen demzufolge aus der Art und Weise, wie mit Informationen umgegangen wird. Verifikation und Validierung stellen Maßnahmen dar, um diese Fehler möglichst zu vermeiden.

Die Verifikation hat das Ziel, Fehler, die während einer Entwicklungsphase auf der linken Seite des V-Modells entstehen, aufzudecken und zu beheben, bevor mit der Umsetzung der folgenden Implementierungsstufe begonnen wird. Somit sollen etwa Fehler, die beim Verfassen der externen Spezifikation eingebracht wurden, aufgedeckt werden, bevor mit dem Systementwurf begonnen wird. Dafür werden die vorläufigen Ergebnisse der externen Spezifikation mit der Ausgabe der vorigen Implementierungsstufe, den Zielen, verglichen und Widersprüche beziehungsweise deren Beseitigung in den Verfassungsprozess der externen Spezifikation rückeingespeist.

Bei der Validierung hingegen wird jede Testaktivität auf der rechten Seite des V-Modells daraufhin abgestimmt, dass diese Fehler der entsprechenden Entwicklungsaktivität auf der linken Seite offenbart. Es wird also gezielt nach einer bestimmten Art von Fehlern gesucht. So wird im Falle von Funktionstests daraufhin gearbeitet, Fehler in der Entwicklung nachzuweisen, die sich in einer Nicht-Einhaltung der externen Spezifikationen äußern.

Schließlich bietet sich für Funktionstests aufgrund deren Betrachtung des Systems als Blackbox an, diese durch ein eigenes Testteam und ohne Beteiligung der EntwicklerInnen und DesignerInnen durchführen zu lassen. Von dieser Möglichkeit wird gemäß Sommerville [2, p. 219] häufig Gebrauch gemacht.

Um späteren Verwirrungen vorzubeugen, soll zudem noch darauf hingewiesen werden, dass im weiteren Verlauf der Arbeit im Allgemeinen immer von „funktionalen Systemtests“ die Rede ist – ein Begriff, der zwar weitverbreitet, in Anbetracht der Unterscheidung zwischen Systemtests und Funktionstests jedoch etwas zweideutig besetzt ist. Es wird daher an dieser Stelle festgehalten, dass sich der Begriff „funktionale Systemtests“ im Zuge dieser Arbeit ausschließlich auf die Stufe der Funktionstests gemäß Meyers et al [3, pp. 114-118] und den Ausführungen dieses Kapitels bezieht.

1.1.2 Testfallentwurfsverfahren

Wie bereits erwähnt betrachten funktionale Systemtests die getestete Anwendung als Blackbox – sie wissen also nicht über die inneren Vorgänge der Software Bescheid. In dieser Hinsicht unterscheiden sie sich stark von low-level Tests wie Komponententests, für deren Durchführung eine genaue Kenntnis des inneren Aufbaus und der internen Abläufe einer Komponente erforderlich ist.

Diese Diskrepanz schlägt sich Spillner et al. [1, p. 58] zufolge auch in der Perspektive nieder, aus welcher die Testfälle entworfen werden. So ist dies im Falle von Komponententests und Integrationstests die technische Sichtweise der SoftwareentwicklerInnen, während funktionale Systemtests aus dem Blickwinkel zukünftiger AnwenderInnen betrachtet werden.

Die an den AnwenderInnen orientierte Perspektive und der Umstand, dass bei der Betrachtung als Blackbox nur die externen Schnittstellen des Systems getestet werden können, bestimmen auch die Art von Testfallentwurfsverfahren, die bei der Entwicklung funktionaler Systemtests zum Einsatz kommen.

Sommerville [2, pp. 219-220] berichtet etwa, dass häufig ein anwendungsfallbasierter oder geschäftsprozessorientierter Testfallentwurf eingesetzt werden. Hierbei werden typische Anwendungsszenarien eines Systems durch potenzielle BenutzerInnen nachgestellt. Da diese meist mehrere Teilbereiche oder Komponenten des Gesamtsystems betreffen, eignen sie sich sehr gut für funktionale Systemtests, denn die korrekte Interaktion aller Teilsysteme über die gesamte Anwendung hinweg kann dadurch sichergestellt werden.

Ebenso geeignet für funktionale Systemtests sind Sommerville [2, pp. 224-225] zufolge anforderungsbasierte Testfallentwurfsverfahren. Diese betrachten die einzelnen Anforderungen an das System und leiten daraus einen Satz von Testfällen ab. Im Falle von Funktionstests stellt die externe Spezifikation die Quelle dieser Anforderungen dar. Sehr wichtig ist es dabei, Traceability zwischen den Testfällen und den durch diese überprüften Anforderungen zu gewährleisten – also eine (beiderseitige) Zuordnung zu ermöglichen. Schließlich steht im Vordergrund der anforderungsbasierten Testfallentwurfsverfahren der Nachweis, dass die Anforderungen vom System korrekt umgesetzt werden, und nicht das Aufdecken neuer Fehler im System.

Franz [4, pp. 102, 42-46] erwähnt außerdem zustandsbasierte Testfallentwurfsverfahren als hilfreiches Mittel bei der Entwicklung funktionaler Systemtests. Dies erklärt sich dadurch, dass funktionale Systemtests im Regelfall die graphische Benutzeroberfläche einer Softwareanwendung betesten, welche im Grunde nichts anderes darstellt als einen Zustandsautomaten. Jede Interaktion mit der graphischen Benutzeroberfläche kann zu einer Veränderung des Systemzustandes führen und bildet somit einen Zustandsübergang ab.

Der Zustand des Systems wird aus der Summe aller in der graphischen Benutzeroberfläche dargestellten Informationen abgeleitet. Testfälle, die mithilfe von zustandsbasierten Testfallentwurfsverfahren entwickelt wurden, konzentrieren sich demgemäß auf die Überprüfung der Zustandsübergänge und der daraus resultierenden Folgezustände.

1.2 Automatisierung funktionaler Systemtests

1.2.1 Motivation und Risiken automatisierter Testausführung

In der Fachliteratur werden zahlreiche Gründe genannt, die für eine Automatisierung der Testausführung im Allgemeinen sprechen. Viele dieser Gründe legen auch und manche sogar insbesondere die Automatisierung funktionaler Systemtests nahe. Jedoch sind mit der Testautomatisierung auch gewisse Risiken verbunden. Dieses Kapitel bietet einen Überblick über die wichtigsten Gründe, die für eine Automatisierung funktionaler Systemtests sprechen und weist gleichzeitig auf die häufigsten damit verbundenen Risiken hin.

Spillner et al. [1, p. 205] sowie Dustin et al. [5] stimmen darin überein, dass die erhöhte Testeffizienz den wichtigsten Grund für eine Testautomatisierung darstellt. Da die Zeit und die Kosten, die mit dem Testen von Software verbunden sind, einen signifikanten Teil des gesamten Projektaufwands ausmachen, haben alle Maßnahmen, die sich in einer Verbesserung der Testeffizienz niederschlagen, einen messbaren Einfluss auf die Kosten des Projekts. Im Falle von funktionalen Systemtests können durch eine Automatisierung die Aufwände für repetitive und zeitaufwändige manuelle Aufgaben drastisch reduziert werden. Ein sehr markantes Beispiel hierfür stellen Regressionstests dar, welche überprüfen, ob durch Änderungen am Programmcode Fehler in die Software eingebracht wurden. Regressionstests sind sehr aufwändig, da sie die Software möglichst umfangreich überprüfen sollen und werden im Idealfall bei jeder Änderung an der Software erneut ausgeführt. Sommerville [2, p. 223] empfiehlt daher, Regressionstests zu automatisieren, um deren Verschleiß menschlicher Ressourcen drastisch zu reduzieren und geht in diesem Zusammenhang von einem enormen Einsparungspotenzial aus.

Einen weiteren Vorteil sehen Spillner et al. [1, pp. 220-221] sowie Dustin et al. [5] in der Möglichkeit, die Testabdeckung mittels Automatisierung zu erhöhen. Dies wird dadurch erreicht, dass mithilfe von automatisierten Tests wesentlich mehr Kombinationen an Datenvariationen und Testszenarios in derselben Zeit abgedeckt werden können als mit manuellen Tests, da die automatisierte Ausführung üblicherweise wesentlich zügiger erfolgt.

Automatisierte funktionale Systemtests können zudem gut als Smoke Tests eingesetzt werden. Diese sind laut Pressman [6, p. 479] dazu bestimmt, schwerwiegende Fehler, die den Erfolg eines Buildvorgangs der Software verhindern, frühzeitig zu erkennen. Smoke Tests müssen nicht vollumfänglich sein, sollen aber eine Auskunft darüber ermöglichen, ob der Build stabil genug ist, um ausgiebig getestet zu werden. Sie führen zu einer gesteigerten Softwarequalität, da funktionale und architektur-bezogene Defekte sowie Design-Fehler auf Komponentenebene früh aufgedeckt und korrigiert werden können.

Auch die verbesserte Zuverlässigkeit stellt ein wesentliches Kriterium für die Automatisierung funktionaler Systemtests dar. Dustin et al. [5] warnen vor der sogenannten „Tester fatigue“, die vor allem bei sehr monotonen, repetitiven und ermüdenden Testtätigkeiten wie Regressionstests häufig zu Tage tritt und die Quelle zahlreicher Fehler darstellt. „Tester fatigue“ entsteht, wenn manuelle TesterInnen aufgrund monotoner Aufgaben unaufmerksam werden oder nach einer mehrfachen Ausführung derselben Tests ein Gewöhnungseffekt eintritt, der dazu führt, dass TesterInnen nur mehr die positiven, funktionierenden Pfade im Programmablauf durchlaufen und die negativen, möglicherweise fehlerhaften Pfade vernachlässigen. Ein automatisiertes Testskript hingegen führt selbst monotone Schritte immer nach demselben, vorgegebenen Ablaufschema aus, beurteilt diese stets objektiv und kann der „Tester fatigue“ somit nicht zum Opfer fallen.

Zusätzlich verbessert eine Automatisierung Dustin et al. [5] zufolge die Dokumentation, Messbarkeit und Traceability funktionaler Systemtests. So bieten die Reihenfolge, die Eingaben und die Ergebnisse der einzelnen Steps, die im Verlauf der automatisierten Testausführung festgehalten werden, eine exzellente Dokumentationsgrundlage. Fehler können durch diese wesentlich einfacher reproduziert werden als mit manuellen Tests. Die durch Automatisierung gewährleistete Wiederholbarkeit wiederum erlaubt es, Qualitätsmetriken zuverlässig zu messen und für eine Optimierung der Tests, sowie für Managemententscheidungen, zu nutzen.

Schließlich sind manche Testarten überhaupt nur mittels automatisiert ausgeführter Tests möglich. Als Beispiele hierfür nennen Dustin et al. [5] Performance Testing, Memory Leak Detection oder Concurrency Testing.

Wie bereits erwähnt existieren jedoch auch Risiken, die im Zusammenhang mit Testautomatisierung berücksichtigt werden müssen. Beispielsweise warnen Spillner et al. [1, pp. 220-221] davor, die Kosten und den Aufwand für die Einführung eines Testautomatisierungswerkzeuges zu unterschätzen. So muss zunächst einige Zeit für die Recherche und Auswahl eines geeigneten Werkzeugs eingeplant werden, und auch die Schulung der MitarbeiterInnen in der Handhabung des Automatisierungswerkzeugs muss miteinkalkuliert werden. Zudem ist es erforderlich, initial viel Aufwand in den Aufbau automatisierter Testfälle zu investieren. Wie lange es dauert, bis sich dieser Aufwand aus wirtschaftlicher Sicht rentiert, kann meist nicht genau im Vorhinein abgeschätzt werden. Um den „Breakeven“-Zeitpunkt jedoch möglichst früh zu erreichen, ist es in jedem Fall wichtig die Testfälle so zu entwickeln, dass diese einfach im Rahmen von Regressionstests eingesetzt werden können und leicht zu warten sind.

Dustin et al. [7] widmen den Risiken der Testautomatisierung ein eigenes Kapitel. Dessen Inhalte dienen als Grundlage für die Beschreibung der folgenden Risikofaktoren.

Ein häufig gemachter Fehler in der Automatisierung funktionaler Systemtests ist es demgemäß, die Automatisierung von Komponententests, welche zeitlich vor den Systemtests ausgeführt werden, zu vernachlässigen. Dies ist problematisch, weil Komponententests vergleichsweise wesentlich billiger zu erstellen und auszuführen sind und durch diese aufgedeckte Fehler einfacher isoliert, früher entdeckt und damit billiger behoben werden können.

Auch ist es gefährlich davon auszugehen, dass die Automatisierung von Testfällen ohne Weiteres von manuellen TesterInnen vorgenommen werden kann. Diese erfordert nämlich zusätzliche Fähigkeiten, über welche manuelle TesterInnen oft nicht verfügen. Dazu gehören an erster Stelle Kenntnisse in der Softwareentwicklung.

Zuletzt soll auch darauf hingewiesen werden, dass selbst trotz des erhöhten Umfangs und der gesteigerten Tiefe automatisierter Tests eine hundertprozentige automatisierte Testabdeckung entweder technisch nicht möglich oder wirtschaftlich nicht sinnvoll ist. Vor allem für Testfälle, die nur selten wiederholt werden, lohnt sich der Aufwand einer Automatisierung nicht und manche Testaktivitäten, wie etwa das Verifizieren eines ausgedruckten Dokuments, sind schlicht und einfach nur manuell durchführbar.

1.2.2 Vereinbarkeit mit Test Driven Development

Im Zuge von low-level Tests wie Komponententests und Integrationstests wird häufig das Konzept von Test Driven Development angewandt, bei dem die Testfälle vor der eigentlichen Implementierung geschrieben werden. Dies ist jedoch bei automatisierten funktionalen Systemtests kaum möglich, da diese, wie Sommerville [2, p. 223] in Erinnerung ruft, prüfen, ob das fertig integrierte System die gestellten Anforderungen erfüllt.

Jedoch kann im Sinne von Behavior Driven Development gemäß Emrich [8, p. 70] die Entwicklung automatisierter funktionaler Systemtests in die Formulierung der Specs (formalisierte Anforderungen) als logische und die Implementierung der eigentlichen Tests als konkrete Testfälle getrennt werden. Die Specs können nun bereits vor der Entwicklung der Software selbst definiert werden, den EntwicklerInnen somit als Hilfe und Vorlage dienen und Widersprüche in den Anforderungen bereits im Vorhinein offenlegen. Die Specs bilden eine Beschreibung ausführbarer Akzeptanzkriterien ab, enthalten jedoch noch nicht die konkrete Implementierung der Testlogik. Diese wird erst im Anschluss an die Entwicklung der Software umgesetzt.

1.2.3 Tools für funktionale Systemtests von Webapplikationen

Für die Automatisierung funktionaler Systemtests existieren am Markt einige kostenpflichtige Lösungen. Je nach Preispunkt kombinieren diese die reine Testautomatisierung zusätzlich mit der Verwaltung und Modellierung von Anforderungen, stellen teils ausgiebige Reporting- und Integrationsmöglichkeiten bereit und unterstützen auch andere Testarten wie etwa Komponenten- oder Performance-Tests. Viele dieser Lösungen begleiten auch die Ausführung manueller funktionaler Systemtests und unterstützen den Einsatz von „Capture and Replay“ Methoden.

So ist etwa „Ranorex“ [9] ein Testwerkzeug, das neben der automatischen Erkennung von GUI Objekten wie Buttons und Input-Feldern das Recording manueller Testfälle und die Generierung visuell ansprechender Testreports umfasst und sowohl mit Web Applikationen als auch mobilen und Desktop Anwendungen zusammenarbeitet.

Zudem ermöglicht Ranorex die Entwicklung von Testfällen mit datengetriebenen und schlüsselwortgetriebenen Ansätzen:

Beim datengetriebenen Ansatz werden die Testkommandos von den Testdaten getrennt, wodurch dieselben Testabläufe mit einer Vielzahl an unterschiedlichen Eingabedaten aufgerufen werden können. Um Testfälle darüber hinaus so zu beschreiben, dass diese von den verschiedensten Projektbeteiligten verstanden werden können, wird mittels Schlüsselwörtern wie „login“ oder „save“ eine zusätzliche Abstraktionsschicht eingeführt, die die technischen Testbefehle hinter natürlicher Sprache kapselt.

Ein Konkurrenzprodukt mit sehr ähnlichen Fähigkeiten wie Ranorex ist „TestComplete“ [10] von der Firma SmartBear. Zusätzlich verfügt dieses über die Möglichkeit, Testläufe per Video aufzuzeichnen und funktionale Tests in Performance Tests zu konvertieren. Außerdem stellt TestComplete ein Plugin-System zur Verfügung, um das Testwerkzeug mit zusätzlichen Funktionalitäten zu erweitern.

Noch einen Schritt weiter geht „Tosca“ [11] von der Firma Tricentis. Dieses Produkt ermöglicht eine modellbasierte Testautomatisierung. Dabei scannt Tosca die getestete Anwendung und unterstützt TesterInnen auf Basis der gesammelten Daten bei der Entwicklung von Testfällen. Alle Testartefakte, technische Definitionen und Testsequenzen werden in einem einzigen, redundanzfreien Repository gespeichert. Die von Tosca erstellten Modelle sind dynamisch und werden mit der getesteten Anwendung synchronisiert, um Änderungen an dieser in den Tests widerzuspiegeln und Wartungsaufwände so gering wie möglich zu halten.

Während alle eben beschriebenen Lösungen einen wichtigen Beitrag zu einer effizienten und umfangreichen Qualitätssicherung von Software leisten können, haben sie allesamt einen entscheidenden Nachteil: Jede von ihnen ist kostenpflichtig.

Für größere Firmen und Softwareprojekte mit einem umfangreichen Budget mag dies keine Hürde darstellen. Allerdings können die Kosten von Qualitätssicherungsmaßnahmen gerade für kleinere Software-Betriebe und Projekte mit knapp bemessenen Budgets eine Herausforderung bedeuten. Zudem begeben sich die Kunden von kostenpflichtigen Werkzeugen zur Testautomatisierung in eine gewisse Rolle der Abhängigkeit gegenüber den Herstellern dieser Tools. Jahrelang aufgebaute Testsuiten könnten somit plötzlich ihre Daseinsberechtigung verlieren, wenn ein Testwerkzeug von Seiten des Herstellers nicht länger unterstützt oder vertrieben wird.

Aus diesen Gründen greifen viele Firmen bevorzugt auf Open Source Produkte zurück, um ihre Software zu testen. Im Bereich der funktionalen Systemtests ist etwa „Selenium“ [12] eine weitverbreitetes Testwerkzeug. Es handelt sich hierbei um eine Suite an Tools, die es erlaubt, eine Webanwendung in einem Browser automatisiert mit geskripteten Befehlen zu betesten. Selenium WebDriver ist eines dieser Tools und ermöglicht über die Accessibility API eines Browsers die Kontrolle über Webanwendungen, die in diesem laufen [13, p. 65]. Es ist mit den weitverbreitetsten Browsern und Betriebssystemen kompatibel und kann von vielen verschiedenen Programmiersprachen angesprochen werden.

Im Vergleich zu den kostenpflichtigen Testautomatisierungslösungen fehlen Selenium jedoch einige für funktionale Systemtests wichtige Features, wie etwa das Management von Anforderungen, die Fähigkeit, Reports zu erstellen oder die codeseitige Abbildung von Webseitenelementen in wiederverwendbaren, objektorientierten und gut wartbaren Komponenten.

1.3 Signifikanz automatisierter funktionaler Systemtests

Um den Stellenwert des Testens in Softwareprojekten zu beurteilen, beschreiben Dustin et al. [5] die Ergebnisse einer Umfrage, die von der Firma „Innovative Defense Technologies“ (IDT) [14], welche Software für das US-amerikanische Verteidigungsministerium entwickelt, durchgeführt wurde. Über 700 Testspezialisten beantworteten die Frage, wieviel Prozent des Gesamtaufwands eines Softwareprojekts Testaktivitäten ausmachen, wie folgt:

46 Prozent der Befragten gaben an, dass Testaktivitäten für 30 bis 50 Prozent, und 19 Prozent der Befragten, dass Testaktivitäten für 50 bis 75 Prozent des gesamten Projektaufwandes in Projekten, in denen sie beteiligt waren, verantwortlich waren.

Die Umfrageteilnehmer wurden auch nach den in ihren Augen größten Hindernissen für die Automatisierung von Tests befragt. 37 Prozent gaben hierbei einen Mangel an Zeit, und 17 Prozent fehlende budgetäre Mittel an.

Diese Ergebnisse stellen einerseits deutlich die signifikante Bedeutung des Testens in Softwareprojekten dar, welches nicht selten den, gemessen am Gesamtaufwand, größten Anteil im kompletten Projekt einnimmt. Darüber hinaus zeigt sich, dass trotz der hohen Wichtigkeit des Testens, diesem oft nicht genug Ressourcen eingeräumt werden. Dies ist umso besorgniserregender, als dass Dustin et al. [15] zusätzlich davon berichten, dass der Markt nach immer schneller ausgelieferten und billigeren Softwareprodukten verlangt, die zugleich immer zuverlässiger funktionieren sollen. In diesem Zusammenhang gewinnen Werkzeuge an Bedeutung, die die benötigten zeitlichen und finanziellen Mittel für die Durchführung von Softwaretests reduzieren können.

Die hohe monetäre Bedeutung des Testens in der Softwareentwicklung wird unter anderem durch die Aussage von Jones [16, p. 25] begründet, dass rund 50 Prozent der Kosten, die für eine Software während deren Lebenszeit anfallen, durch die Suche und die Behebung von Fehlern verursacht werden. Über 20 Jahre hinweg, bis ins Jahr 2008, untersuchte Jones [16, p. 257] zudem, welcher Anteil an der Gesamtarbeitszeit in einem Softwareprojekt durch die Entwicklung des Programmcodes, sowie durch unterschiedliche Testaktivitäten, vereinnahmt wurde. Es zeigte sich, dass, über verschiedenste Sparten der Softwareindustrie hinweg, die Aufwände für die Entwicklung und dynamische Tests in etwa gleichgesetzt werden konnten. So entfielen auf die Implementierung des Programmcodes 20 bis 30 Prozent, auf Systemtests 5 bis 7 Prozent, auf Funktionstests 5 bis 6 Prozent, auf Integrationstests rund 5 Prozent und auf Komponententests 3 bis 5 Prozent. Jones' Erkenntnisse fallen hinsichtlich des Ausmaßes von Softwaretests am Gesamtaufwand eines Projekts somit etwas niedriger aus, als die Ergebnisse der von IDT durchgeführten Studie. Insgesamt kommen aber auch diese auf einen Gesamtanteil des Testens von rund 20 bis 25 Prozent, wobei hier nur dynamische Tests und keine statischen Tests wie Reviews miteinbezogen wurden.

Im Jahr 2016 untersuchte eine Softwaretestumfrage [17, p. 14] der Hochschulen von Bremen, Bremerhaven und Köln in Zusammenarbeit mit dem German Testing Board und dem Swiss Testing Board, wie hoch die Automatisierungsraten der dynamischen Testarten im Vergleich zueinander ausfallen. Die Ergebnisse sind in Abbildung 3 zu sehen:

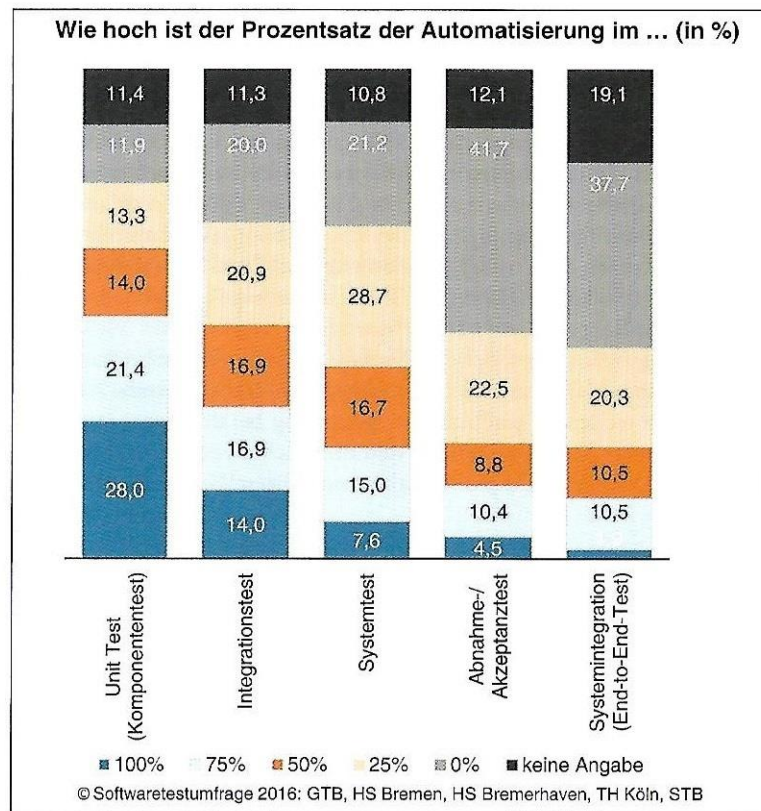


Abbildung 3: Automatisierungsraten dynamischer Testarten (Quelle: [17, p. 14])

Es stellt sich heraus, dass die Automatisierungsraten im Komponententest am höchsten ausfielen und mit zunehmender Teststufe immer weiter absanken. So gaben für den Komponententest 63,4 Prozent der Befragten eine Automatisierungsgrad von über 50 Prozent an. Bei den Systemtests fiel dieser Wert mit 38,7 Prozent wesentlich geringer aus. Dies liegt vermutlich daran, dass Systemtests im Allgemeinen aufwändiger zu automatisieren sind als Komponententests [2, p. 222].

Die Umfrage aus dem Jahr 2016 [17, p. 17] ergab zudem, dass alle Rollen eines Softwareprojekts durchgängig die Intensität des Testens als zu gering ansahen, während diese von kaum jemandem als übertrieben bewertet wurde, wie Abbildung 4 belegt:

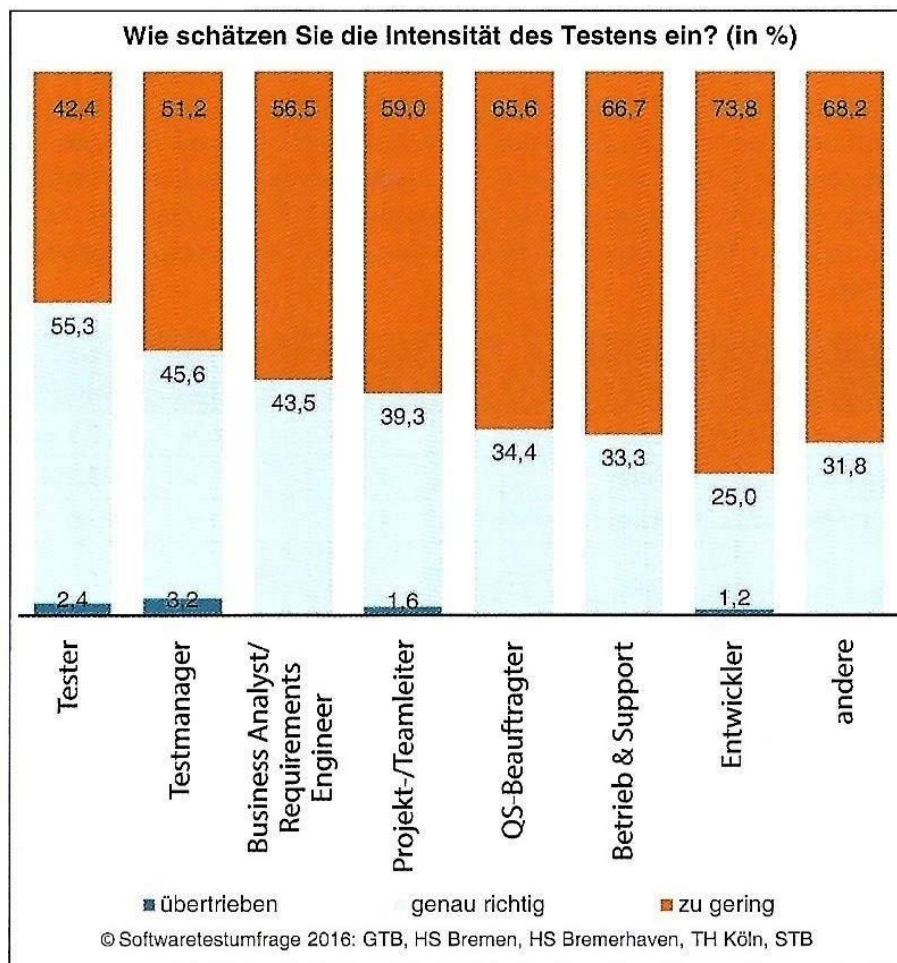


Abbildung 4: Einschätzung der Testintensität (Quelle: [17, p. 17])

Dies unterstreicht abermals die Forderung nach Testwerkzeugen, die in der Lage sind, das Ausmaß und die Qualität von Softwaretests zu steigern, während sie gleichzeitig die dafür benötigten Aufwände reduzieren.

Einen besonderen Stellenwert in der gesamten Softwarebranche nahmen in den letzten Jahren zunehmend Webapplikationen ein, wie Jones [16, p. 246] bestätigt. So konnte dieses Segment von 1995 bis 2007 ein Wachstum von rund 990 Prozent verzeichnen und stellt damit alle anderen Branchenbereiche mit einem durchschnittlichen Wachstum von 124 Prozent klar in den Schatten. Gerade aufgrund der großen Konkurrenz im Webbereich und der hohen Zugänglichkeit des Internets sprechen sich Meyers et al. [3, pp. 203, 211] daher für ein besonders gründliches Testen des Presentation Layers (der graphischen Oberfläche im Browser) von Webapplikationen aus, da die BenutzerInnen dieser Applikationen inzwischen sehr hohe Qualitätsansprüche entwickelt haben und bei schlechter Performance oder Fehlern in der Benutzeroberfläche sehr rasch zu Konkurrenzanbietern wechseln. Umso mehr gewinnen Werkzeuge für das Testen der graphischen Benutzeroberfläche von Webapplikationen an Bedeutung.

1.4 Ziele und Zielgruppen dieser Arbeit

Wie bereits in Kapitel 1.3 näher erläutert wurde, sind Webanwendungen einer der am schnellsten wachsenden Bereiche der Softwarebranche. Doch mit der Verbreitung von Webanwendungen wachsen zugleich die Qualitätsansprüche der AnwenderInnen an diese. Da die Online-Konkurrenz sehr groß ist, sind BenutzerInnen gegenüber fehlerhaften und wenig performanten Applikationen sehr ungeduldig und wechseln bei Unzufriedenheit rasch zu einem alternativen Produkt. Umso wichtiger ist es, Webanwendungen gewissenhaft zu betesteten. Hierzu eignen sich funktionale Systemtests besonders gut, da diese die Anwendung aus Sicht der BenutzerInnen überprüfen.

Trotz dessen hohen Stellenwerts werden dem Testen in der Softwareentwicklung oftmals nur unzureichende Ressourcen eingeräumt und funktionale Systemtests sind in dieser Hinsicht keine Ausnahme. Aus diesem Grund besteht das Ziel dieser Arbeit darin, eine Lösung zu entwickeln, die den mit funktionalen Systemtests verbundenen Aufwand senkt und zugleich die Qualität der funktionalen Systemtests wie auch des gesamten Softwareproduktes erhöht.

Zur Erreichung dieses Ziels wird im Rahmen dieser Arbeit ein Framework entwickelt, das als erste Maßnahme die Automatisierung funktionaler Systemtests ermöglicht, um die Kosten für deren Ausführung auf lange Zeit gesehen deutlich zu reduzieren. Das Framework soll künftig als Open Source Lösung zur Verfügung stehen, um vor allem kleineren Firmen und Projekten mit einem niedrigen Budget eine kostengünstige Durchführung automatisierter funktionaler Systemtests zu ermöglichen. Als technische Basis dient das in Kapitel 1.2.3 vorgestellte Open Source Automatisierungswerkzeug Selenium.

Um weitere Aufwandsreduktionen zu realisieren, und zugleich die Qualität der funktionalen Systemtests zu erhöhen, soll das Framework auf die Bedürfnisse automatisierter funktionaler Systemtests maßgeschneidert werden. Dafür wird zunächst eine umfassende Literaturrecherche durchgeführt, welche die spezifischen Eigenschaften und Erfordernisse automatisierter funktionaler Systemtests erheben soll. Auf Basis der im Rahmen dieser Literaturrecherche gewonnen Erkenntnisse werden daraufhin funktionale und qualitative Anforderungen an das Framework formuliert.

Im Anschluss erfolgen der architektonische Entwurf und die Implementierung des Frameworks. Dabei muss ganz besonders darauf geachtet werden, dass das entwickelte Framework Maßnahmen bereitstellt, welche die zuvor definierten Anforderungen umsetzen. Schließlich soll evaluiert werden, in welchem Ausmaß das entwickelte Framework und die durch dieses umgesetzten Anforderungen tatsächlich zu einer Qualitätssteigerung der funktionalen Systemtests, sowie zu einer Reduktion der mit den funktionalen Systemtests verbundenen und der durch qualitativ mangelhafte Software verursachten Kosten, beitragen konnte.

Das entwickelte Framework wird darüber hinaus auch im Rahmen einer konkreten Webanwendung, welche der Erstellung und Verwaltung geschäftlicher Angebote dient, zum Einsatz gebracht. Anhand der im Praxiseinsatz gewonnen Erfahrungen werden den LeserInnen Empfehlungen mitgegeben, wie der Umgang mit dem Framework am sinnvollsten gestaltet werden kann und worauf bei der Durchführung automatisierter funktionaler Systemtests im Allgemeinen zu achten ist.

Da das Framework auf die Entwicklung automatisierter Testfälle abzielt und dafür keine graphische Benutzeroberfläche zur Verfügung stellt, richtet es sich vorwiegend an SoftwareentwicklerInnen sowie SoftwaretesterInnen mit Programmierkenntnissen.

Allerdings bietet diese Arbeit auch zahlreiche Informationen darüber, wie die Qualität funktionaler Systemtests generell gesteigert und der mit diesen verbundene Aufwand verringert werden kann. Somit zählen auch TestmanagerInnen, Qualitätssicherungsverantwortliche und ProjektleiterInnen, sowie alle an den Themen „Funktionaler Systemtests“ und „Testautomatisierung“ interessierten Personen zur Zielgruppe dieser Arbeit.

1.5 Vorarbeiten

Die Durchführung dieser Masterarbeit wäre ohne zahlreiche literarische und technologische Vorarbeiten nicht möglich gewesen. Daher sollen im Rahmen dieses Kapitels die Leistungen all jener gewürdigt werden, die mit ihrem Schaffen den Grundstein für den Erfolg dieser Arbeit gelegt haben.

Der erste Schritt bei der Literaturrecherche für diese Arbeit bestand darin, sich in die Fachbereiche „Softwaretest“ im Allgemeinen und „Funktionaler Systemtest“ im Speziellen einzuarbeiten. Eine umfassende und detaillierte Einführung in das Thema Softwaretest liefern Spillner et al. [1] mit „Software Testing Foundations“, dem Begleitwerk zum ISTQB Foundation Level Certified Tester. Die Autoren decken mit den Grundlagen des Software Testens, dem Software Lebenszyklus, statischen sowie dynamischen Testverfahren, Testmanagement und Testwerkzeugen alle Aspekte ab, die für die ersten Schritte im Bereich des Softwaretests erforderlich sind. Zudem stellt dieses Buch eine hervorragende Ausgangsbasis für tiefergehende Recherchen zu funktionalen Systemtests dar. Nicht zuletzt liefern Spillner et al. die Grundlage für viele der Anforderungen, die an das im Rahmen dieser Arbeit entwickelte Framework ausgearbeitet wurden.

Ebenso eine gute Quelle für alle Fragen rund um das Thema Softwaretests ist „The Art of Software Testing“ von Meyers et al [3]. Die Autoren gehen vor allem auch auf moderne Themen wie das Testen von Webanwendungen und mobilen Applikationen ein und beschäftigen sich darüber hinaus intensiv mit den Bereichen Debugging, Komponententests, Testentwurfsverfahren und statischen Testverfahren.

Gezielt mit dem Test von Webapplikationen setzt sich zudem Franz in seinem „Handbuch zum Testen von Webapplikationen“ [4] auseinander. Neben einer Beschreibung der unterschiedlichen Teststufen dynamischer funktionaler Testarten und detaillierten Erklärungen der für diese benötigten Testfallentwurfsverfahren zeichnet sich Franz vor allem durch die Erläuterung einer großen Anzahl an nicht-funktionalen und speziell auf Webapplikationen zurechtgeschnittenen Testarten aus und geht zudem auch im Detail auf die Aufgaben des Testmanagements ein. Schließlich vereinfacht Franz mit zahlreichen Beispielen den LeserInnen das Verständnis.

Als besonders nützlich für diese Arbeit erwies sich außerdem „Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality“ von Dustin et al. – allein schon der Titel dieses Buches spiegelt im Wesentlichen die Ziele dieser Masterarbeit wider. Die AutorInnen dieses Werkes teilen zahlreiche Erkenntnisse aus ihrer langjährigen Karriere in der Testautomatisierung mit den LeserInnen. Vor allem die Kapitel „Why Automate?“ [5] und „Why Automated Software Testing Fails and Pitfalls to Avoid“ [7] erlauben es, die Gründe für die Automatisierung von Softwaretests und die damit verbundenen Risiken in dieser Arbeit nachzuvollziehen.

Beim Entwurf des Testframeworks wurde ein großer Schwerpunkt auf die Einhaltung bewährter architektonischer Prinzipien gelegt. Die Recherche dieser Prinzipien erfolgte anhand zweier Werke, die im Bereich der Softwareentwicklung zur Standardliteratur zählen: „Software Engineering – A Practitioner’s Approach“ [6] von Pressman und Maxim sowie „Software Engineering“ [2] von Sommerville. Beide streben danach, im Sinne eines holistischen Ansatzes einen möglichst vollständigen Überblick über alle Fachbereiche zu geben, die im Rahmen von Softwareentwicklung relevant sind. So beschäftigen sie sich unter anderem eingehend mit den Bereichen Modellierung, unterschiedlichen Prozessmodellen, Software- und Qualitätsmanagement, agiler Entwicklung, Implementierung, Sicherheit, Anforderungsgestaltung, Softwaretest und Produkt Metriken. Ein großer Schwerpunkt liegt in beiden Fällen jedoch wie bereits angedeutet auf den Fachgebieten Design und Architektur. Schließlich empfiehlt sich das Studium dieser beiden Standardwerke allein schon aufgrund der großen Themenvielfalt, denn es gibt in der Softwareentwicklung kaum Fragestellungen, die diese nicht zumindest teilweise beantworten können.

Einen großen Teil dieser Arbeit nimmt auch die Fragestellung ein, wie Anforderungen bestmöglich gestaltet werden. In dieser Hinsicht erwiesen sich „Requirements-Engineering und Management“ [18] von Rupp sowie „System Requirements Engineering“ [19] von Loucopoulos und Karakostas als herausragende fachliche Quellen. Loucopoulos und Karakostas hatten sich das Ziel gesetzt, ein in ihrer Rolle als Vortragende, Forscher und Praktiker im Jahr 1995 schmerzlich vermisstes Referenzmaterial zur Gestaltung von Anforderungen zu verfassen. Auf kompakte Art und Weise erlaubten die beiden Autoren Einblick in ihre Forschungsergebnisse und befreiten die Anforderungsgestaltung damit aus ihrer bis dahin vergleichsweise stiefmütterlichen Behandlung.

Heutzutage zählen Rupp und seine Gruppe der „Sophisten“ zu den Vorreitern auf dem Gebiet des „Requirements-Engineering“. Für diese Arbeit stellen sie durch die detaillierte Erläuterung von Anforderungsqualität, sprachlichen Effekten, der Formulierung von Zielen, dem Prüfen von Anforderungen sowie von Traceability überaus nützliches Wissen zur Verfügung.

Vor allem zur Beurteilung der Signifikanz automatisierter funktionaler Systemtests in Kapitel 1.3 aber auch, um die durch den Einsatz des entwickelten Testframeworks erzielten Kostensenkungen in Kapitel 2.5.12 besser einschätzen zu können, sind langfristig in der Softwareindustrie erhobene Richtwerte unabkömmlich. „Applied Software Measurement: Global Analysis of Productivity and Quality“ [16] von Capers Jones stellt eine wahre Fundgrube solcher Richtwerte dar. In der aktuellsten dritten Ausgabe dieses Buches präsentiert Jones Daten, welche er über einen Zeitraum von 20 Jahren bis ins Jahr 2008 über verschiedenste Sparten der Softwareindustrie hinweg erhoben hat. Neben der Bereitstellung dieser wertvollen Datenquelle klärt der Autor LeserInnen zusätzlich darüber auf, wie die Leistung von Software in den Phasen der Anforderungsgestaltung, der

Implementierung, des Testens und der Installation gemessen und gesteigert werden kann. Zur besseren Vergleichbarkeit werden die Werte dabei in Function Points umgewandelt. Einen weiteren großen Fokus legt Jones außerdem auf das Design und das Management von Projektkosten sowie Defekten und gibt Hilfestellung bei der Bewertung der Effizienz von Fehlerbeseitigung im Rahmen von Komponententests und mehrstufigen Testsuiten. Bedauerlicherweise merkt man dem Werk jedoch an, dass dieses bereits vor knapp 10 Jahren erschienen ist – gerade die Daten zu Webanwendungen wirken im Jahr 2018 teilweise nicht mehr ganz zeitgemäß.

Erkenntnisse zu aktuelleren Entwicklungen im Bereich des Softwaretestens können hingegen in der 2016 erschienen Studie „Softwaretest in Praxis und Forschung“ [17], welche von den Hochschulen Bremerhaven, Bremen und Köln in Zusammenarbeit mit dem German Testing Board und dem Swiss Testing Board durchgeführt wurde, nachgelesen werden. Sie unterstreichen die zunehmende Automatisierung von Softwaretests und die Tatsache, dass diesen, trotz ihrer hohen Bedeutung für die Qualität von Software, in den meisten Fällen viel zu wenig Ressourcen zugebilligt werden.

Zu guter Letzt ist es dem Autor dieser Arbeit ein großes Anliegen, auf die Leistungen der Open Source Community hinzuweisen, die jene Tools entwickelt hat und wartet, die die Basis des im Rahmen dieser Arbeit geschaffenen Testframeworks bilden. Insbesondere seien hier die EntwicklerInnen von Selenium Webdriver [12], WebdriverIO [20], Jasmine [21] und Allure [22] hervorgehoben. Auf die Rolle, welche diese Werkzeuge im Zusammenspiel mit dem Testframework einnehmen, wird in Kapitel 2.3.2 näher eingegangen.

2 Konzeption, Implementierung und Evaluierung des Frameworks

2.1 Methodischer Überblick

Wie bereits in Kapitel 1.4 angekündigt, wurde im Rahmen dieser Masterarbeit ein Framework für die Automatisierung funktionaler Systemtests entwickelt. Das Framework mit dem Namen „WDIO-WORKFLO“ steht inzwischen als npm-Paket unter der URL <https://www.npmjs.com/package/wdio-workflo> zum Download bereit.

Bevor jedoch mit der Umsetzung des Frameworks begonnen werden konnte, mussten zunächst die spezifischen Eigenschaften eines Frameworks für automatisierte funktionale Systemtests eruiert und im Rahmen von funktionalen und qualitativen Anforderungen Maßnahmen abgeleitet werden, die diese Eigenschaften sicherstellen. Aus diesem Grund war der erste große Teil dieser Masterarbeit durch eine intensive Literaturrecherche geprägt, die als Basis für die Formulierung der besagten Anforderungen diente. Die Erkenntnisse der Literaturrecherche und die daraus abgeleiteten Anforderungen werden in Kapitel 2.2 beschrieben.

Im Anschluss an die Literaturrecherche und auf Basis der Framework-Anforderungen erfolgte die Konzeptionierung von WDIO-WORKFLO, welche im Fokus von Kapitel 2.3 steht. Dieses befasst sich mit den Designentscheidungen, die vor der Entwicklung des Frameworks getroffen wurden, beschreibt die Systemumgebung sowie die externen Abhängigkeiten von WDIO-WORKFLO und widmet sich ausführlich der Framework-Architektur.

Nach dem Entwurf des Frameworks konnte mit dessen Implementierung begonnen werden. Wie auch in der Konzeptionierung war bei der Entwicklung von WDIO-WORKFLO die möglichst vollständige Umsetzung der Framework-Anforderungen das oberste Ziel. Um den LeserInnen dieser Arbeit eine Vorstellung von der Implementierung der einzelnen Komponenten des Frameworks zu ermöglichen und auf die wichtigsten technischen Konzepte einzugehen, ohne den kompletten Quellcode lesen zu müssen, betrachtet Kapitel 2.4 die Aufgaben und die Umsetzung der Framework-Komponenten im Überblick.

Schließlich überprüft Kapitel 2.5, in welchem Ausmaß die an das Framework gestellten Anforderungen erfüllt werden konnten und inwiefern diese zu einer Steigerung der Qualitätskriterien von Software gemäß ISO/IEC 25010:2011 beigetragen haben. Ebenso werden die Auswirkungen auf die Kosten funktionaler Systemtests, die der Einsatz von WDIO-WORKFLO mit sich bringt, genauer untersucht.

2.2 Analyse der erforderlichen Anforderungen

In diesem Kapitel erfolgt eine Auflistung der Anforderungen an das Testframework, welche aus einer breitgefächerten Literaturrecherche gewonnen wurden.

Zur besseren Übersichtlichkeit sind die Anforderungen in folgende Kapitel untergliedert:

- Eigenschaften von JavaScript und React Web-Applikationen
- Grundzüge des Softwaretests
- Merkmale funktionaler Systemtests
- Allgemeine Anforderungen an Testwerkzeuge
- Besonderheiten von Automatisierungslösungen
- Gestaltung von Anforderungen
- Traceability
- Ansprüche an Reports
- Design eines Software Frameworks
- Softwarequalität gemäß ISO/IEC 25010:2011

Jedes Kapitel umreißt zunächst die jeweilige Domäne in Grundzügen, um dem Leser/der Leserin ein grundsätzliches Verständnis über den Kontext zu bieten, aus welchem sich die zugehörigen Anforderungen ableiten. Im Anschluss erfolgt eine Auflistung der jeweiligen Anforderungen in strukturierter Form. Dazu werden diese mit einer ID versehen, um für spätere Verweise eine eindeutige Identifizierung zu ermöglichen. Zu jeder Anforderung werden zudem deren Name und eine Beschreibung, die genauer über den zugrundeliegenden Sachverhalt informiert, aufgeführt. Darüber hinaus ist über eine Quellenangabe vermerkt, an welchen Stellen weiterführende Informationen zur Problemstellung in der Literatur nachgelesen werden können.

Da in einigen Fällen mehrere Unterkapitel als Basis für ein und dieselbe Anforderung herangezogen werden konnten, wurden diese zur Vermeidung von Redundanzen nur jenem Bereich zugeordnet, der am konkretesten zur Umsetzung dieser Anforderung beiträgt.

2.2.1 Eigenschaften von JavaScript und React-Webapplikationen

2.2.1.1 Eigenheiten von Webapplikationen im Allgemeinen

Die Verbreitung des Internets hatte für die Softwarelandschaft einschneidende Veränderungen zur Folge. Mussten Computerprogramme zuvor auf physischen Medien erworben werden, war deren Verbreitung nun wesentlich unkomplizierter möglich. Mit dem riesigen Angebot an online verfügbarer Software stiegen auch die Qualitätsansprüche der BenutzerInnen und deren Ungeduld gegenüber fehlerhafter, schlecht bedienbarer oder nicht performanter Webanwendungen. Ein gründliches Testen der Webapplikationen wurde in einem Umfeld, in welchem der Erfolg von Software und der des Unternehmens oft gleichzusetzen waren, zur Unerlässlichkeit. [3, pp. 193-211]

Webanwendungen sind in ihrem Aufbau im Wesentlichen Client-Server Anwendungen, wobei der Client in einem Browser (Presentation Layer) auf einem Endgerät der BenutzerInnen läuft, während der Serverteil (Business Layer) auf einem physisch getrennten Webserver oder Applikationsserver betrieben wird. Hinzu kommt für gewöhnlich als dritte Schicht eine Datenbank (Data Layer), weshalb man auch von einer Three-Tier Architecture spricht. Diese wird von Meyers et al. [3, p. 195] beschrieben und in Abbildung 5 dargestellt:

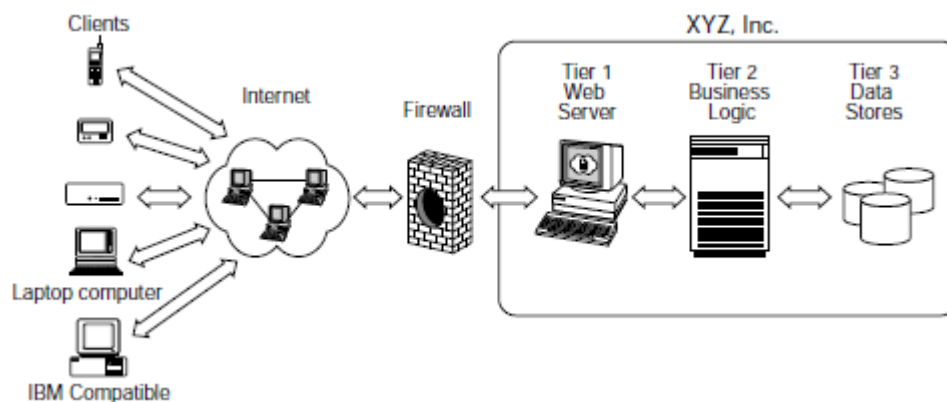


Abbildung 5: Typische Architektur einer E-Commerce Seite (Quelle: [3, p. 195])

Die physikalische Trennung, die Vielschichtigkeit auf Backend-, sowie die Vielfältigkeit an Browsern und Endgeräten auf Client-Seite resultieren in einer Vielzahl möglicher Fehlerquellen. Zusätzlich besitzen Webanwendungen gegenüber herkömmlichen Softwareanwendungen weitere für die Einhaltung hoher Qualität erschwerende Eigenschaften, welche gezielt getestet werden sollten und von Meyers et al. [3, pp. 193-211], sowie Pressman und Maxim [6, pp. 541-542], näher erläutert werden.

Als besonders wichtig erachten diese den Test des Presentation Layers, da dieser als direkte Schnittstelle zu den BenutzerInnen fungiert, gestehen jedoch ebenso ein, dass sich das Testen dieses Layers als sehr arbeitsintensiv darstellt. Eine Herausforderung stellt

hierbei die große Vielzahl an verfügbaren Browsern dar, welche zum Aufrufen einer Webanwendung verwendet werden können. Diesem Aspekt wird durch Browser-Compatibility Testing Rechnung getragen.

Die Überprüfung des Business Layers sollte vor allem durch Blackbox Testing erfolgen, wobei in den Testfällen die Schritte von BenutzerInnen nachgestellt werden.

Zudem müssen bei der Qualitätssicherung auch die Performance, die Bedienbarkeit sowie die Sicherheit von Webapplikationen verstärkt ins Auge gefasst werden und auch die Interoperabilität mit anderen Webapplikationen oder Datenbanken spielt eine wesentliche Rolle beim Testen von Webapplikationen.

All diese Kriterien wurden daher als Anforderungen an das Test-Framework aufgenommen und sind im Teilabschnitt 2.2.1.4 noch einmal detailliert aufgelistet.

2.2.1.2 Moderne Webentwicklung mit dem React-Framework

React [23] ist ein von Facebook entwickeltes Framework zur Client-seitigen Webapplikationsentwicklung. Es basiert auf der Idee, die einzelnen Bestandteile einer Webseite als wiederverwendbare Komponenten im Code abzubilden. Zahlreiche Libraries bieten bereits vorgefertigte Komponenten an, wie etwa die Komponente „Autocomplete“ aus React-Toolbox [24], welche die Auswahl eines oder mehrerer Werte aus einer vorgegebenen Menge erlaubt und dabei anhand bereits eingetippter Zeichen passende Vorschläge vervollständigt und anzeigt. Abbildung 6 veranschaulicht die Darstellung der Komponente „Autocomplete“ in der GUI:



Abbildung 6: "Autocomplete"-Komponente von React-Toolbox [24] in der GUI

Die Verwendung solcher Komponenten-Libraries kann in der Webentwicklung zu großen Einsparungen führen, erschwert aber nicht selten die Testbarkeit, da der innere Aufbau dieser Komponenten oft nicht an die Bedürfnisse eines Tests angepasst werden kann.

Abbildung 7 zeigt den Versuch, eine Instanz der Komponente „Autocomplete“ im DOM identifizierbar zu machen:


```

render () {
  return (
    <Autocomplete
      label="Choose a country"
      multiple={false}
      onChange={this.handleChange}
      source={source}
      value={this.state.countries}
      id="meineId"
      class="meineKlasse"
      myAttr="meinAttribut"
    />
  );
}

```

Abbildung 7: Einbindung der "Autocomplete"-Komponente von React-Toolbox [24]

Wie jedoch in Abbildung 8 ersichtlich wird, können Class-Attribute in diesem Fall nicht gesetzt werden und sind dynamisch generiert. Id-Attribute werden nicht wie erwartet auf dem äußeren <div> Element mit dem Attribut data-react-toolbox="autocomplete" platziert, sondern auf einem innerlich verschachtelten <input> Element und das Attribut myAttr wird überhaupt ignoriert, da React-Toolbox dieses Property nicht kennt und somit nicht an die Bestandteile der Komponente weiterreicht.

```

▼ <div data-react-toolbox="autocomplete" class="theme_autocomplete__lqrTj">
  ▼ <div data-react-toolbox="input" class="theme_input__1lEOC theme_input__2kRvO">
    <input type="text" id="meineId" class="theme_inputElement__caLq1 theme_filled__MxwO">
      ► <span class="theme_bar__30fUU">...</span>
      ► <label class="theme_label__3VQLC">...</label>
      ::after
    </div>
    ► <ul class="theme_suggestions__30N1">...</ul>
  </div>
</div>

```

Abbildung 8: "Autocomplete"-Komponente von React-Toolbox [24] im DOM

Aus Design-technischer Sichtweise (siehe Kapitel 2.2.9) ist wünschenswert, den inneren Aufbau der Komponente auch für Tests nachzubilden und zu kapseln. Dafür müsste jedoch das äußerste, umschließende <div> Element der Komponente angesprochen werden, was im Falle der „Autocomplete“ Komponente nur möglich ist, wenn die strukturelle Abhängigkeit zwischen dem äußeren Container-Element und dem identifizierbaren <input> Element beschrieben werden kann.

2.2.1.3 JavaScript: Mehr als nur eine Browsersprache

JavaScript ist eine plattformübergreifende, Objekt-orientierte Skriptsprache und wird innerhalb einer Host-Umgebung eingesetzt, um sich mit den Objekten der Umgebung zu verbinden und eine programmatische Kontrolle über diese zu überlangen. Aufgrund dieser Fähigkeit hat sich JavaScript zum Rückgrat moderner Webanwendungen im Browser entwickelt, da es Manipulationen des Document Object Models (DOM), welches den Aufbau von Webseiten beschreibt, ermöglicht. [25]

Jedoch ist JavaScript nicht auf Webbrowser beschränkt und wird heutzutage auch Server-seitig eingesetzt, etwa als Basis für die Laufzeitumgebung NodeJS [26], welche durch ihr Event-getriebenes, nicht blockierendes I/O Modell zu einer leichtgewichtigen und effizienten Alternative im Serverbereich aufgestiegen ist.

Anders als etwa Java muss JavaScript nicht kompiliert werden, bietet dafür jedoch auch keine Typsicherheit. Dies hat den Vorteil, dass Objekte in JavaScript zur Laufzeit um zusätzliche Properties erweitert werden können. Jedoch gehen damit auch die Vorzüge von Typsicherheit, wie etwa klar definierte Interfaces und „static checking“, verloren, was zu erhöhter Fehleranfälligkeit im Code beitragen kann. [25]

Eine weitere Eigenheit von JavaScript ist sein Event System. JavaScript ist zwar single-threaded, erlaubt jedoch trotzdem, mehrere Aufgaben anscheinend gleichzeitig und dabei effizient zu verarbeiten. Das Event System basiert auf einer Queue, deren Nachrichten nacheinander abgearbeitet werden. Jede Nachricht wird zur Gänze abgearbeitet, bevor mit der Verarbeitung der nächsten Nachricht begonnen wird. Dadurch werden Bugs aufgrund von Concurrency, wie sie in vielen anderen Sprachen auftreten, vermieden. Zudem wird sichergestellt, dass JavaScript nie blockiert. Node.js etwa nutzt dazu den Umstand, dass moderne Betriebssystem-Kernel multi-threaded operieren – sie können also mehrere Operationen im Hintergrund ausführen. Node.js lädt also, wann immer dies möglich ist, Operationen auf den System Kernel ab und führt in der Zwischenzeit nachgelagerte Nachrichten der Queue aus. Sobald der Kernel diese Operationen erledigt hat, benachrichtigt er Node.js davon, und es wird ein Callback zu Event Queue hinzugefügt. Auf diese Weise kann etwa ein Download vom Server im Hintergrund durch den Kernel übernommen werden, und Mouse oder Keyboard Input Events, die in der Zwischenzeit in der Queue einlangen, werden trotzdem sofort verarbeitet. [27]

Als angenehmer „Nebeneffekt“ wird somit auch die Effizienz der Bearbeitung gesteigert. Dies verdeutlichen Abbildung 9 und Abbildung 10. Während in Abbildung 9 der Program Thread blockiert, bis die Antwort des Servers eingetroffen ist, kann JavaScript am Beispiel von Node.js, wie in Abbildung 10 zu sehen ist, in der Zwischenzeit ein anderes Event (4) verarbeiten, bevor das Callback zur Server Response (3) ausgeführt wird:

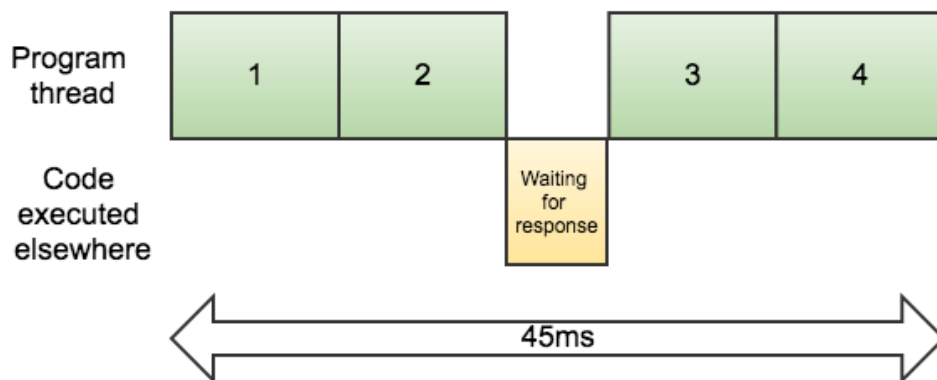


Abbildung 9: Blockierende Queue-Abarbeitung (Quelle: [28])

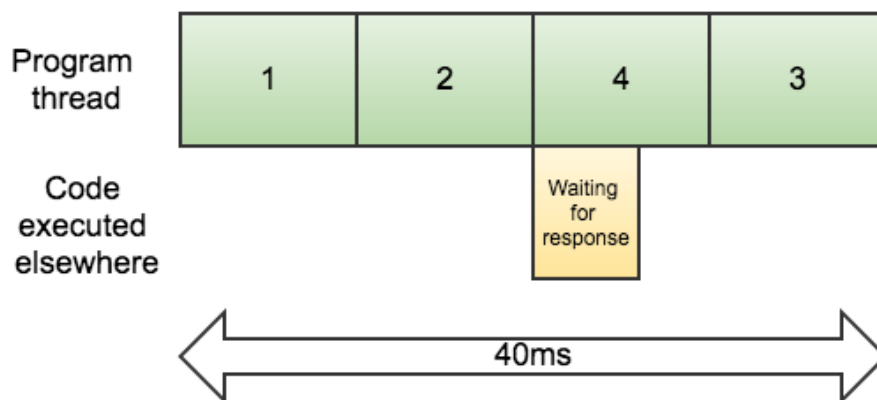


Abbildung 10: Nicht-blockierende Abarbeitung am Beispiel Node.js (Quelle: [28])

2.2.1.4 Anforderungen im Bereich JavaScript und React-Webapplikationen

ID	Name	Quellen
RW1	Unterstützung für Abbildung von Website-Komponenten	[23], [24]
<p>Beschreibung</p> <p>In React und vielen anderen Frontend-Webapplikations-Frameworks werden Webseiten aus wiederverwendbaren Komponenten zusammengebaut. Diese umgehen oftmals klassische HTML-Standards, etwa durch den Einsatz von <div> Tags und Listen anstelle von <select> und <option>. Diese Komponenten sollen im Framework nachgebildet werden können.</p>		

ID	Name	Quellen
RW2	Unterstützung struktureller Selektoren	[24]
<p>Beschreibung</p> <p>Viele Komponenten-Libraries wie React-Toolbox kapseln die innere Struktur ihrer Komponenten in einem äußeren <div> Container (siehe Abbildung 8). Beim „Mapping“ einer solchen Komponente im Test-Framework wird die innere Struktur nachgebildet und gekapselt, so dass nur der Container als Interface nach außen fungiert. Allerdings wird die Testbarkeit oft dadurch erschwert, dass dieser äußere Container nicht über die HTML-Attribute „id“ und „class“ identifiziert werden kann, da diese entweder dynamisch generiert, oder nicht auf dem äußeren Container, sondern auf einem inneren Element, platziert werden. Auch die Möglichkeit einer Identifikation mittels „eigens“ definierter HTML-Attribute besteht zumeist nicht, da die Komponenten-Libraries keine „fremden“ Properties akzeptieren. Daher soll das Framework Selektoren unterstützen, die eine strukturbasierte Identifikation von Website-Elementen ermöglichen, und nicht einzig auf den HTML-Attributen des äußeren Containers beruhen.</p>		

ID	Name	Quellen
RW3	Kompatibilität mit geläufigsten Browsern	[3, p. 198]
<p>Beschreibung</p> <p>Web-Applikationen laufen in einer Vielzahl von Browsern. Das Test-Framework soll die Unterstützung der geläufigsten Browsern ohne großen Zusatzaufwand gewährleisten.</p>		

ID	Name	Quellen
RW4	Funktionalität und Verlinkungen der Web-Applikation müssen über den Presentation Layer effizient testbar sein	[3, pp. 203-205]
<p>Beschreibung</p> <p>Durch das große Angebot im Internet haben die Anwender von Web-Applikationen hohe Ansprüche an die Qualität dieser Produkte entwickelt. Entdecken sie beim Bedienen der Anwendung über die Benutzeroberfläche Fehler, wechseln sie daher recht schnell zu einem der vielen Konkurrenzprodukte. Glenford Meyers et al. weisen darauf hin, dass das Testen des Presentation Layers sehr aufwendig ist. Es ist somit äußerst wichtig, dass das</p>		

Framework die Testbarkeit von Funktionalität und Verlinkungen der Web-Anwendung über die Benutzeroberfläche ermöglicht und den dabei anfallenden Aufwand möglichst klein hält.

ID	Name	Quellen
RW5	Performance der Web-Applikation soll überprüfbar sein	[3, p. 206]
Beschreibung Internet-User sind hohe Geschwindigkeiten in der Bedienung moderner Web-Applikationen gewöhnt und haben kaum Toleranz für eine Web-Anwendung mit schlechter Performance. Daher soll das Framework die Überprüfung der Performance der Web-Applikation ermöglichen.		

ID	Name	Quellen
RW6	Tests sollen Bedienschritte der BenutzerInnen nachahmen	[3, p. 205]
Beschreibung In System-Tests sollen die Schritte, die ein Benutzer in der Verwendung der Applikation tätigt, nachgestellt werden. Glenford Meyers weist darauf hin, dass sich die pragmatische Modellierung dieser Schritte oft anspruchsvoll gestaltet.		

ID	Name	Quellen
RW7	Usability der Web-Anwendung soll geprüft werden können	[6, p. 541]
Beschreibung Die Web-Anwendung sollte für jede Kategorie von vorgesehenen Benutzern erlern- und bedienbar sein. Eine Überprüfung der Usability ist somit erforderlich.		

ID	Name	Quellen
RW8	Security der Web-Applikation soll überprüfbar sein	[6, p. 542]
Beschreibung Aufgrund ihrer exponierten „Lage“ sind Web-Anwendungen oft das Ziel von Sicherheitsverletzungen. Zum Testen der Sicherheit werden potenzielle Angriffsvektoren zunächst eingeschätzt und daraufhin wird versucht, diese auszunutzen.		

ID	Name	Quellen
RW9	Überprüfung der Interoperabilität soll möglich sein	[6, p. 542]
Beschreibung Roger Pressman und Bruce Maxim betonen, dass Web-Anwendungen hinsichtlich ihrer Fähigkeit, mit anderen Anwendungen und Datenbanken zusammenzuarbeiten, überprüft werden müssen.		

2.2.2 Grundzüge des Softwaretests

2.2.2.1 Aktivitäten des Testprozesses

Unabhängig von der Art der entwickelten Software kann der zugehörige Testprozess in gewisse Aktivitäten eingeteilt werden, welche Spillner et. al [1, pp. 19-31] in ihrem Begleitwerk zur ISTQB Certified Tester Foundation Level Zertifizierung, „Software Testing Foundations“, detailliert beschrieben und in Abbildung 11 veranschaulichen. Dieser Abschnitt soll einen Überblick über die wesentlichsten Ziele und Tätigkeiten innerhalb der genannten Aktivitäten bieten.

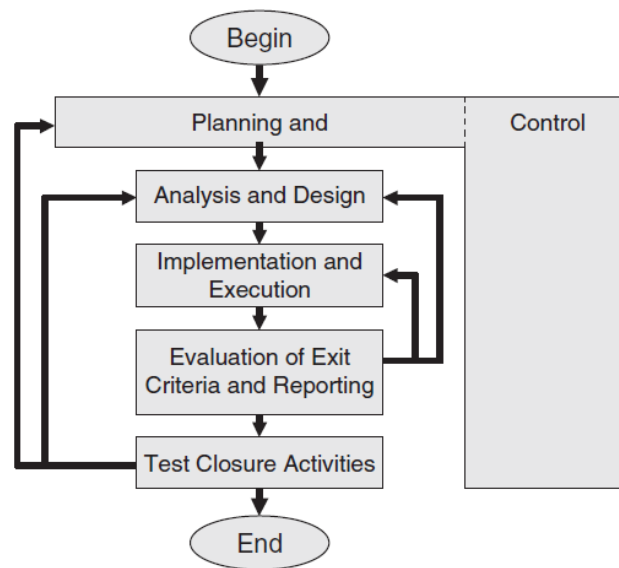


Abbildung 11: ISTQB fundamentaler Testprozess (Quelle: [1, p. 19])

Zu Beginn jedes Testprozesses steht die Testplanung, welche sich vor allem mit der Definition der Testziele und dem Entwurf einer Teststrategie befasst. Letztere beschäftigt sich unter anderem damit, wie intensiv bestimmte Teile eines Systems getestet werden müssen, was eine Priorisierung der Tests erforderlich macht.

Die Einhaltung der Teststrategie sowie der gesetzten Ziele wird im Rahmen der Teststeuerung über den Verlauf aller anderen Aktivitäten hinweg kontrolliert. Bei Bedarf, etwa aufgrund neuer Anforderungen oder geänderter Rahmenbedingungen, werden diese auch angepasst.

Ist die Testplanung abgeschlossen, folgt ein Zyklus sich wiederholender Aktivitäten, beginnend mit der Analyse der Testbasis und dem Entwurf der Testfälle. Zunächst werden die Anforderungen auf ihre Testbarkeit hin überprüft. Sind diese nicht präzise genug formuliert, können zu deren Validierung keine Testfälle formuliert werden und die Anforderungen müssen nochmals überarbeitet werden. Ein wichtiger Schritt in dieser Phase ist auch, eine Rückverfolgbarkeit zwischen den Anforderungen und den daraus entworfenen Testfällen sicherzustellen. Die Testfälle selbst werden zumeist in logische und konkrete

Testfälle unterteilt. Dies birgt den Vorteil, dass mit dem Design von logischen Testfällen auf Basis von Äquivalenzklassen bereits in einer frühen Phase der Softwareentwicklung begonnen werden kann und diese auch den Softwareentwicklern als Vorlage dienen können, während die Testfälle mit konkreten Testdaten während oder erst im Anschluss an die Implementierung erstellt werden. Wichtig ist es in jedem Fall, die Vorbedingungen von Testfällen zu beschreiben und deren erwartetes Verhalten, das sogenannte Testorakel, bereits vor der Testausführung festzulegen.

Im Anschluss folgt die Realisierung und Durchführung, welche sowohl die Implementierung von Testfällen (im Sinne von Programmierung automatisierter Testfälle) sowie deren automatisierte oder manuelle Ausführung umfasst. Zunächst müssen diese jedoch vorbereitet und die Testumgebung und notwendige Testwerkzeuge beschafft und eingerichtet werden. Während der Testausführung ist darauf zu achten, dass die einzelnen Schritte und deren Resultate vollständig und exakt geloggt werden. Dies ermöglicht eine spätere Reproduzierbarkeit der Tests und erleichtert außerdem die Analyse von Abweichungen zwischen erwartetem und tatsächlich eingetretenem Verhalten.

Es empfiehlt sich zudem, Änderungen an der Software durch Regressionstests zu überprüfen, um sicherzustellen, dass durch die neuen Features keine bestehenden Funktionalitäten beschädigt wurden. Bei umfangreichen Projekten können die Durchlaufzeiten eines vollständigen Regressionstests sehr hoch ausfallen, weshalb bei der Ausführung eine Selektion nach bestimmten Kriterien, wie etwa der Priorität der Testfälle oder der verknüpften Anforderungen, erforderlich werden kann.

Schließlich sollten während der Testausführung bestimmte Messwerte gesammelt werden, die die Teststeuerung erleichtern oder überhaupt ermöglichen. Dazu zählen etwa die Testabdeckung oder die Zeit, die für die Ausführung der einzelnen Testfälle benötigt wurde.

Nach Ende der Testausführung folgt die Bewertung und der Bericht. Anhand der Testergebnisse und der gesammelten Messwerte wird beurteilt, ob die Testendekriterien erreicht wurden und somit das Testen beendet werden kann. Sollte sich bei der Analyse der fehlerhaften Testergebnisse herausgestellt haben, dass die Fehlerursachen in der Software und nicht in im Aufbau des Testfalls selbst liegen, müssen die Fehler in einem Bug Tracker eingemeldet, sowie deren Schweregrad und die Priorität der Behebung festgelegt werden.

Zu guter Letzt erfolgt mit dem Abschluss eine oft übersehene Aktivität im Testprozess, der dennoch große Bedeutung zukommt. Sie umfasst etwa die Dokumentation von Abnahmen, die Archivierung und Konservierung von Testmitteln, Testumgebung und Infrastruktur, sowie die Analyse des Testprozesses und ein Festhalten der „Lessons Learned“ für dessen künftige Verbesserung.

Das entwickelte Test-Framework wird mit vielen, wenn nicht allen Phasen des Testprozesses in Berührung kommen und diesen nach allen Kräften unterstützen.

Dementsprechend sind auch die aus dem Testprozess resultierenden Anforderungen über verschiedene Teilbereiche des Frameworks verteilt und finden sich vor allem in den Kapiteln zur Gestaltung von Anforderungen (2.2.6.4), zu funktionalen Systemtests (2.2.3.2) und zu den Ansprüchen an Reports (2.2.8.2) wieder. Anforderungen, die keinem spezifischen Bereich zugeordnet werden konnten, sind am Ende dieses Kapitels aufgeführt.

2.2.2.2 Allgemeine Testgrundsätze

Ergänzend zu den Aktivitäten des Testprozesses sollen in diesem Unterkapitel noch ein paar allgemeine Grundsätze des Softwaretestens genannt werden:

- Spillner et al. [1, pp. 34, 76-77, 222] empfehlen, mit dem Testen schon früh im Projekt zu beginnen. Dazu muss unter anderem sichergestellt werden, dass eingesetzte Testframeworks bereits aufgesetzt sein sollten. Hierfür empfiehlt sich eine eigene Pilotprojektphase. Zudem soll auch das Design von Tests für jeden Testlevel so früh wie möglich, etwa während oder sogar vor den entsprechenden Entwicklungsaktivitäten, erfolgen und Tester sollten in den Review Prozess der Entwicklungsdokumente möglichst früh miteinbezogen werden.
- Weiters weisen Spillner et al. [1, p. 34] auf das Phänomen der Fehlerhäufung hin, welches besagt, dass neue Fehler vermehrt an Stellen gefunden werden, an denen bereits zuvor viele Fehler entdeckt wurden. Diesem Umstand kann entgegengekommen werden, indem die Möglichkeit geschaffen wird, gezielt einzelne Bereiche der Software zu betesten.
- Ein offensichtlicher Teil der Testfalldefinition, welcher dennoch als einer der am häufigsten gemachten Fehler im Softwaretest angesehen wird, ist laut Meyers und Maxim [3, p. 13] die exakte, vorzeitige Definition von erwartetem Verhalten. Wird darauf vergessen oder wird diese nicht exakt genug durchgeführt, kann es passieren, dass fehlerhafte Ergebnisse als korrekte Ergebnisse interpretiert werden. Deshalb sollte jeder Testfall zumindest eine genaue Beschreibung der Eingabedaten, sowie eine präzise Definition des erwarteten Verhaltens des Programms für diese Eingabedaten enthalten.

2.2.2.3 Anforderungen im Bereich Grundzüge des Softwaretests

ID	Name	Quellen
ST1	Trennung zwischen logischen und konkreten Testfällen	[1, pp. 23, 34]
Beschreibung Es wird eine Trennung zwischen logischen Testfällen, welche auf Äquivalenzklassen basieren, und konkreten Testfällen, welche mit „echten“ Eingabewerten gespeist werden, angestrebt.		

ID	Name	Quellen
ST2	Beschreibung der Vorbedingungen	[1, p. 23]
Beschreibung Für jeden Testfall muss die Ausgangssituation (die Vorbedingungen) beschrieben werden. Es soll klar sein, in welchem initialen Zustand sich die Umgebung für jeden Test befinden muss.		

ID	Name	Quellen
ST3	Definition des Testorakels vor Testdurchführung erzwingen	[1, p. 23], [3, p. 13]
Beschreibung Das Testorakel dient zur Bestimmung des erwarteten Verhaltens von Testfällen. Als Basis für das Testorakel dienen vorwiegend die Spezifikationen des Testobjekts. Die Festlegung des erwarteten Verhaltens muss vor der Testdurchführung erfolgen, um zu verhindern, dass fehlerhafte Ergebnisse fälschlicherweise als korrekt eingestuft werden.		

ID	Name	Quellen
ST4	Minimierung des Aufwands für die Einrichtung der Testinfrastruktur	[1, p. 25]
Beschreibung Parallel zum Entwurf der Testfälle soll bereits die Einrichtung und Verifizierung der Testinfrastruktur erfolgen, um Verzögerungen in der Testausführung zu vermeiden. Um dieses Ziel zu erreichen, soll der benötigte Aufwand für das Einrichten der Testinfrastruktur möglichst gering gehalten werden.		

2.2.3 Merkmale funktionaler Systemtests

2.2.3.1 Rekapitulation der Grundlagen funktionaler Systemtests

Die spezifischen Anforderungen im Bereich der funktionalen Systemtests basieren allesamt auf Konzepten, die bereits in Kapitel 1.1 „Grundlagen funktionaler Systemtests“ ausführlich behandelt wurden. Dieses kann bei Bedarf auch als Quelle für tiefergehende Literaturverweise herangezogen werden. Um jedoch die anschließenden Anforderungen in ihrem Kontext näher begreifen zu können, werden die wichtigsten Grundlagen funktionaler Systemtests an dieser Stelle noch einmal zusammengefasst.

Funktionale Systemtests sind im V-Modell an der zweithöchsten Stufe angesiedelt und gelten somit als „Higher Order“ Tests, welche die Software nicht aus der Perspektive technischer Spezifikationen, sondern aus jener der späteren AnwenderInnen evaluieren. Aus diesem Grund bauen die Testfälle funktionaler Systemtests auf Black-Box-Testfallentwurfsverfahren auf. Diese setzen keine Kenntnisse über die inneren Abläufe und den inneren Aufbau des Systems voraus, sondern interagieren, wie auch die späteren BenutzerInnen, nur über die äußeren Schnittstellen (etwa der GUI) mit dem System. Das Design von Testfällen für funktionale Systemtests ist meist wesentlich komplexer und aufwendiger, als dies bei den White-Box-Testfallentwurfsverfahren der Komponententests der Fall ist. Zu den Black-Box-Testfallentwurfsverfahren zählen etwa der Testfallentwurf auf Basis von Anwendungsfällen, der zustandsbasierte sowie der anforderungsbasierte Testfallentwurf, in Kombination mit Grenzwertanalyse und Äquivalenzklassenbildung.

Das Ziel funktionaler Systemtests ist es einerseits, Fehler im Zusammenspiel aller vollständig integrierten Komponenten und Teilsysteme aufzudecken, und andererseits, die externe (funktionale) Spezifikation, sowie nicht-funktionale Systemmerkmale, welche beide aus den BenutzerInnen-Anforderungen abgeleitet wurden, zu validieren. Darüber hinaus soll ein Verifizierungsschritt vor allem jene Fehler aufdecken, die bei der Übersetzung der Ziele in die externe Spezifikation aufgetreten sind und sicherstellen, dass die externe Spezifikation die BenutzerInnen-Anforderungen vollständig, korrekt und konsistent beschreibt.

Da sich die Abnahmekriterien eines Softwareprojekts sehr gut durch externe Spezifikationen und nicht-funktionale Systemmerkmale beschreiben lassen, ist Release Testing ein häufiges Einsatzszenario funktionaler Systemtests. Während das Release Testing, wie der Name schon vermuten lässt, in Vorbereitung eines Releases durchgeführt wird, und daher auf die vollständige Überprüfung der Einhaltung der BenutzerInnen-Anforderungen abzielt, eignen sich funktionale Systemtests auch für die Durchführung von Smoke Tests. Diese werden im Anschluss an Buildvorgänge, und somit in wesentlich kürzeren Intervallen, ausgeführt und sollen sicherstellen, dass das Resultat des Builds stabil genug ist, um gründlicheren Tests unterzogen werden zu können. Smoke Tests müssen nicht vollständig sein, sollen aber wesentliche Probleme in der Software aufdecken können.

2.2.3.2 Anforderungen im Bereich Funktionale Systemtests

ID	Name	Quellen
FS1	Verifizierung der Anforderungen im Rahmen der externen Spezifikation	[1, p. 41] [3, pp. 114-118]
Beschreibung Das Framework soll eine Verifizierung der externen Spezifikation ermöglichen, um etwaige Fehler bei der Übersetzung der Ziele in ebendiese frühzeitig zu beheben. Unter anderem sollen dadurch die Korrektheit, Vollständigkeit und Konsistenz der externen Spezifikation gewährleistet werden.		

ID	Name	Quellen
FS2	Validierung der externen Spezifikation	[1, p. 41] [3, pp. 114-118]
Beschreibung Die Validierung der externen Spezifikation, welche aus den Anforderungen der BenutzerInnen abgeleitet wurde, muss mithilfe des Frameworks möglich sein. Somit soll eine fehlerhafte, unvollständige oder gar fehlende Implementierung der durch die externe Spezifikation beschriebenen Anforderungen aufgedeckt werden.		

ID	Name	Quellen
FS3	Korrektheit, Kompatibilität und Interaktion zwischen Interfaces der einzelnen Komponenten prüfen	[2, p. 219] [4, pp. 102-103]
Beschreibung Im Rahmen von Systemtests werden Fehler aufgedeckt, die im Zusammenspiel der vollständig integrierten Komponenten und Teilsysteme eines Systems auftreten. Daher muss das Framework überprüfen können, ob die vollständig integrierten Komponenten kompatibel zueinander sind, korrekt miteinander interagieren und die richtigen Daten zur richtigen Zeit über ihre Schnittstellen austauschen.		

ID	Name	Quellen
FS4	Reduzierung der Komplexität funktionaler Systemtests	[2, p. 222]
Beschreibung Testfälle funktionaler Systemtests sind üblicherweise wesentlich komplexer und aufwendiger umzusetzen als jene von Komponenten- oder Integrationstests. Die Reduzierung dieser Komplexität stellt daher eine Anforderung an das Framework dar.		

ID	Name	Quellen
FS5	Unterstützung von Release Testing	[2, p. 224]
Beschreibung Das Release Testing konzentriert sich nicht auf das Aufdecken neuer Fehler, sondern validiert, ob ein Softwareprodukt seine Anforderungen gemäß den Abnahmekriterien vollständig erfüllt und somit für den externen Gebrauch einsatzbereit ist.		

ID	Name	Quellen
FS6	Berücksichtigung von Black-Box-Testfallentwurfsverfahren	[2, pp. 219-225] [4, pp. 102-103]
Beschreibung Funktionale Systemtests betrachten ein System aus der Sichtweise der BenutzerInnen, welche üblicherweise keine Kenntnisse über den inneren Aufbau und die inneren Abläufe des Systems aufweisen können. Auch der Testfallentwurf für funktionale Systemtests erfolgt somit ausschließlich unter Zuhilfenahme der äußeren Schnittstellen des Systems, wodurch sich Black-Box-Testfallentwurfsverfahren wie der Testfallentwurf auf Basis von Anwendungsfällen, der zustandsbasierte sowie der anforderungsbasierte Testfallentwurf, in Kombination mit Grenzwertanalyse und Äquivalenzklassenbildung, anbieten.		

ID	Name	Quellen
FS7	Unterstützung von Smoke Testing	[6, p. 479]
Beschreibung Smoke Tests prüfen im Gegensatz zu Release Tests keine vollständige und korrekte Einhaltung der Anforderungen, sondern zielen darauf ab, im Anschluss an einen Buildvorgang wesentliche Fehler im Build aufzudecken, um beurteilen zu können, ob dieser für umfassendere Tests stabil genug ist.		

2.2.4 Allgemeine Anforderungen an Testwerkzeuge

Die Landschaft an Werkzeugen, die TesterInnen bei der Bewältigung der Testaktivitäten hilfreich zur Seite stehen, ist umfangreich und vielfältig. Im Verlauf dieses Kapitels wird eine Übersicht über die verfügbaren Arten von Testwerkzeugen und deren Verwendungszweck geboten. Diese orientiert sich stark am Lehrplan des ISTQB Certified Tester Foundation Level Zertifikats, dessen Inhalte Spillner et al. in „Software Testing Foundations“ [1, pp. 205-217] beschreiben.

2.2.4.1 Generelle Ziele und Einteilung von Testwerkzeugen

Andreas Spillner, Tilo Linz und Hans Schäfer [1, p. 205] zählen drei wesentliche Ziele beim Einsatz von Testwerkzeugen auf:

- **Erhöhung der Testeffizienz:** Repetitive und zeitaufwändige manuelle Aufgaben können etwa mit Hilfe von statischer Analyse und dynamischen Testwerkzeugen automatisiert werden.
- **Ermöglichung von Tests:** Performancetests, Lasttests und viele andere Testarten sind ohne Zuhilfenahme von Testwerkzeugen nicht ökonomisch sinnvoll realisierbar.
- **Verbesserung der Zuverlässigkeit von Tests:** Die Zuverlässigkeit kann durch die Automatisierung manueller Aufgaben, wie dem Vergleich großer Datenmenge oder der Simulation von Programmverhalten, gesteigert werden.

Weiters treffen die Autoren eine grobe Einteilung der unterschiedlichen Arten von Testwerkzeugen in Werkzeuge für Testmanagement, Testspezifikation, statisches Testen, dynamisches Testen und nicht-funktionales Testen. Die folgenden Unterkapitel beleuchten die wichtigsten Merkmale der jeweiligen Kategorien.

2.2.4.2 Werkzeuge für Testmanagement

Bei der Recherche von Werkzeugen für Testmanagement in „Software Testing Foundations“ [1, pp. 206-209] stellt man fest, dass diese wohl die vielfältigste Kategorie an Testwerkzeugen darstellen, da ihr Einsatzgebiet, das Testmanagement, die zentrale Schnittstelle des Testprozesses zu anderen Aktivitäten der Softwareentwicklung darstellt. Somit finden sich auch einige der Anforderungen, die an Testmanagementwerkzeuge gestellt werden können, in anderen Kapiteln der Literaturrecherche wieder, wie etwa in der Gestaltung von Anforderungen (2.2.6.4), in den Ansprüchen an Reports (2.2.8.2) oder im Bereich der Traceability (2.2.7.4).

Testmanagement Tools werden demnach im Allgemeinen für eine vereinfachte Dokumentation, Priorisierung, Auflistung und Wartung von Testfällen eingesetzt.

Fortgeschrittene Varianten von Testmanagementwerkzeugen unterstützen darüber hinaus anforderungsbasiertes Testen, indem sie die (oftmals nahtlose) Integration von Anforderungen und deren Verknüpfung zu den Testfällen ermöglichen. Sie erlauben TesterInnen, die Anforderungen zu priorisieren und deren Implementierungsstatus zu verfolgen. Darüber hinaus helfen sie dabei, Inkonsistenzen in den Anforderungen aufzudecken und nicht getestete Anforderungen zu identifizieren. Schließlich ermöglichen sie den Anstoß der Testausführung unter Auswahl der gewünschten Testfälle und der dadurch überprüften Anforderungen.

Werden Testmanagementwerkzeuge mit einem Incident Management Tool verbunden, ermöglichen sie die Verknüpfung von Fehlern mit den Testfällen, die diese aufgedeckt haben. Dadurch ergibt sich die Möglichkeit, Testfälle erneut auszuführen, wenn die verknüpften Fehler behoben wurden, um sicherzustellen, dass diese auch tatsächlich beseitigt wurden.

Ein weiterer wichtiger Aspekt ist die Unterstützung von Konfigurationsmanagement, um Auskunft darüber geben zu können, in welchen Versionen sich die durch einen Testlauf überprüften Komponenten befunden haben. Dies ermöglicht Aussagen darüber, welche Versionen eines Testobjekts von welchen Fehlern betroffen sind. Der Fortschritt des Fehlerbehebungsprozesses kann so über mehrere Konfigurationen hinweg beurteilt werden.

Sowohl Testmanagementwerkzeuge als auch Incident Management Tools beinhalten teils ausgiebige Analyse- und Reportfeatures - etwa die Möglichkeit, die komplette Testdokumentation (einschließlich Testplan, Testspezifikation und Testabschlussbericht) auf Basis der vom Tool gesammelten Daten zu erstellen. Diese Daten können in vielseitiger Art und Weise für die Beurteilung des Testfortschritts und die Steuerung des Testprozesses eingesetzt werden, etwa durch Zählen der ausgeführten Testfälle und deren Verteilung in erfolgreiche und fehlerhafte Testfälle.

Alles in allem ermöglicht eine derartige Toolchain, den Teststatus von den Anforderungen, über die Testfälle hinweg bis zu deren Ergebnissen, verknüpften Fehlerberichten und den durch diese ausgelösten Codeänderungen, nachzuverfolgen.

2.2.4.3 Werkzeuge für Testspezifikation

Die Autoren von „Software Testing Foundations“ [1, pp. 209-210] beschreiben die Verwendung von Testdatengeneratoren als Werkzeuge für die Testspezifikation. Diese stehen zwar nicht im Fokus des entwickelten Frameworks, aber der Vollständigkeit halber sollen sie kurz erwähnt werden. Testdatengeneratoren unterstützen die DesignerInnen von Testfällen, indem sie die Erstellung großer Mengen an Testdaten unter geringem Aufwand

ermöglichen. Sie können in datenbasierte, codebasierte, Interface-basierte und Spezifikations-basierte Testdatengeneratoren unterschieden werden.

2.2.4.4 Werkzeuge für statisches Testen

Spillner et al. [1, pp. 210-211] zufolge sind Werkzeuge für statisches Testen wichtige Hilfsmittel, um Fehler in den frühen Phasen des Entwicklungszykluses aufzudecken. Sie ermöglichen eine statische Analyse des Codes oder der Spezifikationen, noch bevor diese in ausführbare Programme übersetzt werden. Dadurch finden weniger Fehler ihren Weg in die Phase des dynamischen Testens, was zu einer Senkung von Kosten und Entwicklungszeit führt.

Vorwiegend organisatorisch angesiedelte Vertreter dieser Kategorie sind Werkzeuge zur Unterstützung der Planung, Ausführung und Evaluierung von Reviews. Sie verwalten Informationen zu geplanten und durchgeführten Review Meetings, deren Teilnehmern und während der Meetings aufgedeckten Problemen und deren Lösungen.

Statische Analyseprogramme messen diverse Eigenschaften des Programmcodes, etwa die zyklomatische Zahl und andere Code-Metriken. Dadurch können komplexe Bereiche im Quellcode identifiziert werden, welche besonders fehleranfällig sind und daher äußerst gründlich reviewed werden sollten. Außerdem eignen sich statische Analyseprogramme, um Inkonsistenzen, Fehler und Anomalien im Programmcode aufzuspüren.

Schließlich existieren auch Werkzeuge, die nicht den Programmcode, sondern formale Modelle des Softwaredesigns überprüfen. Sie werden Model Checker genannt und sind in der Lage, fehlende Zustände, Zustandsübergänge oder anderweitige Unstimmigkeiten in den formalen Modellen offenzulegen.

2.2.4.5 Werkzeuge für dynamisches Testen

Sprechen wir im Allgemeinen von Testwerkzeugen, so beziehen wir uns laut Spillner et al. [1, pp. 211-216] oft auf Werkzeuge zur automatisierten Ausführung dynamischer Tests. Daher überschneiden sich die Framework Anforderungen, die aus diesem Unterkapitel resultieren, stark mit jenen Anforderungen, die aus dem Kapitel 2.2.5 „Besonderheiten von Automatisierungslösungen“ hervorgehen. Jedoch sind Automatisierungslösungen und Werkzeuge für dynamisches Testen nicht zwingendermaßen gleichzusetzen, denn Werkzeuge für dynamisches Testen können auch bei einer manuellen Testausführung unterstützen. Somit werden Anforderungen, die auf Eigenschaften basieren, die insbesondere oder ausschließlich auf Automatisierungslösungen zutreffen, im Unterkapitel 2.2.5.4 „Anforderungen im Bereich Testautomatisierungslösungen“ aufgelistet, während jene Anforderungen, die auch auf Testwerkzeuge zutreffen, welche die manuelle Testausführung unterstützen, am Ende dieses Kapitels behandelt werden.

Werkzeuge für dynamisches Testen reduzieren die mechanische Arbeit bei der Testausführung, indem sie Eingabedaten an das Testobjekt senden, dessen Reaktionen aufzeichnen und die Testausführung dokumentieren. Sie sind eng mit dem Testinterface des Testobjekts verknüpft und somit genau auf dieses zugeschnitten, weshalb es eine Vielzahl unterschiedlicher Varianten solcher Werkzeuge gibt.

Testobjekte, die über kein Graphisches User Interface verfügen, müssen beim dynamischen Testen über deren Programmschnittstelle angesprochen werden. Werkzeuge, die dies ermöglichen, werden Testtreiber genannt und vor allem in Komponenten- und Integrationstests eingesetzt. Ihnen fällt eine hohe Bedeutung zu, da die besagten Testobjekte manuell gar nicht dynamisch getestet werden könnten.

Debugger erleichtern die Analyse der Zustände und Abläufe in einem Programm während dessen Ausführung. Sie ermöglichen es, im Code von Zeile zu Zeile zu springen, und dabei Variablen auszulesen und zu setzen. Sie sind für die Reproduktion und Analyse von Fehlerzuständen äußerst wichtig, denn sie erlauben, bestimmte Testsituationen zu erzwingen, deren Nachstellung ohne den Einsatz von Debuggern einen großen Aufwand erfordern würde.

Falls das Graphische User Interface gleichzeitig auch als Testinterface fungiert, können TesterInnen auf sogenannte Testroboter zurückgreifen. Diese zeichnen die Eingaben manueller Tester auf und erlauben, diese erneut abzuspielen. Dieses Prinzip mag verlockend klingen, birgt jedoch einige Fallstricke. So sind die durch Testroboter automatisch generierten Testfälle oftmals nicht sehr robust gegenüber Änderungen in der GUI und müssen häufig manuell nachbearbeitet werden – vor allem, wenn diese für Regressionstests genutzt werden sollen.

Unabhängig davon, ob Testskripte durch Testroboter generiert oder manuell erstellt wurden, sollten sie sich einer bekannten Programmier- oder Skriptsprache bedienen. Dadurch können die TestfallerstellerInnen auf wohlbekannte, allgemeine Eigenschaften von Programmiersprachen wie Entscheidungen, Schleifen, Funktionsaufrufe etc. zurückgreifen, was die Erlernbarkeit der „Testskriptsprache“ wesentlich vereinfacht.

Um die Wiederverwendbarkeit in Testskripten zu steigern, verfolgen diese oftmals bestimmte Architekturansätze. Vertreter hiervon sind Daten-getriebenes und Keyword-getriebenes Testen. Dabei werden oftmals verwendete Abfolgen von Testschritten in Funktionen zusammengefasst, welche über sogenannte Keywords angesprochen werden. Diese Aufrufe sind parametrisierbar, was die mehrmalige Ausführung einer Sequenz von Testschritten mit unterschiedlichen Daten erlaubt.

Ein weiterer essentieller Bestandteil von Werkzeugen für dynamisches Testen sind Komparatoren. Diese vergleichen die erwarteten mit den tatsächlichen Resultaten und weisen Abweichungen zwischen diesen als Fehler aus.

Darüber hinaus beinhaltet die Familie der Werkzeuge für dynamisches Testen noch Hilfsmittel zur Abdeckungsmessung. Im Falle von White-Box-Testing ermitteln diese die Abdeckung der von Testfällen „behandelten“ Codepassagen – etwa in Form von Pfad- und Zweigabdeckung. In Bezug auf funktionale Systemtests kann jedoch genauso das Verhältnis zwischen getesteten und nicht getesteten Requirements oder die Rate automatisiert getesteter Requirements im Vergleich zu manuell getesteten Requirements gemessen werden.

2.2.4.6 Werkzeuge für nicht-funktionales Testen

Schließlich gibt es Werkzeuge, die Unterstützung beim nicht-funktionalen Testen bieten. Diese werden von Spillner et al. [1, pp. 216-217] ebenfalls näher erläutert. Werkzeuge für nicht-funktionales Testen spielen in dem betrachteten Testframework jedoch nur eine untergeordnete Rolle.

Wichtige Vertreter dieser Kategorie sind unter anderem Monitore, welche beim Performance Testing eingesetzt werden. Sie messen die Antwortzeiten und das Übertragungsverhalten des Systems in Abhängigkeit der angewandten Last und erstellen aus diesen Daten Charts, Diagramme und Berichte.

2.2.4.7 Anforderungen im Bereich Testwerkzeuge

ID	Name	Quellen
TW1	Steigerung der Testeffizienz und -abdeckung	[1, p. 205] [7]
Beschreibung Das Framework soll zu einer Steigerung der Testeffizienz durch Automatisierung von sich wiederholenden und zeitaufwendigen Aufgaben beitragen. Darunter fallen beispielsweise die Analyse von Programmcode durch statische Analysemethoden, die Automatisierung dynamischer Testfälle und Werkzeuge zur Durchführung von Lasttests. Ebenso soll das Framework die für die Automatisierung selbst anfallende Zeit und Komplexität reduzieren.		

ID	Name	Quellen
TW2	Steigerung der Zuverlässigkeit von Tests	[1, p. 205] [5]
Beschreibung Durch Automatisierung monotoner oder mühsamer Tätigkeiten, etwa dem Vergleich großer Datenmengen, sowie durch von Werkzeugen objektiv durchgeführte Messungen, soll die Zuverlässigkeit von Tests gesteigert werden. Unter diese monotonen Tätigkeiten fallen etwa auch Regressionstests, da diese üblicherweise langwierig und ermüdend sind. Hier spielt die sogenannte Tester Fatigue eine große Rolle: TesterInnen gewöhnen sich daran, wie ein System funktioniert, und nehmen Probleme abseits der gewohnten Lösung nicht mehr wahr.		

ID	Name	Quellen
TW3	Ermöglichung von Tests, die manuell nicht durchführbar sind	[1, pp. 205, 208]
Beschreibung Eine Anforderung an und häufiges Ziel von Testwerkzeugen ist es, Tests zu ermöglichen, die manuell nicht ökonomisch sinnvoll durchführbar sind. Dazu zählen etwa Performance Tests, die die Antwortzeiten und das Kommunikationsverhalten einer Software in Abhängigkeit der eingesetzten Last beurteilen, aber auch Testtreiber, die die Testbarkeit von Testobjekten, welche nicht über Benutzerschnittstellen verfügen, erlauben.		

ID	Name	Quellen
TW4	Verknüpfungsmöglichkeit zu Incident Management Tool	[1, p. 208]
Beschreibung Ein Testmanagement Werkzeug soll eine Verknüpfung zwischen Testfällen, und den Fehlern, die diese aufgedeckt haben, ermöglichen.		

ID	Name	Quellen
TW5	Möglichkeit, fehlgeschlagene oder „korrigierte“ Testfälle erneut auszuführen	[1, p. 208]
Beschreibung Das Framework soll die Möglichkeit bieten, fehlgeschlagene oder „korrigierte“ Testfälle erneut auszuführen, um sicherzustellen, dass Fehler tatsächlich behoben und durch die Änderungen keine neuen Fehler in bestehende Funktionalitäten eingeführt wurden.		

ID	Name	Quellen
TW6	Unterstützung für Konfigurationsmanagement	[1, p. 208]
Beschreibung Konfigurationsmanagement ermöglicht es, die Resultate eines Testlaufs bestimmten Versionen von Testobjekten zuzuordnen und die Beseitigung von Fehlern und den generellen Zustand der Applikation über mehrere Konfigurationen hinweg zu verfolgen.		

ID	Name	Quellen
TW7	Statische Überprüfung des Source-Codes	[1, pp. 210-211]
Beschreibung Durch Statische Überprüfung des Source-Codes sollen Fehler und Inkonsistenzen im Source Code möglichst frühzeitig, nämlich bereits vor der Ausführung des Programms, aufgedeckt und somit die Kosten und Entwicklungszeit gesenkt werden.		

ID	Name	Quellen
TW8	Unterstützung einer Daten- und Keyword-getriebenen Testfallarchitektur	[1, pp. 214-215]
Beschreibung Durch den Einsatz einer Daten- und Keyword-getriebenen Testfallarchitektur können oft benötigte Sequenzen von Testschritten in Funktionen zusammengefasst, mit unterschiedlichen Daten parametrisiert und dadurch leichter wiederverwendet werden.		

ID	Name	Quellen
TW9	Unterstützung für Debugging	[1, p. 212]
Beschreibung Eine Unterstützung für Debugging von Seiten des Frameworks wäre wünschenswert, um die Analyse und das Nachstellen von Fehlerzuständen zu erleichtern.		

ID	Name	Quellen
TW10	Einsatz von Komparatoren	[1, p. 215]
Beschreibung Das Framework soll Komparatoren einsetzen, welche die erwarteten Resultate mit den tatsächlichen vergleichen und Abweichungen als Fehler ausweisen.		

2.2.5 Besonderheiten von Automatisierungslösungen

Nachdem im vorangehenden Kapitel die verschiedenen Arten und Ziele von Testwerkzeugen im Allgemeinen beleuchtet und Anforderungen an das Testframework aus diesen abgeleitet wurden, befasst sich dieses Kapitel im Speziellen mit dynamischen Testwerkzeugen im Kontext automatisierter funktionaler Systemtests und daraus resultierenden Anforderungen.

2.2.5.1 Eingrenzung dynamischer Testwerkzeuge auf automatisierte Testläufe

Wie bereits in Kapitel 2.2.4.5 „Werkzeuge für dynamisches Testen“ beschrieben, sind Werkzeuge für dynamische Tests nicht mit Testautomatisierungslösungen gleichzusetzen. Beide beziehen sich auf die Unterstützung von TesterInnen während der Ausführung des Testobjekts, allerdings kann diese Ausführung auch manuell erfolgen, wobei dynamische Testwerkzeuge dann zwar nicht die Testausführung selbst automatisieren, aber etwa durch objektive Messungen während der manuellen Testausführung hilfreich zur Seite stehen. Testautomatisierungslösungen können somit als Untermenge dynamischer Testwerkzeuge angesehen werden. Daher ist es nicht verwunderlich, dass einige der in diesem Unterkapitel gesammelten Anforderungen an das Testframework ursprünglich dem Kapitel 2.2.4.5 entspringen und bei Bedarf dort genauer nachgelesen werden können. Zur Bewahrung eines besseren Überblicks werden diese nun nochmals kurz aufgeführt:

Es handelt sich um die Forderungen, die Robustheit der automatisierten Testskripte gegenüber Änderungen, vor allem jenen in der graphischen Benutzeroberfläche des Programms, zu erhöhen, den Einsatz einer bekannten Skriptsprache für das Verfassen der Testskripte zu forcieren und die Testabdeckung zu messen.

2.2.5.2 Checkliste zur Evaluierung von Testautomatisierungslösungen

Elfriede Dustin, Thom Garret und Bernie Gauf [7] haben auf Basis ihrer jahrelangen Berufserfahrung im Bereich der Testautomatisierung eine Liste von Kriterien erstellt, die bei der Evaluierung und Auswahl einer Testautomatisierungslösung beachtet werden sollten. Einige der Anforderungen an das Testframework basieren auf dieser Liste, deren Inhalt im Folgenden in Auszügen wiedergegeben wird.

Viele Kriterien der Checkliste beziehen sich auf Softwarequalitätseigenschaften, welche in Kapitel 2.2.10 Softwarequalität gemäß ISO/IEC 25010:2011 näher beschrieben werden.

So sollen Produkte zur Testautomatisierung einfach zu installieren sein und eine saubere Deinstallation ermöglichen. Sie sollen darüber hinaus an die individuellen Gegebenheiten eines Softwareprojekts angepasst werden können und eine Cross-Plattform Kompatibilität zum Einsatz in verschiedenen Betriebssystemen und technologischen Umgebungen gewährleisten. Diese Punkte werden im Rahmen der Anforderung SQ8 „Portabilität“ ausführlicher behandelt.

Auch die Qualitätsanforderung SQ3 „Kompatibilität“ spielt in der Checkliste eine wichtige Rolle. Das Automatisierungswerkzeug soll etwa eine Integration in ein umfassenderes Framework oder den Qualitätsmanagementprozess an sich (etwa im Rahmen der Continuous Integration) auf effiziente Art und Weise gewährleisten, mit anderen Testwerkzeugen konfliktfrei zusammenarbeiten und im besten Fall die Stärken der jeweiligen Lösungen geschickt kombinieren.

Ein weiterer Punkt auf der Checkliste wurde bereits durch Anforderung TW1 „Steigerung der Testeffizienz“ im Kapitel 2.2.4.7 „Anforderungen im Bereich Testwerkzeuge“ abgedeckt. Dabei fordern die Autoren, dass auch die Fähigkeit einer Automatisierungslösung untersucht werden muss, die Zeit und Komplexität, welche für die Automatisierung nötig sind, zu reduzieren, was sich in einer Verminderung der für die Testautomatisierung eingeplanten Aufwände oder einer Erhöhung der Testabdeckung durch freigewordene Ressourcen niederschlägt.

Zudem sollte bei der Evaluierung von Testautomatisierungslösungen Wert auf die Vollständigkeit und Verständlichkeit der Dokumentation des Werkzeugs gelegt werden. Auch darauf, wieviel Information ein Testwerkzeug von TesterInnen benötigt, um eine Fehleranalyse durchzuführen, sollte geachtet werden. Im Idealfall sollte der menschliche Anteil an der Fehleranalyse möglichst niedrig oder nur einmalig erforderlich sein.

Weiters gilt es zu bedenken, dass die Technologien, mit denen Testobjekte erstellt werden, sich oftmals sehr rasch weiterentwickeln. Ein Testwerkzeug muss einen Weg finden, mit diesem Technologierückstand umzugehen, indem es sich bei Bedarf an technologische Neuerungen anpasst und erweiterbar konzipiert ist.

Das zentrale Kriterium von Testautomatisierungslösungen ist wohl ihre Testfähigkeit, also das Ausmaß, in welchem die Robustheit und Gründlichkeit des automatisierten Testens gesteigert werden, was sich in höher qualitativen Tests niederschlägt. Die Forderung, die Robustheit der Testfälle zu steigern, ist in Anforderung AU6 „Erhöhung der Robustheit automatisierter Tests“ am Ende dieses Kapitels abgebildet. Die Gründlichkeit des automatisierten Testens wird vor allem durch die Anforderungen AU2 „Erhöhung und Messung der Testabdeckung“, ebenfalls am Ende dieses Kapitels, sowie TW2 „Steigerung der Zuverlässigkeit von Tests“ im Rahmen des vorigen Kapitels, gewährleistet.

2.2.5.3 Weitere Ziele von Automatisierungslösungen

Der ISTQB Syllabus 2011 [29, p. 67] umreißt eine weitere bedeutende Anforderung an das Testframework, welche durch die Eigenschaften von Automatisierungslösungen wesentlich geprägt wird: die Konsistenz und die Wiederholbarkeit von Tests. Automatisierte Tests erlauben die Messung von Qualitätsmetriken und sind leicht wiederholbar, da dieselben Schritte unter denselben Bedingungen mit denselben Eingabedaten ausgeführt werden können. Beim manuellen Testen hingegen ist es wesentlich wahrscheinlicher, dass die

während des Testens durchgeführten Schritte sich in mehrmaligen Testdurchläufen nicht genau gleichen. Automatisierte Tests sind somit für die Analyse des Testprozesses essentiell, da sie einen sinnvollen Vergleich von Qualitätsmetriken erlauben, welcher nur unter gleichen Ausgangsbedingungen und unter Einhaltung einer sich exakt gleichenden Abfolge an Schritten möglich ist.

Darüber hinaus bietet auch das Kapitel 1.2.1 „Motivation und Risiken automatisierter Testausführung“ einen Fundus an Zielen, welche Automatisierungslösungen verfolgen und aus denen sich Anforderungen an das Testframework ableiten. Dazu zählen die Messung und die Erhöhung der Testabdeckung. Während die Messung der Testabdeckung in einem Black-Box-basierten Test beispielsweise durch die Erhebung der Rate automatisiert validierter Requirements im Vergleich zu allen definierten Requirements erfolgen kann, basiert die Erhöhung der Testabdeckung auf dem Potenzial automatisierter Tests, mehr Kombinationen an Datenvariationen und Testszenarios in derselben Zeit zu überprüfen.

Schließlich ist auch die Unterstützung von Regressionstests ein Ziel von Testautomatisierungslösungen. Diese werden nach Änderungen an der Softwareanwendung ausgeführt, um sicherzustellen, dass keine neuen Fehler in das Produkt eingeführt wurden. Da Regressionstests eine langwierige, sich oft wiederholende, monotone und ermüdende Tätigkeit darstellen, sollten sie automatisiert durchgeführt werden. Dadurch kann sowohl der Aufwand für die Testausführung reduziert als auch dem Effekt der „Tester fatigue“ entgegengewirkt werden, welcher sich in einer Immunität der TesterInnen gegenüber Fehlern manifestiert (siehe TW2 „Steigerung der Zuverlässigkeit von Tests“).

2.2.5.4 Anforderungen im Bereich Testautomatisierungslösungen

ID	Name	Quellen
AU1	Konsistenz und Wiederholbarkeit der automatisierten Tests	[5]
Beschreibung Das Framework soll die Einhaltung exakt gleichbleibender Bedingungen und Schrittabfolgen in mehrmaligen Testläufen gewährleisten, um eine sinnvolle Analyse von Qualitätsmerkmalen im Verlauf des Testprozesses zu ermöglichen.		

ID	Name	Quellen
AU2	Erhöhung und Messung der Testabdeckung	[1, p. 215] [5]
Beschreibung Die Messung der Testabdeckung durch das Framework soll gegeben sein. Zugleich soll eine größere Testabdeckung durch den Einsatz des Frameworks erzielt werden, indem dieses die Überprüfung einer größeren Anzahl an Kombinationen von Datenvariationen und Testszenarien in derselben Zeit ermöglicht.		

ID	Name	Quellen
AU3	Vollständige und verständliche Dokumentation	[7]
Beschreibung Das Framework soll über eine vollständige und verständliche Dokumentation verfügen, um AnwenderInnen das Erlernen und den Umgang mit dem Framework zu erleichtern.		

ID	Name	Quellen
AU4	Senkung des menschlichen Anteils an der Ergebnisanalyse	[7]
Beschreibung Das Versorgen der Automatisierungslösung mit Informationen, die diese benötigt, um die Korrektheit von Testergebnissen zu analysieren, soll mit einem möglichst geringen Aufwand verbunden sein.		

ID	Name	Quellen
AU5	Kompensation von künftigem Technologie-Rückstand	[7]
Beschreibung Da sich die Technologien, auf denen Testobjekte aufbauen, rasch weiterentwickeln, muss auch das Testframework versuchen, einen technologischen Rückstand zu vermeiden, so dass eine Testbarkeit der Testobjekte auch in Zukunft gewährleistet ist.		

ID	Name	Quellen
AU6	Erhöhung der Robustheit automatisierter Tests	[1, p. 213]
Beschreibung Um die Qualität der automatisierten Testfälle zu erhöhen, soll deren Robustheit gesteigert werden, sodass deren Funktionalität durch Änderungen im Aufbau der GUI oder durch unerwartete Ereignisse nicht so leicht beeinträchtigt wird.		

ID	Name	Quellen
AU7	Unterstützung von Regressionstests	[2, p. 223] [6, p. 478] [5]
Beschreibung Das Framework soll die Durchführung von Regressionstests mit geringem Aufwand unterstützen, um sicherzustellen, dass Änderungen am Testobjekt keine unerwünschten Fehler oder Nebenwirkungen in diesem oder im Rest der Software mit sich bringen.		

ID	Name	Quellen
AU8	Einsatz einer bekannten Skriptsprache für Testskripte	[1, p. 213]
Beschreibung Der Einsatz einer bekannten Skriptsprache für die Erstellung der Testskripte soll es AnwenderInnen erleichtern, den Umgang mit dem Testframework zu erlernen und ihnen die Verwendung wohl bekannter Sprachkonstrukte und -konzepte ermöglichen.		

2.2.6 Gestaltung von Anforderungen

2.2.6.1 Bedeutung von Anforderungen

Die Anforderungen einer Software dienen allen an einem Softwareprojekt Beteiligten als gemeinsame Diskussionsgrundlage und sind somit für die erfolgreiche Umsetzung des Softwareprojekts von essentieller Bedeutung. Im Rahmen der Abnahme dient die Anforderungsspezifikation als Basis, um Abweichungen zwischen dem Ist- und dem Sollzustand der erstellten Software festzustellen. Zudem sind korrekt und vollständig formulierte Anforderungen ein wichtiger Faktor für eine hohe Kundenzufriedenheit, denn sie sorgen dafür, dass Kundenwünsche und deren Optimierungspotenziale erhoben und in das Softwaresystem integriert werden. [18, pp. 18-20]

Christ Rupp und den Sophisten [18, p. 24] zufolge gewinnen InformatikerInnen durch aktive Kommunikation mit KundInnen Anforderungen in natürlicher Sprache. Diese sind oft inkonsistent und unvollständig. Die Aufgabe der InformatikerInnen ist es nun, daraus ein konsistentes, vollständiges und formales Modell der Geschäftsprozesse des Kunden/der Kundin abzuleiten.

Dass der Prozess der Anforderungsdefinition eine sehr fordernde und oft fehleranfällige Aufgabe darstellt, zeigen folgende von Georg Erwin Thaller [30, pp. 93, 100] präsentierte Werte: durchschnittlich 22,5 von 50 Fehlern pro tausend Zeilen Code entstammen über die gesamte Softwarebranche hinweg den Phasen der Anforderungsdefinition und des Designs. Beinahe die Hälfte aller in Software enthaltenen Fehler sind also bereits vor Verfassung des Programmcodes entstanden. Viele dieser frühen Fehler könnten wohl durch besser formulierte Anforderungen vermieden werden, wie die Verteilung von Fehlerursachen in der Entwurfsphase in Tabelle 1 zeigt:

Ursache des Fehlers	Anteil in Prozent
Vergessene Fälle oder Schritte	20
Nicht beachtete Grenzfälle	17
Falsch verstandener Entwurf	14
Unzureichendes Überprüfen des Entwurfs	12
Falsch verstandene Spezifikation	12
Fehler in der Reihenfolge	8
Fehler in der Kommunikation	7

Tabelle 1: Aufteilung von Fehlern im Entwurf (Quelle: [30, p. 100])

Capers Jones, [16, p. 475], stellte zudem fest, dass die schwerwiegendsten Fehler in Software vorwiegend in der Phase der Anforderungsdefinition eingebracht werden.

Betrachtet man die in Abbildung 12 dargestellte Barry Boehm Kurve, welche die Entwicklung von Fehlerkosten über die fortlaufenden Phasen eines Softwareentwicklungsprojektes beschreibt, ist es ökonomisch äußerst sinnvoll, eine hohe Qualität der Anforderungen anzustreben, um die Anzahl der Fehler in der Anforderungsdefinition und in den auf den Anforderungen basierenden Designentwürfen zu senken. Frank Witte [31, pp. 89-92] beschreibt die Größenordnung von Kosten, wenn Fehler erst in späteren Projektphasen gefunden und behoben werden, in einer Faustregel: Fehler, die während der Entwicklung entdeckt werden, sind um den Faktor 10, jene die im Test aufgedeckt werden um den Faktor 100, und solche, die erst im Produktivbetrieb zu Tage treten um den Faktor 1000 teurer als Fehler, die bereits während der Analyse und dem Design entdeckt und korrigiert werden.

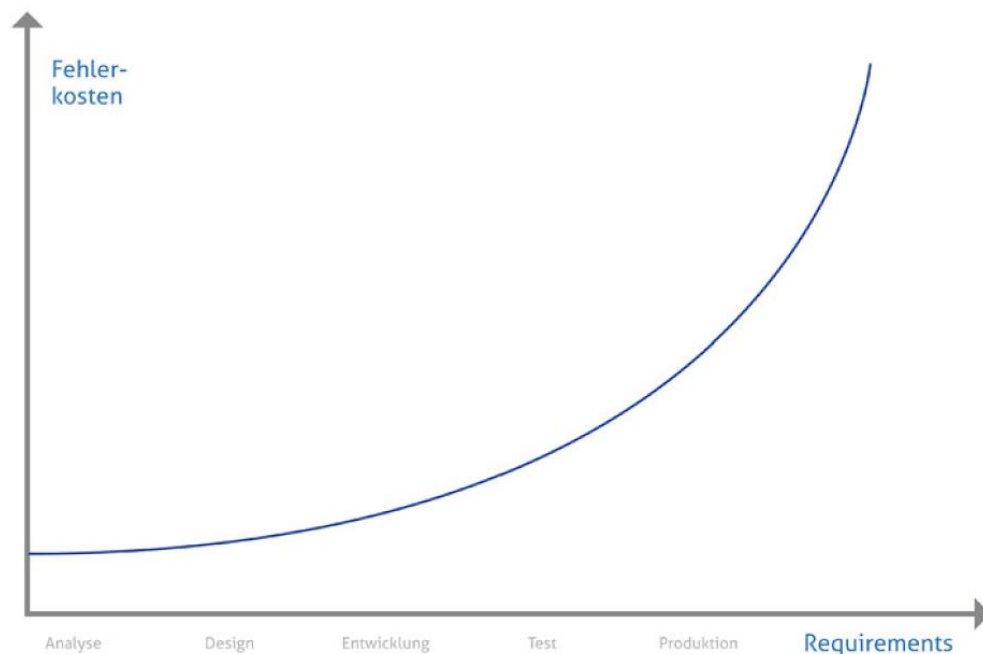


Abbildung 12: Fehlerkostenentwicklung fortlaufender Projektphasen (Quelle: [31, p. 88])

2.2.6.2 Qualitätsansprüche für Software Anforderungen

Christ Rupp und die Sophisten [18, pp. 26-30], sowie Pericles Loucopoulos und Bill Karakostas [19, pp. 70-74], beschreiben Kriterien zur Steigerung der Anforderungsqualität und beziehen sich dabei größtenteils auf die Norm IEEE 830 „Recommended Practice for Software Requirements Specifications“ – diese wurde zwar inzwischen durch die Norm ISO/IEC/IEEE 29148:2011 ersetzt, sie bietet jedoch immer noch einen guten Anhaltspunkt für die Formulierung qualitativ hochwertiger Anforderungen. Dieses Unterkapitel listet einige der von Rupp, Loucopoulos und Karakostas beschriebenen Qualitätskriterien für Anforderungen auf und gruppiert diese nach syntaktisch-formalen und semantisch-inhaltlichen Aspekten sowie nach Gesichtspunkten der Weiterverwendung.

Syntaktisch-formale Aspekte

- **Konsistenz:** Anforderungen müssen in sich und gegenüber anderen Anforderungen widerspruchsfrei sein. Gerade wenn mehrere Benutzer mit unterschiedlichen Erwartungen an ein System befragt werden, stellt dieses Kriterium eine Herausforderung dar. Ein Hilfsmittel zur Sicherstellung der Konsistenz ist, die Anforderungen als logische Sätze zu formulieren, die entweder wahr oder falsch und mit logischen Operatoren wie „and“, „or“ sowie „not“ miteinander verknüpft sind.
- **Verfolgbarkeit:** Eine Anforderung sollte eindeutig zu identifizieren und in Design, Implementierung und Test verfolgbar sein. Dies wird meist über eine eindeutige Anforderungsnummer sichergestellt, die während des gesamten Lebenszyklus der Anforderung unverändert bleibt.
- **Eindeutigkeit:** Anforderungen dürfen nur auf eine Art und Weise verstanden werden. Es soll vermieden werden, dass LeserInnen einer Anforderung diese unterschiedlich interpretieren. Gerade in natürlicher Sprache finden sich jedoch immer wieder Mehrdeutigkeiten. Solche Mehrdeutigkeiten können mithilfe logischer Regeln eliminiert werden.

Semantisch-inhaltliche Aspekte

- **Korrektheit:** Eine Anforderung ist korrekt, wenn ein Sachverhalt in der Problemdomäne mit den Beschreibungen im Anforderungsmodell übereinstimmt. Um dies sicherzustellen, müssen die Stakeholder die Anforderungsspezifikation lesen und verstehen können.
- **Vollständigkeit:** Damit ein Anforderungsmodell als vollständig bezeichnet werden kann, darf es keine wichtigen Informationen auslassen, deren Fehlen dazu führen könnte, dass das System nicht die Bedürfnisse der BenutzerInnen erfüllt. Prototypen können zur Sicherstellung der Vollständigkeit von Anforderungen eingesetzt werden. Zudem können noch unvollständige Anforderungen durch Marker als solche gekennzeichnet und später systematisch gesucht und durch fehlende Informationen ersetzt werden.
- **Gültigkeit:** Anforderungen müssen aktuell gehalten werden, um stets die Realität des Systems zu beschreiben.
- **Redundanzfreiheit:** Anforderungen sollten idealerweise nur an genau einer Stelle im Anforderungsmodell identifiziert werden können. Ändern sich nämlich einzelne Funktionalitäten in redundanten Anforderungen, müssen diese an allen Stellen angepasst werden. Wird dies übersehen, ergeben sich Inkonsistenzen im Anforderungsmodell. Daher gilt es, redundante Anforderungen zu vermeiden.

- **Minimalität:** Die Anforderungen sind nicht der geeignete Ort, um die detaillierte technische Lösung eines Problems zu beschreiben. Vielmehr sollte dies im technischen Systementwurf oder in der technischen Systembeschreibung geschehen. Im Anforderungsmodell sollte diese Form der Überspezifizierung hingegen vermieden werden, um die diversen verfügbaren Lösungsmöglichkeiten eines Problems nicht von vornherein einzuschränken.

Gesichtspunkte der Weiterverwendung

- **Verständlichkeit:** Anforderungen müssen von allen Stakeholdern verstanden werden können. Dazu ist es erforderlich, eine gemeinsame Sprache für alle zu schaffen. Diese besteht oft aus domänenspezifischen Begriffen die in einen Formalisierungsprozess eingebunden werden. Das Ergebnis ist eine pseudo-natürliche Sprache mit logischen Phrasierungsregeln.
- **Bewertbarkeit/Gewichtbarkeit:** Gerade in größeren Softwareprojekten, bei denen nicht alle Funktionalitäten innerhalb eines Releases umgesetzt werden können, müssen Stakeholder eine Gewichtung der Anforderungen durchführen können, um die Priorität deren Umsetzung festzulegen.
- **Prüfbarkeit:** Anforderungen müssen so formuliert werden, dass sie testbar sind. Funktionalitäten, die durch Anforderungen beschrieben werden, müssen sich somit durch Messungen nachweisen lassen.
- **Umsetzbarkeit:** Eine Anforderung muss innerhalb der bekannten Fähigkeiten und Grenzen des Systems sowie seiner Umgebung umgesetzt werden können. Es sollten Mitglieder des Entwicklungsteams an der Bewertung von Zielen und Anforderungen beteiligt sein, um die technologischen Grenzen der Umsetzung und die damit verbundenen Kosten aufzuzeigen, denn nicht selten treten Projektbetroffene von Anforderungen zurück, wenn diese zu hohe Kosten verursachen.

2.2.6.3 Zusätzliche Ansprüche an Anforderungen

Obwohl jedes Softwareprojekt über Anforderungen verfügt – schließlich setzen Stakeholder gewisse Erwartungen in das Projekt – werden diese manchmal nicht niedergeschrieben und existieren nur in den Köpfen der Beteiligten. Die einzige Möglichkeit, eine solche Software zu testen, ist exploratives Testen. Spillner et al. [1, p. 60] warnen ausdrücklich vor diesen „imaginären“ Anforderungen, denn sie nötigen TesterInnen in dem Fall, dass sie weitere Testmethoden durchführen möchten, dazu, in zeitaufwendigen Gesprächen mit den Stakeholdern diese Anforderungen zu sammeln. Aufgrund meist widersprüchlicher Aussagen müssen die TesterInnen dann Entscheidungen über die Auslegung der gesammelten Informationen fällen, die bereits lange Zeit zuvor von anderen Personen hätten getroffen werden sollen. Neben beträchtlichen Verzögerungen im gesamten Projektablauf verfügen in diesen Fällen auch die EntwicklerInnen über keine klaren Zielvorstellungen, und das Projekt ist zum Scheitern verurteilt. Deshalb wird die Niederschrift von Anforderungen in der Phase der Anforderungsdefinition als Forderung an das Testframework aufgenommen.

Zum Schluss dieses Kapitels werden zwei Eigenschaften betrachtet, die eine Modellierungssprache für Anforderungen aufweisen sollte. Loucopoulos und Karakostas definieren diese in ihrem Buch „System Requirements Engineering“ [19, pp. 77-78].

So sollte die Modellierungssprache Rückschlüsse auf den Zustand eines Systems und die Übergänge zwischen den einzelnen Systemzuständen ermöglichen. Die Autoren schlagen zu diesem Zweck die Verwendung der logischen Programmiersprache Prolog vor, weisen jedoch darauf hin, dass viele BenutzerInnen rein logische Spezifikationen schwer verständlich finden. Zur Lösung können die Prolog Definitionen in eine pseudo-natürliche Sprache, die von logischen „IF THEN ELSE“ Regeln durchzogen ist, umformuliert werden. Schließlich muss bei Verwendung der Modellierungssprache auf im Anforderungsmodell versteckte Informationen rückgeschlossen werden können. Dadurch können Anforderungen in unterschiedlichen Abstraktionsstufen beschrieben werden. Definiert eine Anforderung beispielsweise, welche Daten BenutzerInnen für eine gültige Essenbestellung eingeben müssen, so können weitere Anforderungen diese wiederverwenden, ohne die detaillierten Schritte der Essensbestellung erneut zu beschreiben. So könnte die erste Anforderung lauten: „BenutzerInnen müssen, um eine gültige Essenbestellung durchführen zu können, ihren Vor- und Nachnamen, sowie ihre Lieferadresse in das Bestellformular eintragen“. In einer zweiten Anforderung kann dann auf die Bedingungen der ersten Anforderung rückgeschlossen werden: „Gültige Essensbestellungen müssen mit dem Zeitpunkt der Bestellaufnahme in die Datenbank eingetragen werden.“

2.2.6.4 Anforderungen im Bereich Gestaltung von Anforderungen

ID	Name	Quellen
AN1	Anforderungen müssen niedergeschrieben sein	[1, p. 60]
Beschreibung Um in den Testmethoden nicht auf exploratives Testen beschränkt zu sein, müssen Anforderungen ausformuliert und niedergeschrieben werden. Dies ist eines der wichtigsten Kriterien, um ein Softwareprojekt erfolgreich umzusetzen.		

ID	Name	Quellen
AN2	Konsistenz innerhalb der Anforderungen	[18, p. 28] [19, p. 70]
Beschreibung Innerhalb von Anforderungen darf es keine Widersprüche geben. Um dies zu gewährleisten, können auf Logik basierende Sätze, welche entweder das Ergebnis „wahr“ oder das Ergebnis „falsch“ zulassen und durch logische Operatoren verknüpft sind, eingesetzt werden.		

ID	Name	Quellen
AN3	Verfolgbarkeit der Anforderungen	[18, p. 29]
Beschreibung Anforderungen müssen über ihre gesamte Lebenszeit hinweg mit derselben eindeutigen Nummer identifizierbar sein und sollen durch diese in Design, Implementierung und Test verfolgt werden können.		

ID	Name	Quellen
AN4	Eindeutigkeit der Anforderungen	[18, p. 28] [19, pp. 70-71]
Beschreibung Anforderungen dürfen keinen Interpretationsspielraum bieten. Die Mehrdeutigkeiten, welche natürlichen Sprachen oft innewohnen, können durch logische Regeln eliminiert werden.		

ID	Name	Quellen
AN5	Korrektheit der Anforderungen	[18, p. 28] [19, pp. 71-72]
Beschreibung Eine Anforderung ist korrekt, wenn sie den tatsächlichen Sachverhalt in der Realität widerspiegelt. Um sicherzustellen, dass eine Anforderung korrekt ist, muss diese von Stakeholdern gelesen und verstanden werden können.		

ID AN6	Name Vollständigkeit der Anforderungen	Quellen [18, p. 28] [19, pp. 71-72]
Beschreibung Zur Wahrung der Vollständigkeit von Anforderungen werden oftmals Prototypen eingesetzt. Diese sollen verhindern, dass Informationen, die erforderlich sind, damit das System die Bedürfnisse der BenutzerInnen erfüllt, im Anforderungsmodell fehlen.		

ID AN7	Name Gültigkeit der Anforderungen	Quellen [18, p. 29]
Beschreibung Anforderungen sind nur dann gültig, wenn sie aktuell gehalten werden, um stets die Realität des Systems zu beschreiben.		

ID AN8	Name Verständlichkeit der Anforderungen	Quellen [18, p. 29] [19, pp. 77-78, 83]
Beschreibung Anforderungen sollen von allen Stakeholdern verstanden werden. Dazu muss eine gemeinsame Sprache zur Anforderungsdefinition für alle Beteiligten geschaffen werden.		

ID AN9	Name Bewertbarkeit und Gewichtbarkeit der Anforderungen	Quellen [18, p. 30]
Beschreibung Anforderungen sollen gewichtbar sein, so dass eine Priorität für deren Umsetzung festgelegt werden kann.		

ID AN10	Name Prüfbarkeit der Anforderungen	Quellen [18, p. 28]
Beschreibung Alle Funktionalitäten, die durch Anforderungen beschrieben werden, müssen sich durch Messungen nachweisen lassen. Nur dann sind Anforderungen wirklich prüfbar.		

ID AN11	Name Umsetzbarkeit der Anforderungen	Quellen [18, p. 29]
Beschreibung Eine Anforderung muss innerhalb der bekannten Fähigkeiten und Grenzen des Systems sowie seiner Umgebung umgesetzt werden können. Um die technologische Machbarkeit und die Kosten zur Umsetzung einer Anforderung abschätzen zu können, sollten EntwicklerInnen an der Bewertung von Zielen und Anforderungen beteiligt sein.		

ID	Name	Quellen
AN12	Minimalität der Anforderungen	[19, p. 72]
Beschreibung Eine technische Überspezifizierung der Anforderungen sollte vermieden werden, um die diversen verfügbaren Lösungsmöglichkeiten eines Problems nicht von vornherein einzuschränken.		

ID	Name	Quellen
AN13	Vermeidung von Redundanzen in den Anforderungen	[18, p. 26] [19, p. 75]
Beschreibung Da sich Softwareanforderungen oft ändern, sollten redundanten Anforderungen vermieden werden, damit diese nicht immer an mehreren Stellen nachgezogen werden müssen. Dadurch sinkt auch die Anfälligkeit der Anforderungen für Inkonsistenzen.		

ID	Name	Quellen
AN14	Anforderungsmodell soll Vorstellung von Zuständen und Zustandsübergängen ermöglichen	[19, pp. 77-78]
Beschreibung Eine Modellierungssprache für Anforderungen sollte es ermöglichen, Zustände und Zustandsübergänge des Systems aus dem Anforderungsmodell abzuleiten.		

ID	Name	Quellen
AN15	Auf implizierte Anforderungsinformationen soll rückgeschlossen werden können	[19, pp. 77-78]
Beschreibung Anforderungen werden oftmals auf mehreren abstrakten Ebenen beschrieben. Informationen, die in Anforderungen konkreter Ebenen detailliert erläutert werden, sollen in Anforderungen abstrakterer Ebenen rückgeschlossen werden können.		

2.2.7 Traceability

2.2.7.1 Definition von Traceability

Als Traceability wird die Nachvollziehbarkeit von Informationen und deren Abhängigkeiten definiert. Diese ist im Änderungsmanagement von großer Bedeutung, da sie Artefakte, zwischen denen Abhängigkeiten bestehen, mit sogenannten „Traces“ versieht und so miteinander verknüpft. Dadurch können beispielsweise die von einer geänderten Anforderung betroffenen Passagen im Programmcode ausfindig gemacht und an die neue Funktionalität angepasst werden.

Traceability ermöglicht es somit, die Auswirkungen von Änderungen wesentlich einfacher zu verfolgen und Widersprüchlichkeiten schneller aufzudecken, da nur die für die Ergänzung, Änderung oder Löschung von Informationen relevanten Bereiche betrachtet werden müssen. Dies führt zu einer Erhöhung der Konsistenz verknüpfter Artefakte und zu einer Reduzierung des für die Erstellung, Pflege und Auswertung von Abhängigkeiten zwischen Artefakten benötigten Aufwands. [18, p. 410]

2.2.7.2 Voraussetzungen für Traceability

Um Traces (Verweise) für Anforderungen aufbauen und definieren zu können, müssen gemäß Rupp [18, pp. 411-412] eine ganze Reihe von Voraussetzungen erfüllt werden. Einige dieser Voraussetzungen, die für das implementierte Testframework besonders relevant sind, werden nun näher betrachtet:

- Die wichtigste Grundlage für Traceability ist, dass eine Anforderung immer eindeutig identifiziert werden kann - etwa durch eine nur einmalig vergebene Nummer.
- Darüber hinaus sollte auch das Ziel eines Verweises immer ersichtlich sein. Dazu können die Traces mit sogenannten Linkattributen, welche die Art des Verweises beschreiben, versehen werden.
- Da die Menge theoretisch möglicher Traces schon bei einfachen System sehr groß werden kann, und die Einführung von Traceability auch einen gewissen Aufwand mit sich bringt, sollte diese auf das Wesentliche beschränkt werden. Sowohl die Detaillierungsebene als auch die Arten von Artefakten, zwischen denen Verweise aufgebaut werden, müssen wohl überlegt sein und einen positiven Payoff mit sich bringen.
- Schließlich müssen Traces stetig gepflegt und unmittelbar aktualisiert werden. Passiert dies nicht, werden oftmals notwendige Aktualisierungen nicht vorgenommen und die Anzahl falscher oder fehlender Traces nimmt stark zu, was eine effiziente Nutzung und Auswertung unmöglich macht.

2.2.7.3 Festlegung der durch Traceability erfassten Artefakte

Gemäß der Forderung, die Traceability auf das Wesentliche zu beschränken, wird in diesem Unterkapitel festgelegt, welche Arten von Artefakten durch das Testframework mit Verweisen versehen werden sollten.

Sommerville [2, pp. 224-225] sowie Spillner et al. [1, p. 70] machen darauf aufmerksam, dass für anforderungsbasiertes Testen eine Verknüpfung von Anforderungen und Testfällen zwingend erforderlich ist. Zwischen diesen beiden Artefakten sollten also in jedem Fall Verweise angelegt werden.

Spillner et al. [1, p. 208] fordern zudem, dass Testfälle mit einem Incident Management Tool gekoppelt sein sollten, um eine Liste all jener Testfälle generieren zu können, die nochmals ausgeführt werden müssen, wenn die von den Testfällen aufgedeckten Fehler behoben wurden. Traceability muss somit zwischen Testfällen und Defects gegeben sein.

Zuletzt weisen Christ Rupp und die Sophisten [18, p. 410] darauf hin, dass auch Anforderungen unterschiedlicher Ebenen durch Traces nachvollziehbar sein sollten. Dies ist beispielsweise nützlich, um User Stories mit der funktionalen Systembeschreibung in Verbindung zu bringen.

2.2.7.4 Anforderungen im Bereich Traceability

ID	Name	Quellen
TR1	Traceability zwischen Anforderungen unterschiedlicher Ebenen	[18, p. 410]
Beschreibung Das Framework soll eine Nachvollziehbarkeit zwischen Anforderungen unterschiedlicher Ebenen, etwa User Stories und der funktionalen Systembeschreibung, ermöglichen.		

ID	Name	Quellen
TR2	Eindeutige Identifikation von Artefakten muss möglich sein	[18, p. 411]
Beschreibung Die wichtigste Grundlage zur Umsetzung von Traceability ist, dass Artefakte eindeutig identifiziert werden können. Dies kann etwa durch eine einmalig vergebene Nummer sichergestellt werden.		

ID	Name	Quellen
TR3	Das Ziel eines Verweises soll ersichtlich sein	[18, p. 411]
Beschreibung Es sollte immer ersichtlich sein, auf welche Art von Ziel ein Verweis zeigt. Dazu kann dieser mit sogenannten Linkattributen versehen werden.		

ID	Name	Quellen
TR4	Der Aufwand für Traceability soll gering gehalten werden	[18, p. 412]
Beschreibung Die Umsetzung von Traceability in den diversen Artefakten eines Softwareprojekts ist immer mit einem gewissen Aufwand verbunden. Dieser soll so gering wie möglich gehalten werden, damit sich Traceability ökonomisch rentiert.		

ID	Name	Quellen
TR5	Traceability zwischen Anforderungen und Testfällen	[2, pp. 224-225] [1, p. 70]
Beschreibung Zur Durchführung von Regressionstests müssen die Anforderungen mit den Testfällen, die diese validieren, verknüpft sein.		

ID	Name	Quellen
TR6	Traceability zwischen Testfällen und Defects	[1, p. 208]
Beschreibung Verweise zwischen Testfällen und den Defects in einem Incident Management Tool erlauben, eine Liste aller Testfälle zu generieren, die erneut ausgeführt werden müssen, um zu prüfen, ob korrigierte Fehler tatsächlich behoben wurden.		

2.2.8 Ansprüche an Reports

2.2.8.1 Die Rolle von Reports für Testmanagement Werkzeuge und den Testprozess

Die Anforderungen an Reports nehmen im Vergleich zu den anderen Teilbereichen der Literaturrecherche eine gewisse Sonderrolle ein, denn sie basieren zur Gänze auf Aspekten, die im Rahmen der Kapitel 2.2.2.1 „Aktivitäten des Testprozesses“ sowie 2.2.4.2 „Werkzeuge für Testmanagement“ bereits untersucht wurden. Dies liegt daran, dass die hauptsächliche Aufgabe von Reports darin besteht, das Management und die Steuerung von Tests, welche sich über alle Aktivitäten des Testprozesses erstrecken, zu unterstützen. Da der Fokus der beiden obigen Kapitel jedoch nicht auf die Forderungen an Reports gerichtet ist, werden diese im Verlauf dieses Unterkapitels gezielter behandelt.

Wie bereits erwähnt übernehmen Reports in der Teststeuerung eine wichtige Aufgabe, denn sie sammeln Daten, die für die Teststeuerung entscheidungsrelevant sind. Spillner et al. [1, pp. 26-29, 208-209] erwähnen beispielsweise folgende Kriterien:

- unterschiedliche Formen der Testabdeckung, wie etwa im funktionalen Systemtest das Verhältnis bereits automatisiert validierter im Vergleich zu allen definierten Akzeptanzkriterien
- die für den Durchlauf der Testfälle benötigte Zeit sowie das Zeitverhalten der einzelnen Schritte eines Testskripts
- die Gegenüberstellung von fehlgeschlagenen und erfolgreichen Testfällen
- die Fehlerrate als Anzahl der pro Teststunde neu aufgedeckten Fehler, sowie deren Schwere und Kategorie

Anhand dieser Informationen können die Testziele auf deren Einhaltung überprüft und bei Bedarf Maßnahmen zur Gegensteuerung eingeleitet werden. Zudem wird auf Basis der gesammelten Daten beurteilt, ob die Testendekriterien erreicht oder eventuell sogar angepasst werden müssen, wenn der für das Erreichen der Testendekriterien nötige Aufwand als nicht mehr gerechtfertigt betrachtet wird.

Treten beim Testen Abweichungen zwischen den erwarteten und den tatsächlichen Resultaten auf, muss untersucht werden, ob es sich hierbei tatsächlich um Fehler in der Anwendung handelt. Spillner et al. [1, pp. 26-27] zufolge können nämlich auch fehlerhafte oder ungenaue Testfallbeschreibungen, Probleme mit der Infrastruktur oder inkorrekte Testausführung die Ursachen für ein Fehlverhalten im Test darstellen. Reports dienen bei der Fehleranalyse als wichtiges Hilfsmittel.

Eine wichtige Voraussetzung hierfür ist die exakte Reproduzierbarkeit eines fehlerhaften Testfalls. Diese wird durch ein möglichst vollständiges und genaues Logging der Testausführung im Rahmen von Reports gewährleistet. Die geloggten Informationen sollen Auskunft darüber geben, welche Person oder welches System welche Teile der Software zu welchem Zeitpunkt, in welcher Konfiguration und Umgebung, wie intensiv und mit welchen Ergebnissen getestet hat. Sie sollen darüber hinaus so präsentiert werden, dass sie auch für Personen, die nicht direkt mit dem Testen zu tun haben, verständlich sind und dadurch zu einer verbesserten Dokumentation des Testprozesses beitragen. [5], [1, pp. 26, 208]

Neben dem Logging der Testausführung fordern Spillner et al. [1, pp. 30, 208-209] zudem die Möglichkeit, im Rahmen des Reportings anhand der gesammelten Daten die komplette Testdokumentation, welche Testplan, Testspezifikation und Testabschlussbericht umfasst, zu generieren. In Kürze zusammengefasst legt der Testplan den Umfang, die benötigten Ressourcen und die Aktivitäten des Testens fest, die Testspezifikation beschreibt die logischen und konkreten Testfälle, und der Testabschlussbericht gibt nach Abschluss der Testvorgänge allen Stakeholdern Auskunft über die durchgeführten Aktivitäten und die Resultate des Testens.

2.2.8.2 Anforderungen im Bereich Reports

ID	Name	Quellen
RE1	Sammeln von Maßen zur Teststeuerung	[1, pp. 26-29, 208-209]
Beschreibung Die Testabdeckung, das Zeitverhalten von Testfällen, die Fehlerrate und Schwere neu aufgedeckter Fehler sowie das Verhältnis von erfolgreichen und fehlgeschlagenen Testfällen stellen entscheidungsrelevante Informationen für die Teststeuerung dar und dienen etwa als Grundlage für die Beurteilung der Testendekriterien. Sie sollen im Rahmen des Reportings erfasst werden.		

ID	Name	Quellen
RE2	Unterstützung der Fehleranalyse sowie der Festlegung von Schwere und Priorität der Behebung von Fehlern	[1, pp. 26-27]
Beschreibung Nicht jede Abweichung zwischen erwarteten und tatsächlichen Resultaten in Testfällen hat ihre Ursache innerhalb der Softwareanwendung. Oft kommen auch falsche Testfälle oder Probleme mit der Testinfrastruktur als Fehlerursachen in Frage. Reports sollen bei der Analyse dieser Fehler helfen und zudem die Priorisierung und die Bestimmung der Schwere „echter“ Fehler unterstützen.		

ID	Name	Quellen
RE3	Möglichkeit zur Erstellung kompletter Testdokumentation	[1, pp. 30, 208-209]
Beschreibung Anhand der in den Reports gesammelten Daten soll die komplette Testdokumentation, einschließlich Testplan, Testspezifikation und Testabschlussbericht, generiert werden können. Diese muss für alle Stakeholder verständlich präsentiert werden.		

ID	Name	Quellen
RE4	Exaktes und vollständiges Logging der Testausführung	[1, pp. 26, 208] [5]
Beschreibung Reports sollen durch exaktes und vollständiges Logging der Testausführung Auskunft darüber geben, wer wann welche Stellen einer Applikation wie genau, in welcher Umgebung und mit welchen Resultaten getestet hat. Im Sinne eines gründlichen Configuration Managements sollen darüber hinaus die Versionen der Testobjekte in der betesten Konfiguration ersichtlich sein. Neben einer Erleichterung der Analyse von Fehlern steigt somit auch die Reproduzierbarkeit fehlgeschlagener Testfälle.		

2.2.9 Design eines Software Frameworks

Dieses Kapitel erläutert die Motivation für den Einsatz von Software Frameworks anhand jener Eigenschaften, die ein qualitativ hochwertig designtes Framework auszeichnen. Die gewonnenen Erkenntnisse fließen als Anforderungen an das im Rahmen dieser Arbeit entwickelte Testframework mit ein.

2.2.9.1 Frameworks als treibende Kraft der Wiederverwendbarkeit

Ian Sommerville [2, p. 431] sieht in Frameworks die treibende Kraft für die Wiederverwendbarkeit in objektorientierten Entwicklungsprozessen. Wie er zu dieser Aussage kommt, erklärt er anhand der geschichtlichen Entwicklung objektorientierter Programmierung: Deren ursprünglicher Anspruch, Objekte in unterschiedlichen Systemen wiederzuverwenden, scheiterte in der Praxis aufgrund der Tatsache, dass diese meist zu sehr auf die Bedürfnisse eines bestimmten Systems zugeschnitten waren. Aus dieser Notlage heraus wurde die Idee geboren, grobgranularere Abstraktionen zu schaffen, die als Vorlage für die Lösung geläufiger Probleme einer bestimmten Domäne eingesetzt werden konnten. Dies war die Geburtsstunde von Software Frameworks.

Frameworks bilden also eine generische Struktur von Funktionalitäten, die in allen Anwendungen eines ähnlichen Typs einsetzbar sind. Sie stellen EntwicklerInnen ein architektonisches Grundgerüst sowie wiederverwendbare, abstrakte Basisklassen zur Verfügung, welche bei Bedarf konkretisiert und so an die spezifischen Anforderungen einer Anwendung angepasst werden können. Darüber hinaus können sich Frameworks gemäß einer modularen Bauweise aus spezialisierteren Frameworks zusammensetzen, die jeweils einen ganz spezifischen Teil der Entwicklung unterstützen. Sie stellen somit einen effizienten Ansatz von Wiederverwendung dar, bringen aber für gewöhnlich eine nicht zu unterschätzende Einarbeitungsphase mit sich. [2, pp. 431-434]

Sommerville [2, p. 194] unterscheidet zwischen drei Ebenen der Wiederverwendbarkeit, welche in Frameworks eine Rolle spielen:

- Auf **Abstraktionsebene** fördern Frameworks die Wiederverwendbarkeit vor allem in Form von Entwurfs- und Architekturmustern. Diese stellen bewährte Lösungen für bekannte Entwurfsprobleme im Rahmen eines bestimmten Kontexts zur Verfügung und tragen maßgeblich zur Verständlichkeit einer Architektur bei, wenn EntwicklerInnen auf ihnen bereits bekannte Muster stoßen.
- Auf **Objektebene** ermöglichen Frameworks die direkte Wiederverwendung von Objekten einer Softwarebibliothek. Dazu zählen etwa sogenannte „Utility Functions“.
- Auf **Komponentenebene** bieten Frameworks Sammlungen von Objekten und Objektklassen an, welche zusammenarbeiten, um eine bestimmte Funktionalität zu

gewährleisten. Diese Komponenten müssen meist durch EntwicklerInnen angepasst und erweitert werden. Es handelt sich hierbei also vorwiegend um abstrakte Basisklassen oder Interfaces, die ausimplementiert werden müssen.

Schließlich gibt Sommerville [2, pp. 427-428] einen umfassenden Überblick über die Vorteile und auch die Gefahren, welche mit der Verwendung von Frameworks einhergehen. Eine Auswahl daraus stellt folgende Auflistung dar:

- + Die wiederverwendeten Teile einer Software wurden oft bereits in zahlreichen Einsätzen erprobt und getestet, was eine **erhöhte Zuverlässigkeit** zur Folge hat.
- + Da die Kosten existierender Software bereits bekannt, während jene für die Entwicklung neuer Software meist schwer abzuschätzen sind, trägt die Verwendung wiederverwendbarer Komponenten im Rahmen von Frameworks zu einer **Minderung des Prozessrisikos** bei.
- + **Spezialisten** können ihr **Expertenwissen** in wiederverwendbaren Komponenten verpacken, so dass das Rad nicht jedes Mal neu erfunden werden muss.
- + Durch die Wiederverwendung von Software kann der für Entwicklung und Test benötigte **Aufwand** in einem Softwareprojekt **reduziert** und das Produkt **früher auf den Markt** gebracht werden.
- + Durch die **Einführung von Standards** (etwa in der Benutzeroberfläche) im Rahmen von wiederverwendbaren Komponenten wird die Benutzbarkeit von Software gesteigert, da BenutzerInnen im Umgang mit bekannten Schnittstellen weniger Fehler machen.

- Das **Auffinden, Verstehen und Anpassen** wiederverwendbarer Komponenten ist oft mit einem nicht zu unterschätzenden **Aufwand** verbunden.
- Die **Erstellung, Wartung und Nutzung** wiederverwendbarer Komponenten-Bibliotheken muss **konsequent** erfolgen. Der Entwicklungsprozess muss angepasst werden, um die Benutzung der Bibliothek zu gewährleisten.
- Ist der **Source Code** einer wiederverwendeten Komponente **nicht zugänglich**, kann sich dies in **erhöhten Wartungsausgaben** niederschlagen, wenn die Komponente nicht an geänderte Systemanforderungen angepasst werden kann.
- Viele EntwicklerInnen misstrauen Third-Party Software und bevorzugen, diese selbst zu schreiben. Sie leiden unter dem sogenannten „**Not-invented-here**“ Syndrom.

2.2.9.2 Inversion of Control

Wie bereits erwähnt können Frameworks erweitert werden, um den Bedürfnissen einer bestimmten Anwendung zu entsprechen. Diese Erweiterung nimmt jedoch für gewöhnlich keine Änderungen am Framework Code selbst vor. Stattdessen werden konkrete Klassen geschrieben, die von abstrakten Klassen des Frameworks erben oder es werden Interfaces, die das Framework vorgibt, ausimplementiert. Stets wird jedoch der grundsätzliche Aufbau und Ablauf eines Programms durch das Framework vorgegeben, weshalb man diese Vorgehensweise als „Inversion of Control“ bezeichnet. Ein besonders gutes Beispiel hierfür sind Callbacks. Diese erlauben EntwicklerInnen, an vom Framework vorgesehenen Stellen auf Events zu reagieren. [2, p. 433]

2.2.9.3 Designkonzepte rund um Abstraktion und Separation of Concerns

Laut Roger S. Pressman und Bruce R. Maxim [6, p. 232] stellt Abstraktion eine der grundlegendsten Herangehensweisen dar, wie Menschen mit Komplexität umgehen, und wird daher auch in der Softwareentwicklung eingesetzt. Sie erlaubt, auf mehreren Ebenen nach der Lösung eines Problems zu suchen. Auf der höchsten Abstraktionsebene wird das Problem in der Sprache der Problemdomäne, also seiner Umgebung, beschrieben. Auf der untersten Abstraktionsebene hingegen ist die Formulierung so gewählt, dass sie direkt in der Implementierung umgesetzt werden kann. Dazwischen gibt es meist Abstufungen, die in Terminologie und Detailgrad zwischen Domäne und Implementierung vermischen.

Auf allen Ebenen der Abstraktion kann zwischen prozeduraler Abstraktion und der Abstraktion von Daten unterschieden werden. Prozedurale Abstraktion bezieht sich auf eine Abfolge von Anweisungen, die eine bestimmte, begrenzte Funktion erfüllen. Ein Beispiel hierfür ist etwa die Formulierung „Login als Admin User“, welche die einzelnen Schritte „Öffnen der Login Seite“, „Ausfüllen des Admin Usernamens“, „Ausfüllen des Admin Passworts“ und „Klick auf Anmelden Button“ umfasst. Die Abstraktion von Daten manifestiert sich in der Abbildung einer Ansammlung von Daten in Datenobjekten. Beispielsweise stellen „Anmeldedaten“ ein Datenobjekt dar, welches in den meisten Fällen die Attribute „Benutzername“ und „Passwort“ umfasst.

Als weiteres Designkonzept beschreiben Pressman und Maxim [6, p. 234] das Prinzip „Separation of Concerns“. Dieses folgt dem Phänomen, dass die Summe eines Problems von den meisten Menschen als viel komplexer wahrgenommen wird, als wenn dieses in Teilbereiche zerlegt wird. Die Zersplitterung eines Problems in kleine, besser beherrschbare Teile, resultiert in einem verringerten Aufwand für dessen Lösung.

Die häufigste Erscheinungsform von „Separation of Concerns“ ist Modularität. Dabei wird Software zunächst in sogenannte Module zerlegt, die jeweils auf die Lösung eines bestimmten Teilbereichs des gesamten Problems spezialisiert sind. Eine spätere vollständige Integration aller Module zum Gesamtsystem bildet wieder das vollständige Problem ab. Die Verständlichkeit des Systems wird dadurch drastisch erhöht.

Ein wichtiger Aspekt von Modularität ist „Information Hiding“. Dahinter steckt die Absicht, die Details von Datenstrukturen und prozeduralen Vorgängen hinter dem Interface eines Moduls zu verstecken. Das bedeutet, die Außenwelt benötigt keine genauen Kenntnisse der Vorgänge im Inneren eines Moduls. Für sie ist nur die Schnittstelle des Moduls von Relevanz. Über diese wohldefinierten Schnittstellen wird sichergestellt, dass Module nur jene Informationen austauschen, die für die Funktionsweise der Software als Ganzes erforderlich sind. Dadurch wird das Risiko minimiert, dass sich unbeabsichtigte, im Zuge von Änderungen an einem Modul eingeführte Fehler in andere Bereiche der Software ausbreiten. [6, p. 235]

2.2.9.4 Funktionale Unabhängigkeit

Eine direkte Folge der Umsetzung der Designkonzepte Abstraktion, Modularität, „Seperation of Concerns“ und „Information Hiding“ ist die funktionale Unabhängigkeit. Diese ist gegeben, wenn jedes Modul die Erfüllung eines spezifischen Teilbereichs der Anforderungen adressiert und aus Sicht der anderen Teile eines Programms einfache Schnittstellen zur Interaktion anbietet. In Summe erhöhen funktional unabhängige Module die Wiederverwendbarkeit und erleichtern die Wartbarkeit, indem sie Seiteneffekte von Design- oder Codeänderungen in anderen Modulen weitestgehend unterbinden. [6, p. 236]

Die Unabhängigkeit der Module kann Pressman und Maxim [6, pp. 236-237] zufolge durch die Qualitätskriterien Kohäsion und Kopplung bestimmt werden.

Kohäsion beschreibt den Grad der Ausrichtung eines Moduls auf eine einzelne, wohl abgegrenzte Aufgabe. Sie hängt stark mit dem Konzept „Information Hiding“ zusammen, denn die Kohäsion ist dann hoch, wenn ein Modul wenig Interaktion mit Komponenten in anderen Bereichen des Programms erfordert, um seine Aufgabe zu erfüllen.

Kopplung hingegen ist ein Maß für den Grad der Verknüpfung eines Moduls zu anderen Modulen. Module mit hoher Kopplung besitzen vielen Verknüpfungen zu anderen Modulen und sind daher stark von anderen Modulen abhängig, während Module mit geringer Kopplung nur mit wenigen anderen Modulen interagieren.

Insgesamt wird immer eine hohe Kohäsion und eine niedrige Kopplung angestrebt. Jedoch gibt es hierbei Grenzen der Sinnhaftigkeit. So müssen in objektorientierter Programmierung Module zusammenarbeiten, um die gesamte Funktionalität eines Systems zu gewährleisten. Ein Modul ohne Verbindungen zur Außenwelt ist daher fragwürdig. Zudem können Module auch durchaus mehrere Funktionen in sich vereinen, wenn diese auf dasselbe Ziel hinarbeiten. Allerdings sollten „schizophrene“ Module vermieden werden, die versuchen, mehrere Aufgaben in sich zu vereinen. Eine „TestResults“ Klasse etwa kann eine Vielzahl an Funktionen beinhalten, die die Ergebnisse eines Tests verwalten. Ihr Ziel könnte lauten: „Verwaltung von Testergebnissen“. Allerdings sollte sie nicht gleichzeitig für die Darstellung der Resultate verantwortlich sein. Hier empfiehlt sich der Einsatz einer zusätzlichen „Reporter“ Klasse, die die Testergebnisse aufbereitet und darstellt.

Pressman und Maxim [6, pp. 296-298] unterscheiden jeweils drei Arten von Kohäsion und Kopplung:

- **Layer Cohesion** besagt, dass hohe Schichten auf die Services unterer Schichten zugreifen, jedoch untere Schichten keinen Zugriff auf höhere Schichten haben.
- **Functional Cohesion** bezieht sich vorwiegend auf Operationen: Sie tritt auf, wenn Module nur eine einzige Berechnung ausführen und dann das Resultat zurückgeben.
- **Communicational Cohesion** besagt, dass alle Operationen, die auf dieselben Daten zugreifen, innerhalb einer Klasse definiert sind. Im Allgemeinen konzentrieren sich solche „Datenhalter-Klassen“ meist einzig auf den Zugriff und das Speichern der verwalteten Daten.
- **Content Coupling** verstößt gegen das Konzept „Information Hiding“, indem eine Klasse interne Daten einer anderen Klasse modifiziert.
- **Control Coupling** tritt dann auf, wenn Operation A() Operation B() aufruft und dabei ein Control Flag an B übergibt, welches den Kontrollfluss innerhalb von B „umleitet“. Das Problem dieser Form von Kopplung ist, dass nicht direkt zusammenhängende Änderungen in B dazu führen können, dass auch die Bedeutung des von A übergebenen Control Flags angepasst werden muss.
- **External Coupling** beschreibt die Zusammenarbeit einer Klasse mit Infrastrukturkomponenten wie etwa Betriebssystemfunktionen und Datenbank-basierten Modulen. Auch wenn diese Art der Kopplung notwendig ist, sollte sie auf eine kleine Zahl an Klassen innerhalb des Systems beschränkt werden.

2.2.9.5 Die SOLID Prinzipien

Neben der Einhaltung der im vorigen Unterkapitel besprochenen Design-Konzepte kann auch die Befolgung der sogenannten „SOLID“ Prinzipien einer hohen Kohäsion und einer niedrigen Kopplung der Module zuträglich sein. „SOLID“ steht für Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle und Dependency Inversion Principles.

Das Single Responsibility Principle besagt, dass ein Modul nur eine Aufgabe haben sollte, und, dass es nur einen Grund geben sollte, dieses zu ändern (siehe [32]). Es entspricht somit fast wortwörtlich der Definition von Kohäsion. Dies ist vermutlich der Grund, warum Pressman und Maxim dieses Prinzip als einziges nicht mehr näher behandeln. Um auf das „TestResults“ und „Reporter“ Beispiel von vorher zurückzugreifen, wäre dieses Prinzip verletzt, wenn die „TestResults“ Klasse sowohl bei einer Änderung der Struktur der Testergebnisse als auch bei einer Änderung der Formatierung des Reports angepasst werden müsste, weshalb die Erstellung des Reports in die „Reporter“ Klasse ausgelagert wurde.

Die restlichen SOLID Prinzipien beschreiben Pressman und Maxim [6, pp. 242, 292-294] hingegen genauer:

Das Open-Closed Principle wurde im Grunde bereits durch die Forderung nach „Inversion of Control“ gestreift. Es sagt aus, dass eine Komponente so spezifiziert werden sollte, dass Erweiterungen von dieser möglich sind, ohne interne Modifikationen an der Komponente selbst vornehmen zu müssen. Hierzu werden Abstraktionen als „Puffer“ zwischen der Funktionalität, die wahrscheinlich erweitert wird, und der Design Klasse selbst, eingesetzt.

Das Liskov-Substitution Principle schreibt vor, dass Komponenten, welche eine Basisklasse verwenden, weiterhin ordnungsgemäß funktionieren, wenn die Basisklasse gegen eine von ihr abgeleitete Subklasse ausgetauscht wird.

Das Interface Segregation Principle warnt davor, ein allumfassendes Interface einzusetzen, welches eine Vielzahl an Aufgabenbereichen beschreibt. Die Gefahr besteht in diesem Fall darin, dass viele Klassen, die dieses Interface implementieren, dazu gezwungen werden, Funktionalität umzusetzen, die sie eventuell gar nicht benötigen. Anstelle eines großen, allumfassenden Interfaces sollen daher kleine, auf ganz spezifische Aufgabenbereiche zugeschnittene Schnittstellen definiert werden. Eine Klasse kann eine oder mehrere dieser Schnittstellen implementieren, so dass die umgesetzte Funktionalität der Klasse möglichst genau ihrer Aufgabe entspricht.

Das letzte der SOLID Prinzipien, das Dependency Inversion Principle, bezieht sich darauf, wie Klassen einander verwenden. Es legt fest, dass high-level Module niemals direkt von low-level Modulen, sondern beide von Abstraktionen abhängen sollten. Abstraktionen sollten zudem nicht von Details, sondern Details sollten von Abstraktionen abhängen. Dies klingt in der Theorie relativ kompliziert, doch anhand folgenden Beispiels sollte das Prinzip wesentlich einfacher zu verstehen sein:

Angenommen, ein Programm besteht aus „Worker“ und „Manager“ Klassen. Die „Manager“ Klassen enthalten komplexe Logik, die die Geschäftsprozesse einer Firma widerspiegeln, und stellt somit ein high-level Modul dar. Die „Worker“ Klasse hingegen ist für die Durchführung eher einfacherer Aufgaben bestimmt, also ein low-level Modul. Bei Nichtbeachtung des Dependency Inversion Principles verwendet die „Manager“ Klasse die „Worker“ Klasse direkt – sie speichert eine „Worker“ Instanz unter dem Typ der „Worker“ Klasse selbst ab, wie Abbildung 13 veranschaulicht:

```

3  class Worker {
4      public void work() {
5          // ....working
6      }
7  }
8
9  class Manager {
10     Worker worker;
11
12     public void setWorker(Worker w) {
13         worker = w;
14     }
15
16     public void manage() {
17         worker.work();
18     }
19 }
20
21 class SuperWorker {
22     public void work() {
23         //.... working much more
24     }
25 }

```

Abbildung 13: Nichtbeachtung von Dependency Inversion Principle (Quelle: [33])

Angenommen der besagten Firma gelingt es, ein paar besonders motivierte Arbeiter, die eine wesentlich höhere Arbeitsleistung erbringen, einzustellen – diese werden durch die „SuperWorker“ Klasse repräsentiert. Anstelle der normalen „Worker“ Klasse sollen nun die besonders fleißigen „SuperWorker“ für die Erledigung der Arbeit eingesetzt werden. Im Code, der in Abbildung 13 dargestellt ist, müsste in diesem Fall die Manager Klasse umgeschrieben werden, so dass sie anstelle von Instanzen des Typs „Worker“ Instanzen des Typs „SuperWorker“ verwendet. Dies kann schnell zu einem Problem werden, wenn die Belegschaft der Firma eine hohe Fluktuation aufweist, da „SuperWorker“ am Arbeitsplatz sehr begehrt sind und von der Konkurrenz abgeworben werden. Es wäre äußerst ineffizient, bei jedem Abgang eines „SuperWorkers“ die „Manager“ Klasse wieder auf die Verwendung eines normalen „Workers“ umzuschreiben und bei jedem Zugang eines „SuperWorkers“ den Code wieder auf diese anzupassen.

Das Dependency Inversion Principles beseitigt dieses Problem, indem die „Manager“ Klasse keine konkreten low-level Typen mehr festlegt, sondern stattdessen die Einhaltung eines abstrakten Vertrages, an den sowohl „Worker“ als auch „SuperWorker“ gebunden sind, einfordert. Dieser Vertrag wird durch das Interface „IWorker“ beschrieben. Auf diese Weise können normale „Worker“ und die besonders fleißigen „SuperWorker“ sogar zur Laufzeit des Programms ausgetauscht werden, und es sind keine Änderungen an der „Manager“ Klasse mehr notwendig, wie Abbildung 14 verdeutlicht:

```

1  interface IWorker {
2      | public void work();
3  }
4
5  class Worker implements IWorker{
6      | public void work() {
7      |     // ....working
8      | }
9  }
10
11 class SuperWorker implements IWorker{
12     | public void work() {
13     |     //.... working much more
14     | }
15 }
16
17 class Manager {
18     | IWorker worker;
19
20     | public void setWorker(IWorker w) {
21     |     worker = w;
22     | }
23
24     | public void manage() {
25     |     worker.work();
26     | }
27 }

```

Abbildung 14: Einhaltung des Dependency Inversion Principles (Quelle: [33])

Die Befolgung des Dependency Inversion Principles führt zu einer Reduktion der Kopplung, denn die Klasse „Manager“ ist mit den Klassen „Worker“ und „SuperWorker“ nicht mehr direkt verknüpft und eine Änderung des eingesetzten Arbeiter-Typs bedarf keiner Anpassung der „Manager“ Klasse mehr.

2.2.9.6 Anforderungen im Bereich Framework Design

ID	Name	Quellen
FD1	Förderung von Wiederverwendbarkeit	[2, pp. 194, 427-428, 431-434]
Beschreibung Das Testframework soll die Wiederverwendbarkeit auf Abstraktions-, Objekt- und Komponentenebene fördern, um die Zuverlässigkeit der im Rahmen des Testens entwickelten Artefakte zu erhöhen, die Benutzbarkeit durch die Einführung von Standards zu steigern und den Aufwand für die Entwicklung der Tests zu senken.		

ID	Name	Quellen
FD2	Anwendung von Entwurfsmustern	[2, p. 194]
Beschreibung Entwurfsmuster bieten bewährte Lösungen für bekannte Designprobleme in bestimmten Kontexten an. Die Anwendung von Entwurfsmustern ist eigentlich eine Unteranforderung der Anforderungen FD1 „Förderung von Wiederverwendbarkeit“ sowie FD4 „Abstraktion und Separation of Concerns“: Sie stellt eine Wiederverwendung auf Abstraktionsebene dar. Da Entwurfsmuster in der Softwareentwicklung einen hohen Stellenwert besitzen, werden sie jedoch als eigene Anforderung an das Testframework formuliert.		

ID	Name	Quellen
FD3	Inversion of Control	[2, p. 433]
Beschreibung Im Sinne von „Inversion of Control“ sind die Framework Objekte, und nicht die applikationsspezifischen Objekte, für die Kontrolle des Systems verantwortlich. Dafür muss gewährleistet werden, dass das Testframework erweitert werden kann, ohne dass Framework Code umgeschrieben werden muss. Möglichkeiten hierfür sind etwa die Ableitung einer applikationsspezifischen Klasse von einer abstrakten Framework Klasse oder Callbacks, die das Framework EntwicklerInnen zur Verfügung stellt, um auf vom Framework verwaltete Events zu reagieren.		

ID	Name	Quellen
FD4	Abstraktion und Separation of Concerns	[6, pp. 232, 234-235]
Beschreibung Abstraktion und „Separation of Concerns“ sind Designkonzepte, die die Reduktion der Komplexität von Software zum Ziel und eine erhöhte funktionale Unabhängigkeit eines Moduls zur Folge haben. Im Rahmen dieser Anforderung sollen diese beiden, und darüber hinaus auch die Konzepte der Modularität und des „Information Hiding“, vom Framework eingesetzt werden. Die Abstraktion ist dabei unter prozeduralen und datenbezogenen Aspekten zu betrachten.		

ID	Name	Quellen
FD5	Funktionale Unabhängigkeit	[6, pp. 236-242, 292-298]
<p>Beschreibung</p> <p>Funktionale Unabhängigkeit erhöht die Wiederverwendbarkeit und erleichtert die Wartbarkeit von Software, da sie die Anzahl abhängiger Module, die bei Änderungen an einem bestimmten Modul mitangepasst werden müssen, geringhält. Sie wird durch die Qualitätskriterien Kohäsion und Kopplung bestimmt, wobei immer ein möglichst hohes Maß an Kohäsion und ein möglichst niedriges Maß an Kopplung angestrebt werden.</p> <p>Um die Gewährleistung funktionaler Unabhängigkeit im Rahmen des Testframeworks zu evaluieren, soll auch die Einhaltung der SOLID Prinzipien (Single Responsibility, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle und Dependency Inversion Principle) überprüft werden.</p>		

2.2.10 Softwarequalität gemäß ISO/IEC 25010:2011

Während sich die vorhergehenden Kapitel der Literatur- und Anforderungsrecherche zum größten Teil mit den funktionalen Anforderungen des Testframeworks auseinandergesetzt haben, widmet sich dieses letzte Kapitel vorwiegend den nicht-funktionalen Anforderungen. Als Basis dient hierfür die Norm ISO/IEC 25010:2011 [34], welche die Qualitätseigenschaften eines Softwareprodukts in die Kategorien Funktionalität, Effizienz, Kompatibilität, Benutzbarkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit und Portabilität unterteilt.

Diese acht Qualitätskriterien nehmen in der späteren Evaluierung des Testframeworks die wohl bedeutendste Rolle ein, denn so gut wie jede der bereits beschriebenen funktionalen Anforderungen hat neben der reinen Ermöglichung einer bestimmten Funktionalität auch eine damit einhergehende Verbesserung eines oder mehrerer nicht-funktionaler Qualitätskriterien zum Ziel.

Im Folgenden werden die Hauptkategorien der Norm ISO/IEC 25010:2011 als einzelne Anforderungen aufgelistet und in der jeweiligen Beschreibung durch die in der Norm definierten Unterkategorien näher erläutert. Es handelt sich hierbei um eine freie Übersetzung, da die Norm ISO/IEC 25010:2011 zum Zeitpunkt der Verfassung dieser Arbeit nicht in deutscher Sprache verfügbar war. Zu jeder Kategorie ist zu beurteilen, in welchem Ausmaß eine betrachtete Software diese erfüllt.

ID	Name	Quellen
SQ1	Funktionalität	[34]
Beschreibung Die Funktionalität beschreibt das Ausmaß, in welchem ein Produkt oder System Funktionen anbietet, um die gestellten oder implizierten Anforderungen unter bestimmten Bedingungen zu erfüllen. <ul style="list-style-type: none">• Vollständigkeit: Es werden alle spezifizierten Aufgaben und Ziele erfüllt.• Korrektheit: Die korrekten Ergebnisse werden mit der benötigten Genauigkeit erzielt.• Angemessenheit: Die Funktionen des Systems begünstigen die Erreichung der spezifizierten Aufgaben und Ziele.		

ID	Name	Quellen
SQ2	Effizienz	[34]
Beschreibung Die Effizienz beschreibt die Leistung des Systems im Verhältnis zu den unter gegebenen Bedingungen aufgewendeten Ressourcen.		

- **Zeitverhalten:** Die Antwort- und Verarbeitungszeiten, sowie die Durchsatzrate des Systems bei der Erfüllung seiner Funktion, erfüllen die Anforderungen.
- **Ressourcenverbrauch:** Die Art und Menge der vom System bei der Erfüllung seiner Funktion verwendeten Ressourcen entspricht den Anforderungen.
- **Kapazität:** Die maximalen Grenzen eines Systemparameters entsprechen den Anforderungen.

ID	Name	Quellen
SQ3	Kompatibilität	[34]
<p>Beschreibung</p> <p>Die Kompatibilität beschreibt das Ausmaß, in welchem ein System, ein Produkt oder eine Komponente den Informationsaustausch oder die Erfüllung seiner Funktionen im Zusammenspiel mit anderen Systemen, Produkten oder Komponenten innerhalb einer geteilten Hardware oder Software Umgebung bewerkstelligen kann.</p> <ul style="list-style-type: none"> • Ko-Existenz: Das System/Produkt hat keine schädlichen Auswirkungen auf andere Produkte, mit denen es sich eine Umgebung und Ressourcen teilt. • Interoperabilität: Zwei oder mehr Systeme sind in der Lage, Informationen auszutauschen und diese Informationen weiterzuverwenden. 		

ID	Name	Quellen
SQ4	Benutzbarkeit	[34]
<p>Beschreibung</p> <p>Die Benutzbarkeit stellt das Ausmaß dar, in welchem ein System/Produkt von bestimmten Benutzern verwendet werden kann, um in einem bestimmten Kontext gegebene Ziele effektiv, effizient und zufriedenstellend zu erreichen.</p> <ul style="list-style-type: none"> • Ersichtlichkeit der Angemessenheit: BenutzerInnen erkennen, ob ein System/Produkt zur Erfüllung ihrer Bedürfnisse angemessen ist. • Erlernbarkeit: Die Bedienung des Produkts, um bestimmte Ziele zu erreichen, kann effektiv, effizient und risikofrei erlernt werden. • Bedienbarkeit: Das System/Produkt besitzt Eigenschaften, welche es leicht bedienbar machen. • Schutz des Benutzers vor Fehlern: BenutzerInnen werden davor geschützt, Fehler zu begehen. • Ästhetik: Die Benutzeroberfläche ermöglicht BenutzerInnen eine ansprechende und zufriedenstellende Interaktion. • Barrierefreiheit: Das System/Produkt kann von Menschen mit verschiedensten Fähigkeiten und Eigenschaften zur Erreichung gegebener Ziele in einem bestimmten Kontext verwendet werden. 		

ID	Name	Quellen
SQ5	Zuverlässigkeit	[34]
Beschreibung Die Zuverlässigkeit wird durch das Ausmaß bestimmt, in welchem ein System, ein Produkt oder eine Komponente gegebene Funktionen unter gegebenen Bedingungen über einen bestimmten Zeitraum erfüllt. <ul style="list-style-type: none"> • Reife: Das System/Produkt erfüllt die Anforderungen für Zuverlässigkeit unter Normalbetrieb. • Verfügbarkeit: Das System/Produkt ist einsatzbereit und zugänglich, wenn es zur Verwendung kommen soll. • Fehlertoleranz: Das System/Produkt arbeitet wie vorgesehen, obwohl in der Hardware oder Software Fehler bestehen. • Wiederherstellbarkeit: Das System/Produkt kann bei Eintreten einer Unterbrechung oder Störung die betroffenen Daten wiederherstellen und den gewünschten Zustand wiedereinrichten. 		

ID	Name	Quellen
SQ6	Sicherheit	[34]
Beschreibung Sicherheit bestimmt das Ausmaß, in welchem ein Produkt oder System Daten und Informationen schützt, so dass andere Produkte, Systeme oder Personen auf diese nur jenen Grad an Zugriff erlangen, der ihrem Typ oder ihrer Berechtigung angemessen ist. <ul style="list-style-type: none"> • Vertraulichkeit: Das System/Produkt stellt sicher, dass der Zugriff auf Daten nur durch Berechtigte erfolgen kann. • Integrität: Das System, das Produkt oder die Komponente verhindert unberechtigten Zugriff auf, oder die Veränderung von Computerprogrammen und Daten. • Nachweisbarkeit: Es kann bewiesen werden, dass Handlungen oder Ereignisse stattgefunden haben, so dass diese später nicht geleugnet werden können. • Zurechenbarkeit: Die Handlungen einer Entität können dieser eindeutig zugerechnet werden. • Authentizität: Die Identität eines Subjekts oder einer Ressource kann nachgewiesen werden. 		

ID	Name	Quellen
SQ7	Wartbarkeit	[34]
<p>Beschreibung</p> <p>Als Wartbarkeit wird das Ausmaß an Effektivität und Effizienz bezeichnet, in welchem Änderungen am Produkt oder System zu dessen Verbesserung, Korrektur oder Anpassung an eine geänderte Umgebung vorgenommen werden können.</p> <ul style="list-style-type: none"> • Modularität: Das Programm/System besteht aus eigenständigen Komponenten, so dass Änderungen an einer Komponente nur minimale Auswirkungen auf andere Komponenten nach sich ziehen. • Wiederverwendbarkeit: Ein Element des Programms/Systems kann in mehr als einem System, oder im Aufbau eines anderen Elements, verwendet werden. • Analysierbarkeit: Die Auswirkungen einer beabsichtigten Änderung eines Produkts/Systems auf einen oder mehrere seiner Bestandteile können abgeschätzt werden, das Produkt/System kann auf Mängel oder Fehlerquellen hin untersucht werden und zu ändernde Bestandteile können identifiziert werden. • Änderbarkeit: Das Produkt/System kann verändert werden, ohne dabei Fehler einzubringen oder die bestehende Produktqualität zu verschlechtern. • Testbarkeit: Testkriterien für ein System, ein Produkt oder eine Komponente können effektiv und effizient festgelegt und deren Einhaltung überprüft werden. 		

ID	Name	Quellen
SQ8	Portabilität	[34]
<p>Beschreibung</p> <p>Portabilität bestimmt das Ausmaß der Effektivität und Effizienz, mit welchem ein System, ein Produkt oder eine Komponente von einer Hardware, Software oder Umgebung in eine andere übertragen werden können.</p> <ul style="list-style-type: none"> • Anpassbarkeit: Ein Produkt/System kann effektiv und effizient an eine andersartige oder weiterentwickelte Hardware, Software oder Umgebung angepasst werden. • Installierbarkeit: Ein Produkt/System kann effektiv und effizient in einer bestimmten Umgebung installiert und deinstalliert werden. • Austauschbarkeit: Ein Produkt kann ein anderes Software-Produkt, welches dieselbe Aufgabe erfüllt, in derselben Umgebung ersetzen. 		

2.3 Framework-Entwurf

Dieses Kapitel soll den LeserInnen dieser Arbeit helfen, sich einen Überblick über den Entwurf des Testframeworks zu verschaffen. Es werden zunächst einige grundlegende Designentscheidungen behandelt und im Anschluss die Umgebung des Systems und seine technischen Bestandteile betrachtet. Zum Schluss wird anhand der „internen“ Framework-Architektur gezeigt, wie die einzelnen Bausteine des Frameworks miteinander interagieren. Eine genaue Beschreibung der Framework-Bausteine erfolgt hingegen erst im Folgekapitel.

2.3.1 Designentscheidungen und technische Bestandteile

2.3.1.1 Auswahl von Ökosystem und Tools

Die Auswahl der im Testframework zum Einsatz kommenden Werkzeuge wurde durch zwei wesentliche Faktoren bestimmt: Das Framework sollte einerseits als Open-Source Lösung verfügbar sein und daher selbst auf Open-Source Werkzeugen basieren, und musste andererseits in der Lage sein, funktionale Systemtests im Browser zu automatisieren. Während es inzwischen einige Werkzeuge gibt, welche automatisierte funktionale Systemtests in Webbrowsern ermöglichen (siehe Kapitel 1.2.3), sind die meisten davon kostenpflichtig. Die Wahl fiel damit auf die wohl weitverbreitetste Open-Source Alternative: Selenium Webdriver.

Eine detaillierte Beschreibung von Selenium Webdriver findet sich in Kapitel 1.2.3. Um den Lesefluss jedoch nicht zu unterbrechen, folgt nun eine kurze Wiederholung dessen wichtigster Eigenschaften: Selenium Webdriver ermöglicht eine geskriptete Interaktion mit Webbrowsern über deren Accessibility-API und kann von zahlreichen Programmiersprachen angesprochen werden, insofern für diese Programmiersprache Bindings für Selenium Webdriver verfügbar sind.

Somit stellte sich im nächsten Schritt die Frage, mit welcher Programmiersprache Selenium Webdriver durch das Testframework angesteuert werden sollte. Es wurde entschieden, hierfür JavaScript in einer Node.js Umgebung einzusetzen.

Einer der Hauptgründe für diese Entscheidung war, dass Node.js mit seinem Package Manager npm zum Zeitpunkt der Implementierung des Testframeworks über das größte Ökosystem für Open-Source Softwarebibliotheken der Welt verfügte [26]. Die Chancen, qualitativ hochwertige und frei verfügbare Bibliotheken für das Testframework zu finden, wurden so optimiert. Darüber hinaus erweitert Node.js JavaScript um Funktionen, die sich bei der Implementierung des Testframeworks als nützlich erwiesen, etwa der Unterstützung von Lese- und Schreiboperationen im Filesystem oder der Möglichkeit, Kindprozesse zu erzeugen [35].

Ein positiver Nebeneffekt der Verwendung von Node.js ergab sich zudem dadurch, dass auch die durch das Framework getestete React-Webapplikation im Frontend auf JavaScript basiert. Somit konnten Ressourcen des Frontends wie beispielsweise Übersetzungen für die Systemtests wiederverwendet werden, und auch die Erlernbarkeit des Testframeworks wurde für die mit JavaScript bereits vertrauten Frontendentwickler erleichtert.

Nachdem Selenium als Schnittstelle zwischen Testskripten und Webbrowsern, sowie JavaScript und Node.js als Eckpfeiler des Technology Stacks fixiert waren, galt es schließlich, Tools zu finden, die bei der Implementierung des Testframeworks wiederverwendet werden konnten, um das Rad nicht neu erfinden zu müssen. Gesucht wurde nach Selenium Bindings für Node.js, einem Testrunner, einer Assertion Library sowie einem oder mehreren Reporter. Tabelle 2 präsentiert die ausgewählten Werkzeuge:

Werkzeugname	Quelle	Features
WebdriverIO	[20]	<ul style="list-style-type: none"> • Node.js Bindings für Selenium • Erweiterte Funktionalität der Selenium Commands • Synchroner Testrunner • Zusammenarbeit mit gängigen Test Driven und Behavior Driven Development Frameworks • Einfache Integration mit anderen Testtools (beispielsweise selenium-standalone Service)
Jasmine	[21]	<ul style="list-style-type: none"> • Behavior Driven Development Framework für JavaScript • Umfangreiche Assertion Library • In Browser und Server-Umgebung (Node.js) lauffähig
Allure	[22]	<ul style="list-style-type: none"> • Reporter für graphisch aufbereitete Darstellung von Testresultaten im Browser • Import von JUnit/XUnit Resultaten möglich • siehe Kapitel 2.4.8.3
wdio-spec-reporter	[36]	<ul style="list-style-type: none"> • Reporter für Darstellung von Testresultaten im aus Behavior Driven Development bekannten Spec-Format • Ausgabe in der Konsole • siehe Kapitel 2.4.8.2

Tabelle 2: Im Testframework eingesetzte Werkzeuge

2.3.1.2 Zielgruppe

Das entwickelte Testframework verzichtet auf eine Bedienung mittels Graphischer Benutzeroberfläche. Alle benötigten Artefakte müssen in Form von Code erstellt werden. Das Framework richtet sich daher gezielt an EntwicklerInnen und SpezialistInnen für Testautomatisierung, welche über Programmierkenntnisse verfügen.

2.3.1.3 Keine Unterstützung von Capture/Replay Tools

Spillner et al. [1, p. 212] berichten von sogenannten „Capture/Replay Tools“, welche in dem Fall, dass die Benutzeroberfläche als Testschnittstelle dient, eingesetzt werden können, um die Interaktionen zwischen TesterInnen und dem System aufzuzeichnen und später erneut abzuspielen. Diese Funktionalität wird durch das Testframework nicht unterstützt.

Dies liegt einerseits an der fehlenden graphischen Benutzeroberfläche des Frameworks, was den Umgang mit einem Capture/Replay Tool beträchtlich erschwert, und andererseits daran, dass Capture/Replay Tools weniger robust als manuell geschriebene Testskripte und teils sehr ineffizient sind.

So weisen Spillner et al. [1, p. 213] darauf hin, dass mit Capture/Replay Tools aufgezeichnete Skripte in so gut wie allen Fällen nachträglich bearbeitet werden müssen, um für Regressionstests geeignet zu sein.

Dustin et al. [7] warnen sogar ausdrücklich vor der Verwendung von Capture/Replay Tools. Diese seien im besten Fall ineffizient, und im schlechtesten Fall komplett nutzlos. Sie basieren nämlich auf statischen Werten (den Tastatureingaben), und sind daher sehr anfällig gegenüber Änderungen der Testdaten. Zudem unterstützen die durch Capture/Replay Tools generierten Skripte nicht die „Best Practices“ der Softwareentwicklung und sind ohne nachträgliche Bearbeitung nicht modular und generell schlecht wartbar.

Diese Einschätzung scheint ein Großteil der Testautomatisierungsbranche zu teilen: Gemäß der Studie „Softwaretest in Praxis und Forschung“ [17, p. 16] sank die Rate an Firmen, welche Capture/Replay Tools zur Testautomatisierung einsetzen, innerhalb der Jahre 2011-2017 von 68 Prozent auf 51 Prozent.

2.3.1.4 Typensicherheit

Wie bereits in Kapitel 2.2.1.3 erwähnt wurde, ist JavaScript keine typensichere Sprache. Während dieser Umstand durchaus auch Vorteile bietet (etwa die Möglichkeit, einem Objekt dynamisch neue Properties hinzuzufügen), ist eine statische Codeanalyse, um Fehler bereits vor der Ausführung eines Programms zu finden, dadurch nur eingeschränkt möglich. Darüber hinaus wird auch die Effizienz beim Arbeiten mit einer Entwicklungsumgebung wie Visual Studio Code eingeschränkt, da zahlreiche die Entwicklung unterstützende Funktionen (etwa das Vorschlagen aller Properties, die auf einem Objekt definiert sind) nicht genutzt werden können.

Daher wird im Testframework TypeScript eingesetzt, um JavaScript zu typisieren und das Beste aus den Welten von JavaScript und typsicheren Sprachen zu vereinen.

TypeScript [37] ist eine Übermenge von JavaScript, welche neben Typen auch die neuesten JavaScript Vorschläge ECMAScript 6 und sogar manche Features von ECMAScript 7 unterstützt. Im Zusammenspiel mit einer Entwicklungsumgebung wie Visual Studio Code [38], welche ebenso wie TypeScript von Microsoft entwickelt wurde und in sämtlichen TypeScript Demos von Microsoft zum Einsatz kommt, entfaltet TypeScript eine ganze Reihe an Vorteilen, darunter:

- Unterstützung der neuesten JavaScript Language Features
- Kompilierung in reines JavaScript – somit in jeder JavaScript Umgebung lauffähig
- Static Checking
 - Viele Fehler und Inkonsistenzen im Code werden bereits vor der Programmausführung aufgedeckt
 - Ermöglicht „intelligente Unterstützung“ durch die Entwicklungsumgebung, etwa durch Vorschlagen von Property Namen, Refactoring, „suche alle Verweise“, Autocompletion etc.
- Interfaces, typensichere Funktionen und Klassen
 - Die korrekte Benutzung einer wiederverwendbaren Softwarekomponente durch EntwicklerInnen kann mittels „Verträgen“ sichergestellt werden
 - TypeScript kompiliert nicht, wenn „Typen-Verträge“ nicht eingehalten werden

2.3.2 Systemgrenzen und Zusammenspiel der Bestandteile

Nachdem der Fokus im vorangehenden Kapitel auf den wesentlichen Eigenschaften der technischen Bestandteile des Testframeworks mit dem Namen „WDIO-WORKFLO“, sowie den Gründen, wieso genau diese für den Einsatz ausgewählt wurden, lag, verdeutlicht dieses Kapitel, wie die einzelnen technischen Bestandteile zusammenarbeiten und welche Grenzen sie dabei überwinden müssen. Zur Veranschaulichung demonstriert Abbildung 15 die Einbettung der einzelnen Bestandteile in das Gesamtsystem:

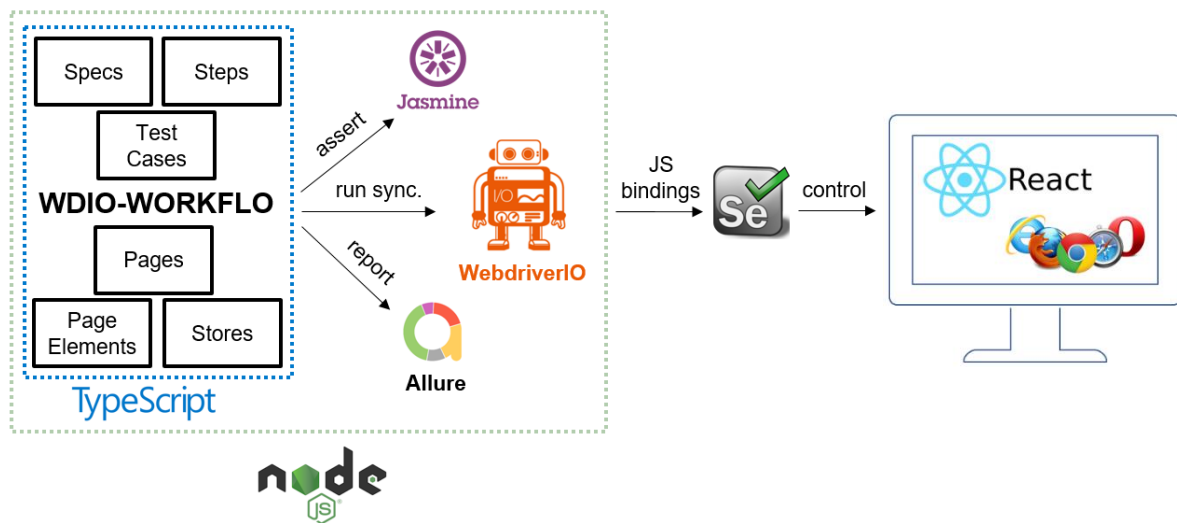


Abbildung 15: Skizze der Systemumgebung von "WDIO-WORKFLO"

2.3.2.1 Systemgrenzen

Bei Betrachtung von Abbildung 15 wird deutlich, dass in der Umgebung von WDIO-WORKFLO drei Systemgrenzen existieren, die in der Planung des Frameworks besonders berücksichtigt werden mussten:

So wurden alle internen Komponenten von WDIO-WORKFLO in TypeScript entwickelt. WDIO-WORKFLO und alle Werkzeuge, mit denen es direkt zusammenarbeitet, befinden sich darüber hinaus in einer Node.js Umgebung. Jedoch sind diese anderen Werkzeuge nicht in TypeScript, sondern in „reinem“ JavaScript geschrieben. Was theoretisch die erste Systemgrenze beschreibt und nach einem Problem klingt, stellt in der Praxis jedoch kaum ein Hindernis dar: Da TypeScript in JavaScript kompiliert wird, ist WDIO-WORKFLO mit seinen externen Abhängigkeiten kompatibel. Die fehlende Typisierung der externen Werkzeuge macht im Grunde nur in der Entwicklungsumgebung einen Unterschied, da diese somit keine Rückschlüsse auf den Aufbau der externen Werkzeuge durchführen kann. Jedoch gibt es auch hierfür Abhilfe: TypeScript erlaubt es, sogenannte „Type Definitions“ für JavaScript Bibliotheken zu erstellen, die in reinem JavaScript entwickelt wurden. Diese fügen nachträglich Typeninformationen hinzu, sodass Features wie Static Checking nun auch mit den externen Werkzeugen funktionieren.

Für Jasmine und WebdriverIO waren derartige Type Definitions bereits verfügbar und mussten nur heruntergeladen und installiert werden. Allure hingegen verfügte zum Zeitpunkt der Framework Implementierung noch nicht über Type Definitions. In diesem Fall ordnet TypeScript den Objekten ohne Type Definitions den universellen Typ „any“ zu. Diese Objekte werden von der Entwicklungsumgebung und dem Compiler wie ganz normale JavaScript Objekte behandelt und unterliegen keinen Type Checks beim Kompilieren.

Die zweite Systemgrenze liegt zwischen dem Testrunner WebdriverIO und Selenium. Da Selenium nicht in JavaScript implementiert ist, stellt WebdriverIO sogenannte „Bindings“ zur Verfügung, die eine Übersetzung der JavaScript Befehle in für Selenium verständliche Anweisungen durchführen.

Die letzte Systemgrenze bildet die Schnittstelle zwischen dem Selenium Server und dem Webbrowser, in welchem die getestete Applikation läuft. Diese ist besonders fehleranfällig, denn es existiert eine große Vielzahl an Webbrowsern, die Standards teilweise unterschiedlich oder gar nicht umsetzen. Hier wird für jeden Browser ein vom Browserhersteller entwickelter Treiber benötigt, der Tools wie Selenium einen geskripteten Zugriff auf Webanwendungen ermöglicht. Im Falle von Selenium findet dieser über die Befehle der Accessibility API des Browsers statt. Abhängig davon, wie gut die Browserhersteller den Treiber-Support umsetzen, kann auch die Zuverlässigkeit der Tests beeinträchtigt sein.

2.3.2.2 Zusammenspiel der Framework-Bestandteile

Für die Erklärung des Zusammenspiels zwischen WDIO-WORKFLO und seinen externen Bestandteilen wird der in TypeScript entwickelte Kern des Frameworks zunächst als Blackbox betrachtet. Deren Inhalt wird im nächsten Unterkapitel genauer beleuchtet. Zunächst jedoch werden die Rollen beschrieben, welche die einzelnen externen Bestandteile einnehmen, und genauer untersucht, wie diese zusammenwirken.

Nachdem die Testfälle und die zugehörigen Anforderungen, die diese validieren, im TypeScript Kern von WDIO-WORKFLO definiert wurden, gelangen diese durch WebdriverIO, oder genauer gesagt, eine auf die Bedürfnisse von WDIO-WORKFLO angepasste Version von WebdriverIO, zur Ausführung. WebdriverIO stellt dafür einen synchronen Testrunner zur Verfügung, was die Handhabung der an sich asynchronen Testmechaniken wesentlich vereinfacht und den Code besser lesbar macht, wie ein Vergleich von Abbildung 16 und Abbildung 17 verdeutlicht:

```

1  const title = browser.getTitle()
2
3  expect(title).toMatch('WebdriverIO')
4
5  browser.element('//input[@id="titleInput"]').setValue(title)
6  browser.element('//button[@id="titleSubmit"]').click()

```

Abbildung 16: WebdriverIO Codebeispiel mit synchronem Testrunner

```

1  browser.getTitle().then(title => {
2
3      expect(title).toMatch('WebdriverIO')
4
5      browser.element('//input[@id="titleInput"]').setValue(title).then(() => {
6          browser.element('//button[@id="titleSubmit"]').click()
7      })
8  })

```

Abbildung 17: WebdriverIO Codebeispiel mit asynchronem Testrunner

Während in Abbildung 16 jeder asynchrone Request durch den Testrunner automatisch synchronisiert wird, muss die Antwort des Selenium Servers in Abbildung 17 „manuell“ abgewartet und ein Callback definiert werden, das festlegt, wie mit dem Rückgabewert verfahren werden soll.

WebdriverIO (in Form der Instanz „browser“) übersetzt die in Abbildung 16 dargestellten JavaScript Befehle in für Selenium verständliche Anweisungen, die an den Selenium Server geschickt werden, welcher daraufhin über den jeweiligen Browser-Treiber mit der Webanwendung interagiert und das Resultat wieder an den WebdriverIO Client retourniert.

Daraufhin kann bestimmt werden, ob ein retournierter Wert einem erwarteten Wert entspricht oder von diesem abweicht. Hierzu werden sogenannte „Assertions“ eingesetzt, welche bei WDIO-WORKFLO von der Jasmine Library zur Verfügung gestellt werden. In Abbildung 16 wird solch eine Assertion in Zeile 3 angewendet, um zu prüfen, ob der Titel einer Webseite mit dem String „WebdriverIO“ übereinstimmt.

Die Ergebnisse dieser Vergleiche werden im Anschluss als Testresultate an einen oder mehrere Reporter übermittelt, die diese gemeinsam mit zusätzlichen Informationen (etwa dem Namen des Testfalls) in einer bestimmten Form ausgeben. In WDIO-WORKFLO stehen hierfür zwei Reporter, die unterschiedliche Ziele verfolgen, zur Verfügung:

Der Spec Reporter gibt die Ergebnisse in der Entwicklerkonsole aus und bietet somit optimale Unterstützung bei der Implementierung konkreter Testfälle. Der Allure Reporter hingegen bereitet die Testergebnisse, sowie alle durchgeführten Schritte und viele weitere Informationen im Browser für das Testmanagement graphisch ansprechend auf.

2.3.3 Architektur der Framework-Bausteine

Während im vorangehenden Kapitel das Zusammenspiel der externen Bestandteile von WDIO-WORKFLO erklärt und der „Core“ von WDIO-WORKFLO noch als Blackbox behandelt wurden, betrachtet dieses Kapitel nun die Rolle der einzelnen internen Komponenten von WDIO-WORKFLO sowie deren Beziehungen zueinander.

Ausgangspunkt für die Betrachtung stellt das auf der nächsten Seite in Abbildung 18 gezeigte Klassendiagramm dar. Dieses entspricht streng genommen nicht genau der tatsächlichen Implementierung, denn es soll hauptsächlich der Verständlichkeit der „inneren Architektur“ des Frameworks dienen und nicht als exakte technische Beschreibung. Eine technisch korrekte und detaillierte Darstellung der einzelnen Framework-Bausteine bietet dagegen das Kapitel 2.4.

Es wurde im Klassendiagramm zudem auf die Darstellung der Sichtbarkeit von Attributen und Operationen verzichtet. Als Grundregel gilt, dass mit Ausnahme eines einzigen Attributs (mehr dazu im weiteren Verlauf dieses Kapitels) sämtliche Attribute die Sichtbarkeit „protected“ besitzen und sämtliche dargestellte Operationen als „public“ definiert sind. Somit können die Werte sämtlicher Attribute nur über die von außen zugänglichen Operationen verändert werden. Dieser Umstand ist auch in der tatsächlichen Implementierung gegeben und trägt entscheidend zur Sicherstellung des Design Konzepts „Information Hiding“ bei.

Klasseninterne Operationen werden im vorliegenden Diagramm ausgespart, da sie dem Verständnis des Zusammenspiels der unterschiedlichen Klassen nicht zuträglich sind.

Die Klassen „Feature“, „Story“, „Suite“ und „Testcase“ sind zudem eigentlich gar keine Klassen, sondern Funktionen. In JavaScript macht dies jedoch rein technisch gesehen keinen Unterschied, denn streng genommen ist in JavaScript jede Klasse eine Funktion, wobei die „Inhalte“ der Klasse im Scope des Function Bodies definiert werden.

Die Reihenfolge der anschließenden Betrachtung der einzelnen Komponenten orientiert sich an dem zeitlichen Vorgehen bei der Entwicklung neuer automatisierter Tests. Sie beginnt links oben im Klassendiagramm und endet in dessen unterem Bereich:

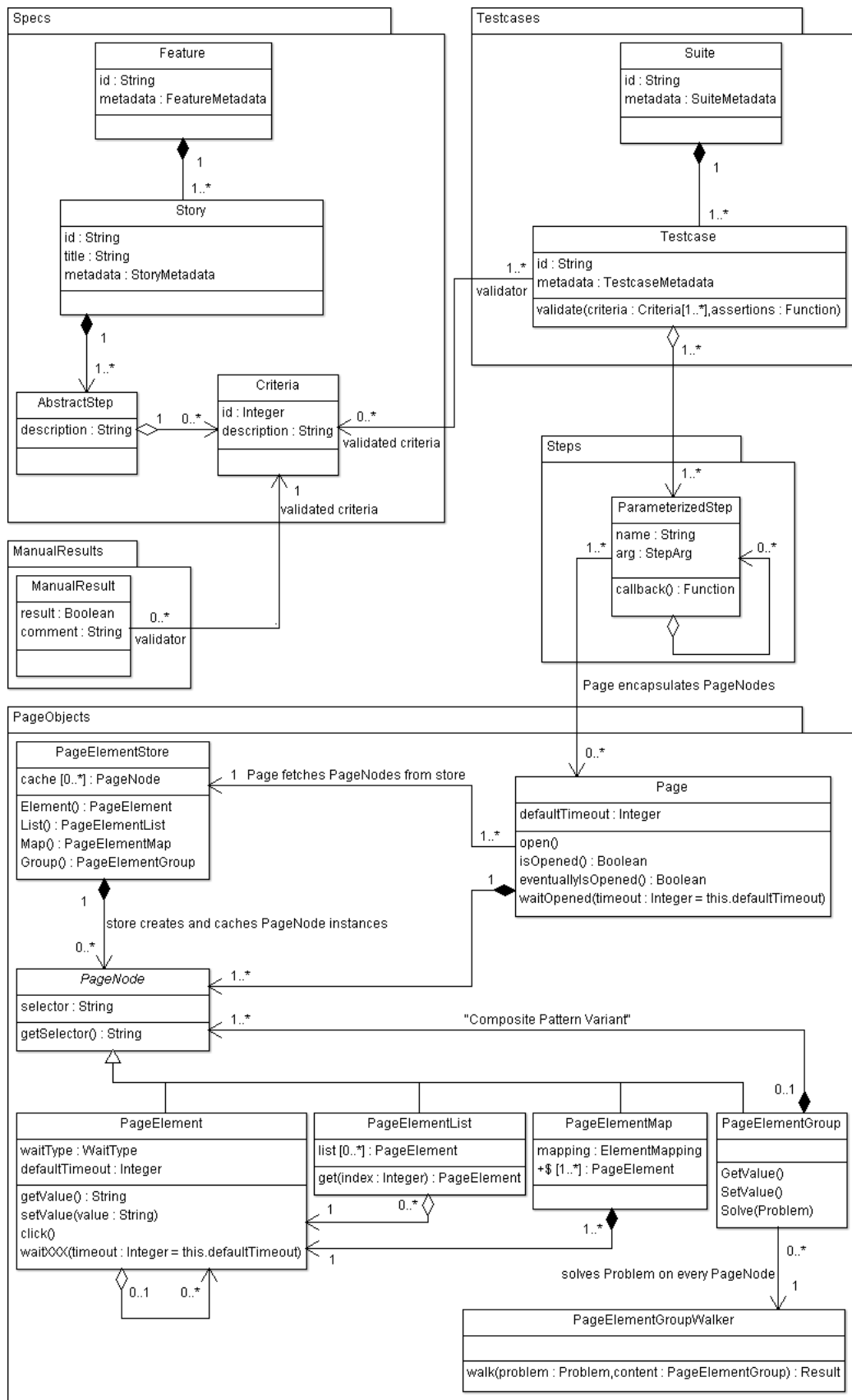


Abbildung 18: Klassendiagramm der internen Komponenten von WDIO-WORKFLO

Zunächst müssen die Anforderungen der Software in eine für das Framework testbare Form gebracht werden. Dafür werden die grundlegenden Bereiche oder Fähigkeiten des Systems in „Features“ zusammengefasst, welche in der Terminologie agiler Entwicklungsmethoden dem Issue-Typ „Epic“ entsprechen und im klassischen Wasserfallmodell etwa durch die Hauptkapitel des Lastenhefts beschrieben werden können.

Einem Feature werden (User) „Stories“ zugeordnet, die eine bestimmte Anforderung an das System abbilden. Im Rahmen von WDIO-WORKFLO wird jedoch nicht die eigentliche User Story formuliert, sondern eine funktionale Systembeschreibung auf Basis der User Story vorgenommen. In Form abstrakter Testschritte werden die einzelnen Interaktionen zwischen BenutzerInnen und System nachgestellt.

Jeder dieser „AbstractSteps“ kann Akzeptanzkriterien enthalten, die für eine erfolgreiche Validierung der Anforderung hinter der User Story erfüllt werden müssen. Ein Akzeptanzkriterium kann entweder manuell oder automatisiert überprüft werden.

Im Falle der automatisierten Validierung von Akzeptanzkriterien müssen nun Testfälle geschrieben werden, die diese vornehmen. Ein einzelnes Akzeptanzkriterium kann in einem oder auch in mehreren Testfällen validiert werden. Ein Testfall wiederum überprüft üblicherweise ein oder mehrere Akzeptanzkriterien. Jedoch ist es im Sinne einer Erleichterung der Testfallimplementierung auch möglich, Testfälle auszuführen, die (noch) keine Akzeptanzkriterien validieren. Für die Überprüfung des derzeitigen Systemzustands steht die Funktion „validate()“ zur Verfügung, welche eine Angabe der betroffenen Akzeptanzkriterien sowie der dafür benötigten (Jasmine) Assertions ermöglicht.

Jeder Testfall wird einer „Suite“ zugeordnet, die einer besseren Organisation der unterschiedlichen Testfälle dient, und setzt sich aus einem oder mehreren „ParameterizedSteps“ zusammen.

Ein „Step“ stellt eine wiederverwendbare, gekapselte Zustandsänderung des Systems dar. Er ist parametrisierbar, um etwa unterschiedliche Äquivalenzklassen des manipulierten Testobjekts zu behandeln. Für die Überprüfung des Systemzustands nach der Ausführung des Steps stellt dieser ein Callback zur Verfügung, in welchem die oben beschriebene validate() Funktion aufgerufen werden kann.

Steps können in unterschiedlichen Abstraktionsebenen definiert werden und sich daher aus mehreren anderen Steps zusammensetzen.

Um mit dem System zu interagieren, hat jeder Step Zugriff auf „PageObjects“ - einer Gruppe von Klassen, deren Aufgabe darin besteht, die Struktur und die Funktionsweise der einzelnen Seiten einer Webapplikation zu abstrahieren. PageObjects sind Martin Fowler [39] zufolge ein im Testen graphischer Oberflächen häufig eingesetztes Entwurfsmuster, denn sie kapseln Details im Aufbau und den inneren Abläufen eines Bausteins der Oberfläche hinter einer applikationsspezifischen Schnittstelle und trennen diese so von der Logik des Testfallablaufs.

In WDIO-WORKFLO stellt die Klasse „Page“ das „Tor in die Welt der PageObjects“ dar. Sie bildet die Zusammensetzung einer Webseite aus mehreren „PageNodes“ ab und stellt Funktionen zur Manipulationen und Abfrage des Zustands einer Webseite zur Verfügung. Steps können entweder die Schnittstellenfunktionen einer Page aufrufen, oder direkt mit den öffentlich sichtbaren Bausteinen einer Page interagieren, um den Zustand des Systems abzufragen oder zu verändern.

PageNode bildet den kleinsten gemeinsamen Nenner aller Bausteine einer Website ab: einen Selektor, der eine eindeutige Identifikation des Bausteins ermöglicht. Da PageNode selbst eine abstrakte Klasse ist, und daher nicht instanziiert werden kann, leiten die Klassen „PageElement“, „PageElementList“, „PageElementMap“ und „PageElementGroup“ von dieser ab.

PageElement repräsentiert einen einzelnen Webseitenbaustein und bietet Funktionen an, um mit diesem zu interagieren. Da in modernen Webapplikationen viele Bestandteile nicht initial (beim Neuladen einer Seite) zur Verfügung stehen, sondern oft erst asynchron nachgeladen werden, um den Eindruck eines schnelleren Seitenaufbaus zu erwecken, verfügt PageElement darüber hinaus über einen impliziten Wartemechanismus. Dieser erlaubt die Definition bestimmter Wartetypen (etwa „exist“, „visible“, „text“) bei Erstellung des PageElements und führt vor jeder Interaktion ein „implicitWait“ aus. Über gesonderte „waitXXX()“ Funktionen kann darüber hinaus auf viele verschiedene Ereignisse explizit gewartet werden.

Sammlungen mehrerer PageElement Bausteine desselben Typs können über die Klassen PageElementList und PageElementMap abgebildet werden.

PageElementList erlaubt es, PageElements dynamisch, also zur Laufzeit der Tests, aufgrund von Eigenschaften wie deren Beschriftung in der GUI oder deren Index zu identifizieren. Sie eignen sich für Szenarien, in denen der Inhalt der Listenelemente im Vorhinein nicht bekannt ist - etwa einem Newsfeed.

PageElementMap hingegen erlaubt die Identifikation von PageElements auf statische Weise - zum Beispiel über ein Mapping des Class-Attributes im HTML Code. Ihr großer Vorteil liegt in der höheren Geschwindigkeit, die sich aufgrund der Tatsache ergibt, dass keine extra Abfragen ausgeführt werden müssen, um Eigenschaften zur Identifikation der PageElements in Erfahrung zu bringen. Ein klassischer Anwendungsbereich hierfür sind die Einträge eines Navigationsmenüs. Die Klasse PageElementMap stellt im Klassendiagramm zudem eine Ausnahme dar: Sie enthält das Attribut „\$“, welches öffentlich sichtbar und damit nicht „private“ (wenn auch read-only) ist und den Zugriff auf die gemappten PageElements über deren im mappingObject definierten Bezeichner ermöglichen.

Darüber hinaus ist auch die Verwendung einer `PageElementGroup` möglich. Diese kann aus allen Arten von `PageNodes` bestehen und ist etwa für das Testen von Formularen nützlich. In Zusammenarbeit mit dem „`PageElementGroupWalker`“ stellt sie eine Variante des Composite Patterns dar, denn sie ermöglicht es mit nur einem Funktionsaufruf, dasselbe Problem auf all ihren „Kindern“ zu lösen - beispielsweise das Ausfüllen vom Formularfeldern.

Erzeugt werden `PageNodes` von „`PageElementStores`“. Diese ermöglichen es, je nach benötigtem Einsatzbereich nur jene `PageNodes` anzubieten, die in dem jeweiligen Kontext auch gebraucht werden. Beispielsweise können eigene `PageElementStores` für Formularfelder oder für jene Bausteine, die nur auf einer bestimmten Seite vorkommen, angelegt werden. Darüber hinaus werden `PageElementStores` für die Festlegung von Default-Parametern und -Typen der `PageNodes` verwendet und sie cachen Instanzen von `PageNodes`, indem sie alle `PageNodes` mit demselben Selektor und Typ als Singleton-Instanzen verwalten.

2.4 Framework-Bausteine und Tool Customization

Im vorhergehenden Kapitel wurde die Architektur des Frameworks aus der Vogelperspektive betrachtet. Dieses Kapitel erhöht nun die „Zoomstufe“: Anhand von Code-Beispielen werden Aufbau und Funktionsweise der einzelnen Framework-Bausteine näher beleuchtet. Zudem wird darüber berichtet, an welchen Stellen Anpassungen an externen Framework-Bestandteilen erforderlich waren, um die Kompatibilität mit dem „Core“ von WDIO-WORKFLO zu gewährleisten. Schließlich erfolgt auch ein kurzer Einblick in das Commandline Interface von WDIO-WORKFLO.

2.4.1 Specs

Specs repräsentieren die Anforderungen an eine Softwareanwendung in einer für WDIO-WORKFLO testbaren Form:

```
1  // additionalRequirements.spec.ts
2
3  Feature("Additional Requirements", {}, () => {
4    Story("4.1", "Main Menu", {issues: ["KBCPP-324"], severity: "critical"}, () => {
5      Given("the user is logged in")
6      .And("the user resides on the dashboard page", () => {
7        When("the user clicks the 'open menu' button", () => {
8          Then(1, "the menu is displayed")
9          Then(2, "the username is displayed in the menu")
10         Then(3, "the menu items are displayed")
11
12         When("the user clicks on a menu item", () => {
13           Then(4, "the corresponding page is loaded")
14         })
15       })
16     })
17   })
18 })
```

Abbildung 19: Beispiel von Specs in WDIO-WORKFLO

Abbildung 19 zeigt, dass Specs aus Stories bestehen, welche in Features zusammengefasst werden. Die Features entsprechen den „Epics“ der agilen Terminologie und dienen der Gruppierung der Stories in übergeordnete Bereiche.

Jede Story besitzt eine ID („4.1“), einen Titel („Main Menu“), sowie ein Metadatenobjekt, welches beispielsweise den Schweregrad (bei Versagen) einer Anforderung definiert. Zudem können über das Metadatenobjekt Verlinkungen zu den Quellen der beschriebenen Anforderung, etwa dem Issue-Tracking Tool Jira [40], hergestellt werden.

Im Body der Story Funktion wird die durch die User Story beschriebene Anforderung in eine funktionale Systembeschreibung übersetzt, die die Interaktionen zwischen System und BenutzerInnen nachstellt und durch Akzeptanzkriterien die erwarteten Resultate definiert.

Hierfür wird eine Formalisierung der Anforderung durch eine Kombination aus natürlicher Sprache und den Schlüsselwörtern „Given“, „When“, „Then“ und „And“ vorgenommen – auch bekannt unter dem Namen „Gherkin Language“, welche Emrich [8, pp. 70-73] als eine mögliche Form von Behavior Driven Development genauer erläutert.

Demzufolge ist diese Art der Entwicklung aus Test Driven Development, also dem Schreiben der Testfälle vor der Implementierung der Anforderungen, heraus entstanden. Im Falle von Behavior Driven Development werden jedoch nicht nur die Testfälle, sondern sogar die Akzeptanzkriterien in „ausführbarer“ Form definiert. Diese als „ubiquitous language“ bezeichnete Formulierungsweise erlaubt eine gemeinsame Sprache zwischen EntwicklerInnen, TesterInnen und BenutzerInnen zu schaffen.

„Given“ definiert die Vorbedingungen, welche für das zu überprüfende Verhalten gegeben sein müssen. „When“ drückt eine Interaktion der BenutzerInnen mit dem System aus und „Then“ spezifiziert die Akzeptanzkriterien der Anforderung, welche sowohl automatisiert als auch manuell validiert werden können, und macht diese über eine eindeutige Nummer identifizierbar.

„Given“ und „When“ beschreiben abstrakte Testschritte und können sowohl in sequentiellen (mittels „And“) als auch verschachtelten Abfolgen aneinander gekettet werden. Zum Schluss (oder „ganz innen“) der Kette steht stets das Schlüsselwort „Then“. Die komplette Abfolge an abstrakten Testschritten vom ersten „Given“ bis zu einem „Then“ wird in weiterer Folge als „Handlungsstrang“ bezeichnet.

Ein Handlungsstrang sucht immer den „Weg zur Mitte“ - er läuft die abstrakten, verschachtelten Schritte von außen nach innen ab. Pro „Then“ ist ein Handlungsstrang erforderlich. So lautet der vollständige Handlungsstrang für Akzeptanzkriterium 1 aus Abbildung 19:

*Given the user is logged in
And the user resides on the dashboard page
When the user clicks the ‚open menu‘ button
Then the menu is displayed*

Folgen in einem Handlungsstrang mehrere gleiche, verschachtelte Schlüsselwörter aufeinander, werden alle Vorkommen nach dem Ersten durch „And“ ersetzt. Der Handlungsstrang für Akzeptanzkriterium 4 aus Abbildung 19 lautet demgemäß:

*Given the user is logged in
And the user resides on the dashboard page
When the user clicks the ‚open menu‘ button
And the user clicks on a menu item
Then the corresponding page is loaded*

2.4.2 Manual Results

Im Falle, dass bestimmte Akzeptanzkriterien der Specs nur händisch überprüfbar sind, oder wenn die Entscheidung getroffen wurde, diese aus Zeit- oder Kostengründen nicht automatisiert zu testen, bietet WDIO-WORKFLO die Möglichkeit, Akzeptanzkriterien manuell zu validieren.

Die Resultate der händischen Validierung können durch Implementierung des TypeScript-Interfaces „Workflo.IManualTestcaseResults“ bereitgestellt werden, wie Abbildung 20 zeigt:

```
1  // additionalRequirements.man.ts
2
3  const additionalRequirementsResults: Workflo.IManualTestcaseResults = {
4    "4.1": {
5      2: {result: true, date: '10.12.2017'},
6      4: {result: false, date: '10.12.2017', comment: 'fails only in IE11'}
7    }
8  }
9
10 export default additionalRequirementsResults
```

Abbildung 20: Beispiel von Manual Results in WDIO-WORKFLO

Im Grunde handelt es sich hierbei lediglich um ein typsicheres JSON-Objekt, welches die IDs der überprüften Stories, sowie deren Akzeptanzkriterien mit den Attributen „result“, „date“ und „comment“ enthält.

Jedes „result“ kann nur die Werte „true“ und „false“ annehmen. Das Attribut „date“ hilft bei der Planung manueller funktionaler Systemtests, wenn etwa alle händisch validierten Anforderungen erneut getestet werden sollen, deren letzte Überprüfung mehr als drei Wochen zurückliegt (siehe Kapitel 2.4.10).

Das letzte Attribut „comment“ erlaubt es TesterInnen, den manuellen Testfall betreffende Besonderheiten zu einem bestimmten Akzeptanzkriterium zu vermerken.

WDIO-WORKFLO arbeitet mit einer fest vorgegebenen Ordnerstruktur und liest bei der Ausführung von Tests alle im Manual Results Ordner abgelegten „.man.ts“ Files ein, sodass TesterInnen der Aufwand, diese selbst importieren zu müssen, erspart bleibt.

2.4.3 Testcases

Das Testskript automatisierter Testfälle, welches die Nachstellung von Anwendungsfällen und Geschäftsprozessen ermöglicht, wird in WDIO-WORKFLO durch Testcases definiert, wie Abbildung 21 veranschaulicht:

```
1 // menu.tc.ts
2
3 import steps from '?/steps'
4 import {pages, common} from '?/page_objects'
5 import * as helpers from '?/helpers'
6
7 suite("Menu Suite", {}, () => {
8   testcase("toggle menu", {}, () => {
9     const user = helpers.dataHelpers.getStandardUser()
10
11     given(steps["logged in as standard user"]())
12     .when(steps["open menu"]){
13       cb: () => {
14         const menuUsername = common.menu.userName.getText()
15
16         validate({"4.1": [1]}, () => {
17           expect(common.menu.isOpen()).toBe(true)
18         })
19         validate({"4.1": [2]}, () => {
20           expect(menuUsername).toEqual(`${user.vname.toUpperCase()} ${user.nname.toUpperCase()}`)
21         })
22       }
23     })
24   })
25
26   testcase("go to proposals page", {bugs: ["KBCPP-666"], severity: "blocker"}, () => {
27     given(steps["logged in as standard user"]())
28     .when(steps["open %{page} page via menu"]){
29       arg: {page: PeterPan.PageKeys.proposals},
30       cb: () => {
31         validate({"4.1": [3, 4], "10.1": [1]}, () => {
32           expect(pages[ PeterPan.PageKeys.proposals ].isOpened()).toBe(true)
33         })
34       }
35     })
36   })
37 })
```

Abbildung 21: Beispiel von Testcases in WDIO-WORKFLO

Testcases werden in Suites gruppiert und können über die Zusammensetzung der Namen von Suite und Testcase eindeutig identifiziert werden. Wie auch bei Specs ist bei Testcases eine Angabe des Schweregrads (im Falle eines Scheiterns) des Testcases möglich und es können Verlinkungen zu Incident Management Tools angegeben werden.

Ein Testcase setzt sich aus einem oder mehreren parametrisierten Steps zusammen. Im Gegensatz zu den abstrakten Testschritten in den Specs sind die Steps in den Testcases jedoch konkrete Implementierungen, was sich in Abbildung 21 unter anderem am Import des „steps“ Objekts in Zeile 1, sowie der Parametrisierung des Steps „open %{page} page via menu“ in Zeile 27 zeigt.

In den Callbacks der Steps kann der Zustand des Systems nach Ausführung der Steps mit Hilfe der „validate()“ Funktion überprüft werden. Der erste Parameter der validate() Funktion erlaubt es, Verknüpfungen zu den durch die Validierung überprüften Stories und deren Akzeptanzkriterien herzustellen. Im Body der validate() Funktion werden die mithilfe von Jasmine verfassten Assertions ausgeführt, die die tatsächlichen Resultate mit den erwarteten vergleichen.

2.4.4 Steps

Steps stellen gekapselte, wiederverwendbare Interaktionen mit dem System dar. Der Aufbau und die Eigenschaften von Steps werden nun anhand von Abbildung 22 näher beschrieben:

```
1  //userManagement.step.ts
2
3  import { ParameterizedStep } from 'wdio-workflo'
4  import steps from './steps'
5  import { pages } from './page_objects'
6
7  interface ICredentials {
8    username: string
9    password: string
10 }
11
12 const UserManagementSteps = {
13   "open login page": (params?: IOptStepArgs<{}, string>): IParameterizedStep =>
14     new ParameterizedStep(params, (): string => {
15       browser.reload()
16       pages.login.open()
17       return browser.getUrl()
18     }),
19
20   "login as %{username}": (params: IStepArgs<ICredentials, void>): IParameterizedStep =>
21     new ParameterizedStep(params, ({username, password}): void => {
22       pages.login.form.SetValue({values: {
23         username, password
24       }})
25       pages.login.login()
26     }),
27
28   "logged in as %{username}": (params: IStepArgs<ICredentials, void>): IParameterizedStep =>
29     new ParameterizedStep(params, (credentials): void => {
30       steps["open login page"]({
31         cb: url => console.log(`logged in at ${url}`)
32       }).execute()
33       steps["login as %{username}"]({ arg: credentials }).execute()
34     })
35 }
36
37 export default UserManagementSteps
```

Abbildung 22: Beispiel von Steps in WDIO-WORKFLO

Jeder Step in WDIO-WORKFLO ist eine Instanz der Klasse „ParameterizedStep“ und kann über das Objekt „steps“ aufgerufen werden. Dabei werden TesterInnen durch das IntelliSense Feature von Visual Studio Code unterstützt, welches die Suche nach Steps über deren Namen ermöglicht, und auf Basis der bisherigen Eingabe jene Treffer mit der höchsten Übereinstimmung vorschlägt, wie in Abbildung 23 zu sehen ist:

```
given(steps["logged in as"]())
.when(steps["open %{page}"]()) {
  arg: {page: PeterPan.Pa
  cb: () => {
    validate({"4.1": [3,
    expect(pages[ Peter
```

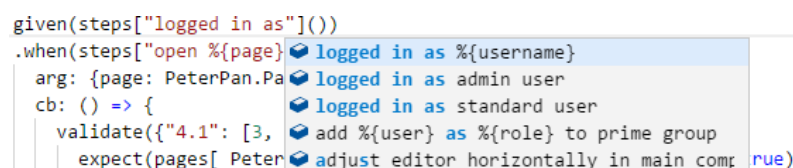


Abbildung 23: IntelliSense unterstützt bei der Auswahl von Steps

Zum Aufruf eines Steps kann ein Argument-Object sowie ein Callback festgelegt werden, welches die Überprüfung des Systemzustands direkt nach Abschluss eines Steps ermöglicht. Sowohl Argument-Object als auch Callback sind typisiert, um Fehlbedienungen des Steps zu minimieren.

In Zeile 20 von Abbildung 22 wird etwa der Typ des Argument-Objects als „ICredentials“ definiert, welcher sich aus den Strings „username“ und „password“ zusammensetzt, und der Typ des Step Results ist „void“. In Zeile 13 hingegen ist der Typ des Step Results „string“, da der Step „open login page“ die URL der geöffneten Login Seite als String zurückgibt.

Der Rückgabewert eines Steps wird an das Callback übergeben, wie zum Beispiel in Zeile 32 von Abbildung 22: Hier wird die vom Step „open login page“ retournierte URL innerhalb des Callbacks in der Konsole ausgegeben.

Der Name eines Steps kann auf Basis des Argument-Objects interpoliert werden, was vor allem beim Lesen der Logs der Testausführung von großem Nutzen ist. So wird in Zeile 20 von Abbildung 22 die Passage „%{username}“ in der Logausgabe durch den Wert des „username“ Properties des Argument-Objects ersetzt.

Eine besonders nützliche Eigenschaft von Steps ist, dass sich diese aus beliebig vielen anderen Steps zusammensetzen lassen. Der Step „logged in as %{username}“ in Abbildung 22 etwa besteht aus den beiden Steps „open login page“ und „login as %{username}“. Auf diese Weise können Steps unterschiedlicher Abstraktionsebenen definiert werden.

Auf der untersten Abstraktionsebene, welche die technische Implementierung darstellt, greifen Steps auf Page Objects zu, um mit dem System zu interagieren. Ein Beispiel hierfür ist der Step „login as %{username}“. Dieser befüllt zunächst das Formular auf der Login Seite mit den Werten „username“ und „password“ und ruft danach die login() Funktion der Login Seite auf.

2.4.5 Page Objects

2.4.5.1 Page Object Pattern

Wie bereits in Kapitel 2.3.3 beschrieben, sind Page Objects ein Entwurfsmuster, welches häufig beim Testen graphischer Benutzeroberflächen eingesetzt wird, um den inneren Aufbau und die Logik von Bausteinen der Oberfläche hinter einer applikationsspezifischen Schnittstelle zu verbergen. Die Robustheit der automatisierten Testfälle wird somit erhöht, da Änderungen an der Struktur oder dem Verhalten eines Bausteins bei Anwendung des Page Object Patterns nur an einer Stelle, nämlich im Page Object selbst, angepasst werden müssen. Zudem steigen Verständlichkeit und Lesbarkeit des Testcodes, wenn dieser mit einer applikationsspezifischen Schnittstelle kommuniziert, wie der Vergleich einer Testpassage für das Schließen eines Dialogs ohne und mit Page Object Pattern verdeutlicht:

```
browser.elements('//div[@class="Dialog" and .//h2[.="Angebotseinstellungen"]//button[@id="closeButton"]').click()
browser.waitForExist('//div[@class="Dialog" and .//h2[.="Angebotseinstellungen"]]', 5000, false)
```

Abbildung 24: Schließen eines Dialogs ohne Einsatz des Page Object Patterns

Der Code in Abbildung 24 ist auf das HTML Markup des Dialogs und einzelne Selenium Befehle fokussiert und verwendet das Page Object Pattern nicht. Er zeigt die WebDriverIO-Instanz „browser“, die über das „elements“ Command auf einen Button klickt, der den Dialog schließt. Der String zur Selektion des Buttons ist schwer lesbar und anfällig für Fehler und Änderungen im Aufbau des Dialogs. In der zweiten Zeile wird auf das Verschwinden des Dialogs innerhalb von fünf Sekunden gewartet, was auf den ersten Blick auch nicht sofort ersichtlich ist.

Im Vergleich dazu lässt Abbildung 25 durch Anwendung des Page Object Patterns viel eher die Absicht des Tests erkennen, den Angebotseinstellungen-Dialog zu schließen:

```
dialogs.proposalSettings.close()
```

Abbildung 25: Schließen eines Dialogs unter Einsatz des Page Object Patterns

Das Page Object Pattern wird in WDIO-WORKFLO durch die Klassen „Pages“, „PageElements“, „PageElementList“, „PageElementMap“ und „PageElementGroup“ umgesetzt. Zudem werden die Klassen „PageElementStore“ und „PageElementGroupWalker“ zur Verwaltung und Bedienung von Page Objects zur Verfügung gestellt. All diese Klassen stellen eine allgemeine Struktur und Basisfunktionalität zur Verfügung. AnwenderInnen des Testframeworks können Kinderklassen erstellen, die von diesen Basisklassen erben, deren Verhalten erweitern oder bestimmte Basisfunktionalitäten bei Bedarf auch überschreiben.

Der Rest dieses Kapitels beschreibt die einzelnen Page Object Klassen von WDIO-WORKFLO im Detail.

2.4.5.2 Pages

Pages spiegeln den Aufbau sowie das Verhalten einer bestimmten Webseite, eines Dialogs oder eines Seitenfragments (beispielsweise einer von mehreren „Tabs“ auf einer Seite) wieder. Sie setzen sich aus mehreren Bausteinen, in WDIO-WORKFLO „PageNodes“ genannt, zusammen, und bilden deren grundlegende Interaktionen untereinander, sowie mit den BenutzerInnen des Systems, ab, wie Abbildung 26 präsentiert:

```
1  // login.page.ts
2
3  import * as config from '~/config/test_config'
4  import { workflow, core } from '?/page_objects'
5
6  export class LoginPage extends workflow.pages.Page<core.stores.AnnaStore> {
7    constructor() {
8      super({elementStore: core.stores.anna})
9    }
10
11    get container() {
12      return this.elementStore
13        .Element('//div[@id="loginPage"]')
14    }
15
16    get form() {
17      const page = this
18
19      return page.elementStore
20        .ValueGroup({
21          get username() {
22            return page.container.$
23              .Input('//input[@name="username"]')
24          },
25          get password() {
26            return page.container.$
27              .Input('//input[@name="password"]')
28          }
29        })
30    }
31
32    get loginButton() {
33      return this.container.$
34        .Element('//button[@type="submit"]')
35    }
36
37    login() {
38      this.loginButton.click()
39    }
40
41    open() {
42      browser.url('/')
43      this.waitForOpened()
44    }
45  }
```

Abbildung 26: Beispiel einer Page in WDIO-WORKFLO

Die besagten PageNodes stellen allerdings lediglich eine abstrakte Klasse dar. Deren konkrete Implementierung kann in Form der Klassen „PageElement“, „PageElementList“, „PageElementMap“ oder „PageElementGroup“, welche in den folgenden Unterkapiteln noch näher erläutert werden, erfolgen. Der Zugriff auf Instanzen dieser vier Klassen wird von „PageElementStores“ verwaltet, weshalb Pages allesamt von der Basisklasse „workflo.pages.Page“ ableiten, welche sicherstellt, dass jede Page zumindest über einen „PageElementStore“ verfügt. Realisiert wird dieser Zugriff über JavaScript Getter Funktionen, welche erst zum Zeitpunkt ihres Aufrufs evaluiert werden, und nicht zum Zeitpunkt der Instanzierung der Page Klasse. Der Vorteil, der sich daraus ergibt, kann anhand der Zeilen 11 bis 14 von Abbildung 26 veranschaulicht werden:

Es wird vom „elementStore“ ein PageElement angefordert, welches ein HTML Element vom Typ „div“ und einem Wert des id-Attributs von „loginPage“ im Testframework abbildet, und dieses unter dem Namen „container“ den AnwenderInnen der Page Klasse zur Verfügung gestellt. Das „container“ PageElement kann während der Testausführung beliebig oft angefordert und auf diesem weitere Funktionen aufgerufen werden, etwa „isVisible()“. Nun sollte bei jedem Aufruf von „isVisible()“ der derzeitige Zustand der GUI überprüft werden, und nicht der Zustand der GUI zu jenem Zeitpunkt, zu dem die Page Klasse instanziiert wurde. Wäre dies nicht der Fall und das „container“ PageElement während der Klasseninstanzierung nicht auf der Webseite sichtbar gewesen, so würden alle spätere Überprüfungen der Sichtbarkeit ebenfalls den Wert „false“ zurückliefern – selbst wenn das Element zu diesem Zeitpunkt tatsächlich auf der Webseite sichtbar ist.

In Kapitel 2.3.3 wurden Pages als das „Tor in die Welt der Page Objects“ bezeichnet – sie fungieren also als Schnittstelle der Page Objects nach außen hin. Wie erwähnt stellen Pages jedoch ebenso eine Zusammensetzung aus PageNodes dar, und bilden somit ein „Mapping“ der getesteten Webseiten auf Page Objects ab. Pages werden dieser Doppelfunktion gerecht, indem ihre public API einerseits einen direkten Zugriff auf die gemappten PageNodes ermöglicht (insofern diese nicht explizit in ihrer Sichtbarkeit eingeschränkt wurden), und andererseits Schnittstellenfunktionen anbietet, die komplexe innere Vorgänge im Zusammenspiel der einzelnen PageNodes nach außen hin kapseln. In diesem Sinne können Pages auch als Mischung einer Komposition in der Terminologie von UML und dem „Facade“ Entwurfsmuster [41] bezeichnet werden.

In der Praxis hat sich diese Dualität als eine effiziente Lösung erwiesen, denn während viele häufig auftretende Anfragen an Pages - etwa, eine Seite zu öffnen und auf deren vollständiges Laden zu warten - durch Schnittstellenfunktionen wie „open()“ (siehe Zeilen 41 bis 44 in Abbildung 26) abgedeckt werden können, gibt es immer wieder Situationen, in denen ein direkter Zugriff auf die einzelnen PageNodes Sinn macht – etwa, um zu prüfen, ob ein Button sichtbar ist. Es wäre ineffizient und unübersichtlich, dafür die Schnittstellenfunktion „isLoginButtonVisible()“ zu definieren, denn eine Page kann aus einer großen Anzahl an PageNodes bestehen, von denen die meisten dann ebenfalls einer eigenen „isXXXVisible()“ Funktion bedürften.

Stattdessen wird die Sichtbarkeitsüberprüfung auf das Button-Element selbst übertragen, wie Abbildung 27, welche den Aufruf von Schnittstellenfunktionen der Page dem direkten Zugriff auf die einzelnen PageNodes bildlich gegenüberstellt, demonstriert:

```
// access interface function "open()" of Page "login"
pages.login.open()

// access PageNode "loginButton" of Page "login" directly
// use API of "loginButton" to check for its visibility
const result = pages.login.loginButton.isVisible()
```

Abbildung 27: Schnittstellenfunktionsaufrufe sowie direkter Zugriff auf PageNodes

2.4.5.3 PageElement

Wurden Pages als das „Tor in die Welt der Page Objects“ beschrieben, so trifft auf PageElements am ehesten die Bezeichnung „Herz der Page Objects“ zu. PageElements spiegeln die einzelnen GUI-Komponenten der getesteten Webanwendung in WDIO-WORKFLO wieder und stellen damit jene wiederverwendbaren Grundbausteine dar, auf denen alle weiteren konkreten Page Object Klassen aufbauen. Abbildung 28 zeigt eine PageElement Implementierung der „Checkbox“ Komponente von React Toolbox [24]:

```
1  import { pageObjects as workflow } from 'wdio-workflo'
2  import { AnnaStore } from '../stores'
3
4  export interface ICheckboxOpts<Store extends AnnaStore> extends workflow.elements.IPageElementOpts<Store>
5  {}
6
7  export class Checkbox<Store extends AnnaStore> extends workflow.elements.PageElement<Store> {
8    constructor(selector: string, options : ICheckboxOpts<Store>) {
9      super(selector, options)
10   }
11
12   get input() {
13     return this.$.GetOnlyInput('//input[@type="checkbox"]')
14   }
15
16   get div() {
17     return this.$.Element('//div[@data-react-toolbox="checkbox"]')
18   }
19
20   get span() {
21     return this.$.Element('//span[@data-react-toolbox="label"]')
22   }
23
24   isEnabled() {
25     return !this.getAttribute('class').includes( 'theme__disabled__' )
26   }
27
28   toggle() {
29     this.div.element.click()
30     return this
31   }
32
33   check() {
34     if ( !this.div.element.getAttribute('class').includes( 'theme__checked__' ) ) {
35       this.toggle()
36     }
37     return this
38   }
39 }
```

Abbildung 28: Implementierung der Checkbox Komponente von React Toolbox [24]

Die Komplexität von PageElements kann stark variieren und etwa von einfachen Buttons bis hin zu „Multi-Select Dropdowns“ mit Reset-Funktion reichen. In diesem Punkt erfolgt nun eine Klarstellung in Bezug auf Kapitel 2.3.3, welches PageElements bereits überblicksmäßig beschrieben, dabei jedoch eine wichtige Tatsache verschwiegen hat:

Genauso wie „PageElementLists“, „PageElementMaps“ und „PageElementGroups“ können sich auch PageElements selbst aus anderen PageElements zusammensetzen, wie das in Abbildung 28 präsentierte Beispiel einer „Checkbox“ zeigt, welche aus zwei PageElements vom Typ „Element“ und einem PageElement vom Typ „GetOnlyInput“ besteht.

Allerdings sind PageElements die einzige Page Object Klasse, die als Blattknoten im strukturellen Aufbau einer Webseite in Frage kommt. Aus diesem Grund beherbergen PageElements nahezu sämtliche Schnittstellenfunktionen, welche Interaktionen mit der Weboberfläche des getesteten Systems ermöglichen. In Abbildung 28 etwa stellt „Checkbox“ die Schnittstellenfunktionen „check()“ und „toggle()“ zur Verfügung, welche speziell auf diese Implementierung eines PageElements zugeschnitten sind.

Eine große Anzahl an Schnittstellenfunktionen erben alle PageElement Implementierungen darüber hinaus von ihrer Basisklasse „workflo.elements.PageElement“, welche bis auf wenige Ausnahmen als einzige Page Object Klasse direkt mit der API von WebDriverIO kommuniziert.

Die Schnittstelle der PageElement Basisklasse gliedert sich in folgende, im Anschluss näher erläuterte Bereiche:

- „waitXXX()“ Funktionen
- Funktionen zur Zustands- und Werteabfrage
- „eventuallyXXX()“ Funktionen
- die Funktion „click()“
- das Property „\$“

Durch die verteilte Architektur von Webapplikationen kommt es vor allem an Systemschnittstellen ständig zu Wartezeiten, die ein Testframework berücksichtigen muss. Das Laden einer Seite, die Authentifizierung von neu angemeldeten Usern oder die Abfrage großer Datenmengen, etwa in Form einer Tabelle, stellen nur einen kleinen Auszug dieser Warteszenarien dar. Doch selbst ohne systemübergreifende Kommunikation kommt es im Browser immer wieder zu Verzögerungen, wenn etwa Berechnungen auf Client-Seite durchgeführt werden oder eine große Anzahl an Elementen gerendert wird.

Die PageElements Basisklasse stellt daher eine ganze Reihe an „waitXXX()“ Schnittstellenfunktionen zur Verfügung (etwa „waitExist()“, „waitVisible()“, „waitText()“), welche eine explizite Angabe des Wartetyps und des dafür maximal eingeräumten Timeouts ermöglichen.

Um nicht den gesamten Code mit Wartefunktionen zu übersäen, unterstützt sie zudem einen impliziten Wartemechanismus, welcher die Konfiguration eines generellen Wartetyps und Timeouts bereits bei der Erzeugung eines PageElements erlaubt. Diese implizierte Wartefunktion wird vor jeder „eventuallyXXX()“ und „click()“ Funktion, sowie vor jedem Zugriff auf das „\$“ Property implizit vom PageElement selbst aufgerufen.

Die Funktionen zur Zustands- und Werteabfrage geben Auskunft über Informationen, die sich aus dem Aufbau der gemappten HTML-Elemente und deren Attributen herleiten. Dazu zählen unter anderem die Funktionen „getId()“, „getClass()“ oder „getText()“, welche einen String, sowie die Funktionen „isEnabled()“, „isExisting()“ und „containsText()“, welche einen Booleschen Wert zurückliefern.

Die Funktion „isEnabled()“ in Abbildung 28 ist außerdem ein Beispiel dafür, dass sämtliche Schnittstellenfunktionen der PageElement Basisklasse bei Bedarf durch gleichnamige Funktionen in abgeleiteten PageElements Klassen überschrieben werden können.

Eine Kombination aus Zustands- und Werteabfragen, sowie „waitXXX()“ Funktionen, stellen die „eventuallyXXX()“ Funktionen dar. Sie prüfen, ob ein bestimmter Zustand innerhalb einer gewissen Zeit eintritt.

Funktionen zur Zustandsänderung beschränken sich in der PageElement Basisklasse auf die Funktion „click()“. Diese erlaubt etwa die Auswahl von Werten in Dropdowns, Checkboxes und Radioboxes, oder das Klicken von Buttons und deckt dadurch bereits die meisten Formen von Zustandsänderung ab.

Die Möglichkeit der Texteingabe mittels der WebdriverIO Funktion „setValue()“ wird durch die PageElement Basisklasse jedoch nicht unterstützt, da sie sich auf Inputs und TextAreas beschränkt und daher nicht allen Arten von PageElements zur Verfügung steht. Gleiches gilt für die Abfrage von „Values“ der Input- und TextArea Felder durch die WebdriverIO Funktion „getValue()“. Es empfiehlt sich in diesen Fällen die Implementierung eines eigenen „Input“ PageElements, welches von der PageElement Basisklasse ableitet und die beiden beschriebenen WebdriverIO Funktionen kapselt.

Eine besondere Rolle kommt dem read-only Property „\$“ zu, welches als einziger Bestandteil der öffentlichen Schnittstelle der PageElement Basisklasse nicht als Funktion aufgerufen wird. Es dient dazu, verschachtelte PageNodes vom PageElementStore des PageElements anzufordern. Dafür werden die Selektoren des ursprünglichen PageElements und des neuen, verschachtelten PageNodes verknüpft. Zeile 21 in Abbildung 28 demonstriert den Einsatz des „\$“ Properties zur Erzeugung verschachtelter PageElements am Beispiel eines „span“ Elements, welches Teil der inneren, gemappten HTML-Struktur von „Checkbox“ ist. Der Aufruf dieses verschachtelten PageElements von außen erfolgt mithilfe der „.“ Notation, die in JavaScript allgemein zum Zugriff auf innere Properties von Objekten eingesetzt wird, und ist daher für JavaScript EntwicklerInnen intuitiv, wie Abbildung 29 darstellt:

```

11  get container() {
12      return this.elementStore
13          .Element('//div[@id="loginPage"]')
14  }
15
16  get rememberMeCheckbox() {
17      return this.container.$.Checkbox('//div[@data-react-toolbox="checkbox"]')
18  }
19
20  doSomething() {
21      const rememberMeSelector = this.rememberMeCheckbox.span.getSelector()
22
23      /*
24       * prints:
25       * '//div[@id="loginPage"]//div[@data-react-toolbox="checkbox"]//span[@data-react-toolbox="label"]'
26       */
27      console.log(rememberMeSelector)
28  }

```

Abbildung 29: Verknüpfung der Selektoren verschachtelter PageElements

Auf diese Weise können beliebig tief verschachtelte Strukturen in Page Objects durchlaufen werden. In Abbildung 29 wird beispielsweise zunächst die „rememberMeCheckbox“ innerhalb des Containers der Login Seite ausgewählt, und danach auf dem „span“ PageElement, welches ein innerer Bestandteil der „Checkbox“ ist, die Funktion „getSelector()“ aufgerufen. Der in der Konsole ausgegebene Selektor zeigt, von links nach rechts, die von außen nach innen verschachtelten, gemappten Bausteine der HTML Seite. Das Format der in WDIO-WORKFLO eingesetzten Selektoren ist XPath [42] – eine Sprache zur Selektion von Nodes in XML (und HTML) Dokumenten, welche eine sehr hohe Flexibilität erlaubt, da sie die Struktur des Dokuments ebenso berücksichtigen kann, wie die Werte der Attribute von Nodes, sowie, im Falle von Text-Nodes, deren textuellen Inhalt. WDIO-WORKFLO's Page Objects können XPath-Selektoren als Strings entgegennehmen und bieten darüber hinaus einen XPath-Query-Builder an, der den Umgang mit der teils etwas komplizierten Syntax vereinfacht, wie Abbildung 30 veranschaulicht:

```

get pane() {
    // results in selector '//div[contains(@class, "__pane__") and .="My Pane"]'
    return this.$.Element(xpath('//div').containedClass('__pane__').text('My Pane'))
}

```

Abbildung 30: Der XPath-Query-Builder von WDIO-WORKFLO

Abschließend sei noch auf ein grundlegendes Prinzip von PageElements hingewiesen, welches sich anhand von Abbildung 28 gut beobachten lässt: Es werden keine Informationen über den Zustand der Weboberfläche innerhalb der PageElements gespeichert – diese sind also „stateless“. Die „reale“ Systemoberfläche und deren Schattenabbildung durch Page Objects können somit nie auseinanderlaufen.

2.4.5.4 PageElementList

Die Basisklasse PageElementList erlaubt eine Sammlung mehrerer PageElement Instanzen desselben Typs zu verwalten. Sie ist für Einsatzszenarien vorgesehen, in denen die Anzahl und der Inhalt der einzelnen PageElements im Vorhinein nicht bekannt ist – etwa Posts in einem Forum, welche zur Laufzeit hinzugefügt oder gelöscht werden können. Die Schnittstellenfunktionen der PageElementList Klassen gliedern sich in vier Gruppen:

- Funktionen, die Auskunft über die Anzahl der Elemente in der Liste geben
- Funktionen, die die Auswahl von Listenelementen nach gewissen Kriterien erlauben
- „waitXXX()“ Funktionen
- „eventuallyXXX()“ Funktionen

Zur ersten Gruppe, welche Auskunft über die Anzahl der Elemente in der Liste gibt, zählen die Funktionen „getLength()“ sowie „isEmpty()“.

Die zweite Gruppe, welche die Auswahl von Listenelementen nach gewissen Kriterien erlaubt, ermöglicht dies über zweierlei Arten:

Einerseits können mit der Funktion „getAll()“ zunächst alle Listenelemente zurückgegeben, und diese danach durch Iteration der einzelnen PageElements und Aufruf deren Schnittstellenfunktionen gefiltert oder anderweitig verwendet werden, wie Abbildung 31 verdeutlicht:

```
/**
 * Returns a value object, in which the keys correspond to the checkbox
 * labels and the values are a boolean representation of whether the
 * checkbox is checked or not.
 */
getValue() {
    const value: Record<string, boolean> = {}

    this.checkboxList.getAll().forEach(
        checkbox => {
            value[checkbox.span.getText()] = checkbox.getValue()
        }
    )

    return value
}
```

Abbildung 31: Iteration über alle von einer PageElementList verwalteten PageElements

Andererseits bietet PageElementList die Funktionskette „firstBy()“ an, welche das erste Listenelement auswählt, dessen Selektor den definierten Funktionskriterien entspricht. Da hierbei gleich das gewünschte Element aus der Liste selektiert wird, und nicht zuerst alle Listenelemente für eine spätere Filterung zurückgegeben werden müssen, ist diese Methode der Selektion eines Listenelements üblicherweise weitaus schneller.

Ein Beispiel hierfür ist in Abbildung 32 zu sehen, wobei aus der „optionsList“ jenes PageElement angeklickt wird, dessen Textinhalt dem Wert von „translation“ entspricht:

```
const translation = "I was told about this website by a friend"

this.optionsList.firstBy().text(translation).get().click()
```

Abbildung 32: Auswahl eines Listeneintrags über die „firstBy()“ Funktionskette

Die dritte Gruppe der Wartefunktionen umfasst etwa die Schnittstellenfunktionen „waitExist()“, „waitVisible()“ und „waitText()“, welche darauf warten, dass zumindest ein Element der Liste existiert/sichtbar ist/einen Textinhalt hat. Darüber hinaus bietet die Funktion „waitLength()“ die Möglichkeit, die Testausführung anzuhalten, bis etwa eine neu generierte Tabellenzeile im DOM der Webseite hinzugefügt wurde.

Die vierte und letzte Gruppe der Schnittstellenfunktionen von PageElementList stellt analog zu PageElement die Klasse der „eventuallyXXX()“ Funktionen dar, welche überprüfen, ob die Liste nach einer bestimmten Zeit einen gewissen Zustand erreicht hat. Hierzu zählen beispielsweise die Funktionen „eventuallyHasLength()“ und „eventuallyIsEmpty()“.

2.4.5.5 PageElementMap

Die Basisklasse PageElementMap erlaubt es, PageElements desselben Typs anhand bestimmter Selektor-Eigenschaften auf zuvor definierte Keys zu mappen. Sie werden in Situationen eingesetzt, in denen die Struktur der gemappten PageElements im Vorhinein bekannt und während des Testens unveränderbar ist. Ein häufiges Anwendungsbeispiel stellen etwa Navigationsmenüs von Webseiten dar, wie Abbildung 33 verdeutlicht:

```
53 | protected displayedMenuEntries =
54 | {
55 |   dashboard: 'Dashboard',
56 |   infoboard: 'Infoboard',
57 |   profile: 'Profile',
58 |   proposals: 'Proposals',
59 |   containers: 'Content Administration',
60 | }
61 |
62 | get navigation() {
63 |   return this.container.$.ElementMap(
64 |     '//span[@data-react-toolbox="list-item-text"]', //mapSelector
65 |     {
66 |       identifier: {
67 |         mappingObject: this.displayedMenuEntries,
68 |         func: (mapSelector, mappingValue) => xpath(mapSelector).text(mappingValue)
69 |       }
70 |     }
71 |   )
72 | }
```

Abbildung 33: Anwendungsbeispiel von PageElementMap für Navigationsbereich

PageElementMaps ermöglichen somit, eine große Anzahl von gleichartigen PageElements mit wenig Code zu erzeugen und Änderungen an der Struktur der gemappten PageElements sehr rasch, nämlich allein durch Anpassung des mappingObjects, durchzuführen.

In Abbildung 33 erkennt man, dass für die Erstellung von PageElementMaps zwei Parameter erforderlich sind: erstens ein „mapSelector“, der die gesamte Menge aller gemappten PageElements selektiert, und zweitens ein „identifier“ Objekt, welches sich aus einem „mappingObject“ sowie einer „func“ zusammensetzt und die Identifikation der einzelnen PageElements ermöglicht. Dazu wird für jeden Eintrag im „mappingObject“ die „func“ mit den Parametern „mapSelector“ sowie „mappingValue“ aufgerufen, wobei „mappingValue“ den Wert des gerade iterierten Key-Value Pairs aus „mappingObject“ darstellt. Im Body der „func“ kann anhand der Parameter „mapSelector“ sowie „mappingValue“ ein beliebiger XPath-Selektor zusammengebaut werden, welcher die einzelnen PageElements eindeutig identifiziert.

Der Zugriff auf die gemappten PageElements erfolgt über das bereits bekannte „\$“ Property, nur dass diesmal das jeweilige PageElement nicht vom PageElementStore, sondern von PageElementMap selbst verwaltet wird und daher nicht über Funktionsaufrufe, sondern über die im „mappingObject“ definierten Keys verfügbar ist. Abbildung 34 illustriert die Verwendung eines Navigationsmenüeintrags aus dem obigen Beispiel via „\$“ Property:

```
// access mapped PageElement by its key name via "$"  
common.menu.navigation.$.dashboard.click()
```

Abbildung 34: Zugriff auf mit PageElementMap gemappte PageElements via "\$"

2.4.5.6 PageElementGroup und PageElementGroupWalker

Eine Page Object Klasse, die speziell für Formulare entwickelt wurde, ist „PageElementGroup“. So wie auch Formulare aus unterschiedlichen Eingabeelementen aufgebaut sein können, setzen sich PageElementGroups aus beliebigen Arten von PageNodes zusammen, also aus PageElements, PageElementLists, PageElementMaps sowie weiteren PageElementGroups. Während PageElementList und PageElementMap jedoch nur PageElements desselben Typs unterstützen, können PageElementGroups aus PageNode Implementierungen unterschiedlichster Typen (etwa Input, Checkbox und Dropdown) bestehen.

Der große Vorteil von PageElementGroups besteht darin, dass mit einem Funktionsaufruf die Werte sämtlicher enthaltenen PageNodes beispielsweise gesetzt und abgefragt werden können. Formulare werden auf diese Weise schnell mit unterschiedlichen Testdatensätzen gefüllt und überprüft.

Jede PageElementGroup besitzt zu diesem Zweck einen „PageElementGroupWalker“, welcher alle PageNodes der PageElementGroup durchläuft. Trifft er dabei auf PageElementLists, PageElementMaps oder verschachtelte PageElementGroups, so iteriert

er über deren Bestandteile und zerlegt diese weiter, bis er auf ein PageElement stößt. PageElements stellen für den PageElementGroupWalker einen Blattknoten in der traversierten Baumstruktur dar. Auf diesem werden dann die gewünschten Operationen ausgeführt.

Es handelt sich bei PageElementGroups also um eine Variante des Composite Design Patterns [43], bei der die Traversierung nicht durch das Composite selbst, sondern durch den PageElementGroupWalker vorgenommen wird.

Sollte ein PageElement die ausgeführte Operation nicht implementiert haben, so kann der PageElementGroupWalker entweder eine Fehlermeldung ausgeben, oder die betroffenen PageElements einfach ignorieren. Die zweite Option ist vor allem deswegen sinnvoll, weil PageElementGroups auch eine semantische Zusammengehörigkeit ihrer Bestandteile ausdrücken. So sind das Label eines Input-Feldes und das Input-Feld selbst zwar unterschiedliche PageElement Typen, semantisch jedoch sind sie miteinander verknüpft und werden daher gemeinsam in einer PageElementGroup platziert - selbst wenn das Label das Setzen oder Abfragen eines Wertes nicht unterstützt. Abbildung 35 zeigt die Kombination der PageElement Typen „Element“ sowie „Input“ anhand eines Login Formulars:

```
30  get loginForm() {
31      const pageContainer = this.container
32
33      return this.elementStore
34      .ValueGroup({
35          get usernameLabel() {
36              return pageContainer.$.Element('//span[.="Username"']')
37          },
38          get username() {
39              return pageContainer.$.Input('//input[@name="username"']')
40          },
41          get passwordLabel() {
42              return pageContainer.$.Element('//span[.="Password"']')
43          },
44          get password() {
45              return pageContainer.$.Input('//input[@name="password"']')
46          }
47      })
48  }
```

Abbildung 35: Verwendung einer PageElementGroup für ein Login-Formular

Damit die gewünschten Operationen auf einer PageElementGroup ausgeführt werden können, stellt die PageElementGroup Basisklasse die generische Funktion „Solve()“ zur Verfügung, welche von abgeleiteten Klassen parametrisiert werden kann und in Abbildung 36 dargestellt ist:

```

65 Solve<ValueType, ResultType>(
66     problem: Workflo.IProblem<ValueType, ResultType>,
67     options: Workflo.IWalkerOptions = {throwUnmatchedKey: true, throwSolveError: true}
68 ) : Workflo.IRecObj<ResultType> {
69     return this.__walker.walk(problem, this.__content, options)
70 }

```

Abbildung 36: "Solve()" Funktion der PageElementGroup Basisklasse

Wie man erkennen kann, erfordert die „Solve()“ Funktion die Definition von „ValueType“ und „ResultType“. Diese entsprechen den Typen von Übergabeparameter und Return-Wert jener Operation, die auf allen Blattknoten der PageElementGroup ausgeführt wird.

Als Parameter können der „Solve()“ Funktion ein „problem“, welches „Workflo.IProblem“ implementiert, sowie „options“ für den PageElementGroupWalker übergeben werden. Diese leitet daraufhin die Traversierung im PageElementGroupWalker über die Funktion „walk()“ ein und stellt diesem dafür die Parameter „problem“, „this.__content“ und „options“ bereit. Der zweite Parameter „this.__content“ entspricht dabei dem Inhalt der PageElementGroup – in Abbildung 35 etwa den zwei „Element“ sowie den beiden „Input“ PageElements mit den Namen „usernameLabel“, „username“, „passwordLabel“ und „password“.

Das Interface „Workflo.IProblem“, welches in Abbildung 37 gezeigt wird, setzt sich aus den Properties „values“ und „solve“ zusammen. „values“ ist ein rekursives Datenobjekt von Key-Values Pairs und die „solve“ Funktion stellt jene bereits oftmals erwähnte Operation dar, die auf den Blattknoten der PageElementGroup aufgerufen wird.

Dabei wird jedem Blattknoten jener Value aus dem „values“ Datenobjekt zugewiesen, dessen Key gleichnamig dem Key des Blattknotens im Content ist. Die Struktur und die Namen der Keys von „values“ Datenobjekt und Content sollten also ident sein. Die Values hingegen unterscheiden sich darin, dass diese im „values“ Datenobjekt am Beispiel des obigen Login-Formulars jene Werte enthalten, die in die Formularfelder eingetragen werden, wohingegen sie im Content die Instanzen von PageElements repräsentieren, welche die zugehörigen Formularfelder als Page Objects abbilden. Es wird also ein Mapping zwischen PageElements und Datenwerten auf Basis der Key-Namen vorgenommen.

Die „solve“ Funktion gibt ein „ISolveResult“ zurück, welches aus den beiden Properties „nodeSupported“ und „result“ besteht. „nodeSupported“ gibt Auskunft darüber, ob der gerade traversierte Blattknoten die auszuführende Operation in seiner Schnittstelle überhaupt unterstützt - wie bereits erwähnt kann eine nicht vorhandene Unterstützung durch den Walker entweder ignoriert oder ein Fehler geworfen werden. „result“ hingegen ist optional: Es liefert im Falle von Getter Funktionen den abgefragten Wert zurück und muss für Setter Funktionen nicht angegeben werden.

```

58 interface IProblem<ValueType, ResultType> {
59     values: IRecObj<ValueType>,
60     solve: <NodeType extends Workflo.PageNode.INode>{
61         node: NodeType,
62         value?: ValueType
63     } => ISolveResult<ResultType>
64 }
65
66 interface ISolveResult<ResultType>{
67     nodeSupported: boolean,
68     result?: ResultType
69 }

```

Abbildung 37: Die Interfaces "IProblem" und "ISolveResult"

Im Login-Formular in Abbildung 35 kommt die von der PageElementGroup Basisklasse abgeleitete „ValueGroup“ zum Einsatz, welche drei parametrisierte Varianten der generischen „Solve()“ Funktion anbietet: „SetValue()“, „GetValue()“ und „GetText()“. Abbildung 38 veranschaulicht den Aufruf dieser Funktionen sowie die dabei übergebenen und retournierten Datenobjekte:

```

25 const loginData = {
26     username: "Admin",
27     password: "MySecretPassword"
28 }
29
30 pages.login.loginForm.SetValue({values: loginData})
31
32 const allFilledInData = pages.login.loginForm.GetValue()
33
34 /**
35  * Prints:
36  * {
37  *   username: "Admin",
38  *   password: "*****"
39  * }
40  */
41 console.log(allFilledInData)
42
43 const filteredLabelTextData = pages.login.loginForm.GetText({
44     filter: {
45         usernameLabel: true
46     }
47 })
48
49 /**
50  * Prints:
51  * {
52  *   usernameLabel: "Username"
53  * }
54  */
55 console.log(filteredLabelTextData)

```

Abbildung 38: Aufruf der "ValueGroup"-Funktionen

Hierbei wurde die Option ausgewählt, PageElements, die eine bestimmte Operation nicht unterstützen, einfach zu ignorieren. Dies erkennt man daran, dass in Zeile 41 von Abbildung 38 die Werte der beiden PageElements „usernameLabel“ und „passwordLabel“ nicht mitausgegeben werden, da deren Typ „Element“ die Funktion „GetValue()“ nicht beherrscht. Es besteht zudem die Möglichkeit, die traversierten PageNodes zu filtern. Dieses Feature wird beim Aufruf von „GetText()“ in den Zeilen 43-47 dargestellt, wobei in diesem Fall die Funktion einzig auf dem PageNode „usernameLabel“ aufgerufen wird. Vor allem dann, wenn man in größeren Formularen nur einzelne Formularfelder ändern oder abfragen möchte, kann dadurch ein unnötiger Performance-Overhead vermieden werden.

Dass bei der Verwendung von PageElementGroups ein Formular als ganze Einheit betrachtet wird, wirkt sich auch auf das Erscheinungsbild des Testcodes aus. Anstelle der Interaktionen mit einzelnen Formularelementen treten die Input- und Output-Daten in den Vordergrund – es wird also ein datengetriebener Ansatz verfolgt. Dieser Umstand tritt in Abbildung 39, welche dieselbe Code-Passage mit und ohne Einsatz der Funktion „GetValue()“ zeigt, besonders deutlich in Erscheinung:

```
34 // check form values without using ValueGroup's GetValue() Function
35
36 const expectedUsernameValue = "Admin"
37 const expectedPasswordValue = "*****"
38
39 const actualUsernameValue = pages.login.loginForm.username.getValue()
40 const actualPasswordValue = pages.login.loginForm.password.getValue()
41
42 expect(actualUsernameValue).toEqual(expectedUsernameValue)
43 expect(actualPasswordValue).toEqual(expectedPasswordValue)
44
45 // check form values using ValueGroup's GetValue() Function
46
47 const expectedFormValues = {
48   username: "Admin",
49   password: "*****"
50 }
51
52 const actualFormValues = pages.login.loginForm.GetValue()
53
54 expect(actualFormValues).toEqual(expectedFormValues)
```

Abbildung 39: Überprüfung von Formularwerten mit und ohne Einsatz von "GetValue()"

Es lässt sich zudem feststellen, dass durch den Einsatz von PageElementGroup Funktionen in Formularen der benötigte Code deutlich reduziert werden kann.

Schließlich machen die Zeilen 39 und 40 von Abbildung 39 deutlich, dass auf PageNodes innerhalb der PageElementGroup („username“ und „password“) auf dieselbe Weise wie auch bei verschachtelten PageElements zugegriffen werden kann: mit der „.“ Notation.

2.4.5.7 PageElementStore

Die letzte Klasse, welche im Rahmen dieses Kapitels über Page Objects betrachtet wird, ist „PageElementStore“. Sie kümmert sich um die Verwaltung aller PageNode Instanzen. Ihre wohl bedeutendste Aufgabe ist die Instanziierung der PageNodes, welche in WDIO-WORKFLO ausschließlich durch PageElementStores vorgenommen werden darf. Dabei unterstützen PageElementStores ihre Clients, indem sie die instanziierten PageNodes mit Default-Parametern und Default-Typen versehen, wie anhand der Methoden „Element“ und „ExistElement“ in Abbildung 40 beobachtet werden kann:

```
3  import {
4    PageElement, IPageElementOpts,
5    PageElementList, IPageElementListOpts,
6    PageElementMap, IPageElementMapOpts, IPageElementMapIdentifier,
7    PageElementGroup, IPageElementGroupOpts,
8    TextGroup, ITextGroupOpts,
9    ValueGroup, IValueGroupOpts
10 } from '../page_elements'
11
12 export class PageElementStore {
13   protected instanceCache: {[id: string]: any}
14
15   constructor() {
16     this.instanceCache = Object.create(null)
17   }
18
19   Element(
20     selector: Workflo.XPath,
21     options?: Pick<IPageElementOpts<this>, "timeout" | "wait">
22   ) {
23     return this.get<PageElement<this>, IPageElementOpts<this>>(
24       selector,
25       PageElement,
26       {
27         store: this,
28         ...options
29       }
30     )
31   }
32
33   ExistElement(
34     selector: Workflo.XPath,
35     options?: Pick<IPageElementOpts<this>, "timeout">
36   ) {
37     return this.Element(
38       selector,
39       {
40         wait: Workflo.WaitType.exist,
41         ...options
42       }
43     )
44   }
45 }
```

Abbildung 40: Default-Typen und -Parameter von PageNodes dank PageElementStore

So erzeugen die Methoden „Element“ und „ExistElement“ beide ein PageElement, nur dass bei „ExistElement“ der Parameter „wait“ in Zeile 40 von Abbildung 40 bereits standardmäßig einen Wert von „Workflo.WaitType.exist“ inne hat. Der „store“ Parameter von „Element“ in

Zeile 27 ist zudem mit dem Wert „this“ vordefiniert und in Zeile 23 erkennt man, dass der Typ des in PageElement verwendeten PageElementStores ebenfalls auf „this“ gesetzt wird. Ohne diese „Vorarbeit“ müsste ein Client sowohl die Werte der „wait“ und „store“ Parameter manuell setzen, als auch den Typen des PageElementStores händisch definieren, was in Abbildung 41 dargestellt wird:

```
7  import { pageObjects as workflow } from 'wdio-workflo'
8
9  // manual creation - 'forbidden' in WDIO-WORKFLO
10 const existElement1 = new workflow.elements.PageElement<workflow.stores.PageElementStore>(
11   '//div',
12   {
13     wait: Workflow.WaitType.exist,
14     store: workflow.stores.pageElement
15   }
16 )
17
18 // creation via PageElementStore
19 const existElement2 = workflow.stores.pageElement.ExistElement('//div')
```

Abbildung 41: Vergleich manuell und durch PageElementStore erzeugter PageNodes

Wenn man die „manuelle“, in WDIO-WORKFLO verbotene PageNode Instanzierung in den Zeilen 10 bis 16 von Abbildung 41 mit der Erzeugung via PageElementStore in Zeile 19 vergleicht, erkennt man, dass durch das Setzen von Default-Parametern und Default-Typen in PageElementStore viel Schreibarbeit eingespart wird.

Bei Bedarf können die von PageElementStore gesetzten Default-Parameter jedoch vom Client überschrieben werden, auch wenn dies in der Praxis nur sehr selten erforderlich ist.

Eine weitere wichtige Aufgabe von PageElementStores ist das Cachen von PageNode Instanzen. Dieses verhindert, dass idente PageNodes, also solche, deren Typ, Selektor und Optionen komplett übereinstimmen, mehrere Male instanziiert werden können und den Speicherverbrauch damit unnötig in die Höhe treiben. Stattdessen liefert PageElementStore bei mehrmaligen Anforderungen identier PageNodes bereits gecachte Instanzen zurück, wie die Zeilen 311 bis 318 in Abbildung 42 veranschaulichen:

```

299     protected get<Type, Options>(
300         selector: Workflo.XPath,
301         type: { new(selector: string, options: Options): Type },
302         options: Options = Object.create(Object.prototype)
303     ) : Type {
304         const _selector = (selector instanceof XPathBuilder) ? this.xpathBuilder.build() : selector
305
306         // catch: selector must not contain |
307         if (_selector.indexOf('|||') > -1) {
308             throw new Error(`Selector must not contain character sequence '|||': ${_selector}`)
309         }
310
311         const id = `${_selector}|||${type}|||${options.toString()}`
312
313         if(!(id in this.instanceCache)) {
314             const result = new type(_selector, options)
315             this.instanceCache[id] = result
316         }
317
318         return this.instanceCache[id]
319     }

```

Abbildung 42: Caching von identen PageNodes in PageElementStore

Es empfiehlt sich, für PageNodes, die nur auf ganz bestimmten Seiten oder in ganz bestimmten Bereichen der Weboberfläche auftreten können, eigene Subklassen der PageElementStore Basisklasse anzulegen. Dadurch wird verhindert, dass PageNodes an unbeabsichtigten Stellen verwendet werden und auch die API der über PageElementStores angebotenen Instanziierungsmethoden kann so „sauber“ gehalten werden. Die Auswahlmöglichkeit „PreviewComments“, welche IntelliSense in Abbildung 43 vorschlägt, ist beispielsweise nur im „ContainerStore“ verfügbar, und wird bei der Verwendung eines anderen PageElementStores nicht angeboten.

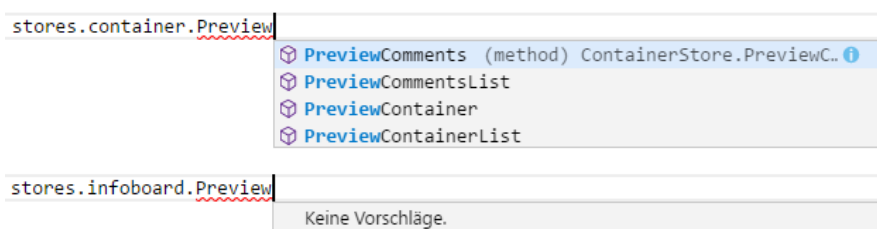


Abbildung 43: Vorschläge für Instanziierungsmethoden durch IntelliSense

Aus architektonischer Perspektive haben PageElementStores den Vorteil, dass alle Importe von PageNode Konstruktoren an einer Stelle gesammelt werden. Falls sich nun die Parameter zum Erzeugen eines bestimmten PageNodes ändern, reicht in den meisten Fällen eine Adaption der Default-Parameter und Default-Typen in der betroffenen Instanziierungsmethode aus. Nur wenn auch Parameter von Änderungen betroffen sind, welche nicht auf einen Default-Wert gesetzt werden können oder deren Default-Wert von einem Client manuell überschrieben wurde, sind Anpassungen auch im Client-Code nötig.

Hinsichtlich Entwurfsmustern stellen PageElementStores eine Mischung aus Singleton und Factory Methods [44] dar:

So wird sichergestellt, dass für idente PageNodes nur eine Instanz erzeugt werden kann und bei jeder Anfrage diese eine Instanz zurückgegeben wird – ähnlich geschieht dies auch beim Singleton Pattern, nur, dass bei diesem eine Klasse seine eigene einzigartige Instanz verwaltet, während PageElementStores die Instanzen der PageNode Klassen verwalten.

Factory Methods kapseln die Erzeugung einer Instanz in einer Methode, so dass keine Kopplung zwischen dem Client und dem Konstruktor der erzeugten Instanz besteht – dies trifft genauso auf PageElementStores zu. Die beiden gleichen sich zudem darin, dass anders als beim Aufruf eines Konstruktors, existierende Objekte wiederverwendet werden können und, dass anschaulich benannte Instanziierungsmethoden das Erzeugen von Variationen der instanziierten Objekte mit unterschiedlichen Parametern erlauben. Wie auch die Factory Methods stellen PageElementStores ein Interface zur Instanzierung von Objekten dar, jedoch unterscheiden sich PageElementStores darin, dass sie die Instanzierung nicht an Subklassen abschieben, sondern diese selbst durchführen.

2.4.6 Lists

Lists ermöglichen das Abspeichern einer bestimmten Menge von Specs und Testcases, die in einem Testlauf ausgeführt werden sollen. Sie eignen sich somit sehr gut dazu, jene wichtigsten Bereiche einer Webanwendung zu definieren, welche in Smoke Tests überprüft werden. Abbildung 44 zeigt alle innerhalb von Listen zur Verfügung stehenden Ausführungsfiler, die zur Einschränkung der Menge ausgeführter Specs und Testcases verwendet werden können:

```
// example.list.ts

const exampleList: Workflo.FilterList = {
  listFiles: ['subList'], // lists/subList.list.ts
  specFiles: ['dashboard'], // specs/dashboard.spec.ts
  testcaseFiles: ['userManagement'], // testcases/userManagement.tc.ts
  features: ['Proposal Creation'],
  specs: ['7.7', '10.1'],
  testcases: ['Container Suite', 'Menu Suite.test navigation'],
}

export default exampleList
```

Abbildung 44: Innerhalb von List-Files verfügbare Ausführungsfiler

Der Vollständigkeit halber sei erwähnt, dass Abbildung 44 nur der Auflistung der unterschiedlichen Arten von Ausführungsfilern dient. In der Realität würde durch diese Auswahl wohl keine einzige Spec und kein einziger Testcase ausgeführt werden, da die einzelnen Filteroptionen zunehmend restriktiv wirken (siehe Kapitel 2.4.10).

2.4.7 Utility Functions

Um den datengetriebenen Ansatz der PageElementGroups optimal zu unterstützen, stellt WDIO-WORKFLO einige Utility Funktionen zur Transformation von Datenstrukturen zur Verfügung. Zwei der wichtigsten Vertreter sind die Funktionen „mapToObject()“ sowie „mapProperties()“. „mapToObject()“ wandelt ein Array in ein Object um, wobei die Elemente des Arrays zu den Keys des Objects werden und die Values durch eine Map-Funktion bestimmt werden. „mapProperties()“ iteriert über alle Key-Value Pairs eines Objects und erlaubt dem Client, aufgrund der Werte von Key und Value einen neuen Value zu definieren, während der Key gleich bleibt. Die Anwendung dieser beiden Utility Funktionen demonstriert Abbildung 45:

```
26  const menuItemArray = ['dashboard', 'profile', 'proposalManager']
27
28  const menuItemObject: Record<string, boolean> = Workflo.Array.mapToObject(
29    menuItemArray,
30    element => true
31  )
32
33  /**
34   * Prints:
35   * {
36   *   dashboard: true,
37   *   profile: true,
38   *   proposalManager: true
39   * }
40   */
41  console.log(menuItemObject)
42
43  const menuItemCharCodeObject = Workflo.Object.mapProperties(
44    menuItemObject,
45    (value, key) => key.charCodeAt(0)
46  )
47
48  /**
49   * Prints:
50   * {
51   *   dashboard: 100,
52   *   profile: 112,
53   *   proposalManager: 112
54   * }
55   */
56  console.log(menuItemCharCodeObject)
```

Abbildung 45: "mapToObject()" und "mapProperties()" als Beispiele für Utility Functions

Eine besondere Form von Utility Function ist „getUid()“, welche es erlaubt, während dem Testen eindeutige Bezeichner für Testdaten zu generieren. Im Gegensatz zu den restlichen Utility Functions ist „getUid()“ nicht stateless, sondern greift im Hintergrund auf den UID Store zu. Dieser verwaltet Key-Value Pairs, wobei der Key für einen Bezeichner steht und der Value dafür, wie oft dieser Bezeichner bisher angefordert wurde.

Bei jedem Aufruf der im globalen Namespace verfügbaren Funktion „getUid()“ wird der gewünschte Bezeichner als String übergeben. Der zu dem Bezeichner im UID Store

gespeicherte Werte wird um 1 erhöht und mit dem Bezeichner zu einem neuen String verkettet und zurückgegeben, wie Abbildung 46 veranschaulicht:

```
// returns "My Proposal_1"
const proposalName = getUid('My Proposal')

// returns "My Proposal_2"
const anotherProposalName = getUid('My Proposal')
```

Abbildung 46: Der UID Store generiert eindeutige Bezeichner für Testdaten

Der Einsatz von „getUid()“ stellt sicher, dass Testcases nicht irrtümlich Testobjekte mit identen Bezeichnern anlegen, was die Funktionsweise der Tests beeinträchtigen könnte. Gerade in der Testfallentwicklung ist dies ein brisantes Thema, denn hier wird ein Testfall oft mehrmals direkt hintereinander ausgeführt und erzeugt dabei dieselben Testdaten.

2.4.8 Reporters

2.4.8.1 Überblick über zwei grundsätzliche verschiedene Reporter

Reporters sind für die Darstellung der Testergebnisse und die daraus folgende Bestimmung der Testendekriterien von großer Bedeutung. WDIO-WORKFLO bietet zwei Reporter an, deren Einsatzgebiete sich stark unterscheiden: Spec Reporter und Allure Reporter.

Beiden gemein ist, dass sie eine Auswertung der Testabdeckung sowie Statistiken über die Anzahl der getesteten Features, Specs, Suites, Testcases, Spec Files und Testcase Files anzeigen. Die Eigenheiten der beiden Reporter hingegen werden in den restlichen zwei Unterabschnitten dieses Kapitels im Detail geschildert.

2.4.8.2 Spec Reporter

Der Spec Reporter dient als schnelles Feedback und gut in VisualStudio Code integrierte Unterstützung bei der Entwicklung der automatisierten Tests. Seine Ausgabe erfolgt in der Konsole, wobei VisualStudio Code hier praktischerweise ein integriertes Terminal zur Verfügung stellt. Dadurch können die entwickelten Tests und die zugehörigen Testausgaben im selben Program angezeigt und sogar miteinander verknüpft werden. So erlaubt Visual Studio Code etwa, bei Fehlerausgaben durch einen Klick in eine Zeile des Stacktraces direkt zur betroffenen Stelle im Code zu springen. Außerdem kann der Verlauf eines ausgeführten Tests somit gleichzeitig in der Konsolenausgabe und dem Quellcode verfolgt werden.

Die Testergebnisse werden im Spec-Format ausgegeben, woraus sich der Name des Reporters ableitet. Dieses wird zur Definition von Testfällen im Stil von Behavior Driven Development (siehe Kapitel 2.4.1) eingesetzt, und soll beim Fehlschlagen eines Testfalls sofortige Rückschlüsse auf das beeinträchtigte Verhalten/die betroffenen Anforderungen der Anwendung zulassen.

Der Spec-Reporter listet dafür zunächst alle Testcases, gruppiert nach Testcase Files und Suites, und danach alle Specs, gruppiert nach Spec Files und Features, auf. Zwischen der Ausgabe der Ergebnisse der einzelnen Files werden Fehlerblöcke aller währenddessen aufgetretenen Fehler angezeigt. Um beim Entwickeln der Tests schnelleres Feedback zu erhalten (die Durchlaufzeit funktionaler Systemtests fällt im Vergleich sehr hoch aus), kann zudem eine sofortige Fehlerausgabe zum Zeitpunkt des Auftretens eines Fehlers konfiguriert werden.

Die Darstellung der Specs und Testcases im BDD-Format, sowie die Anzeige der Fehler, veranschaulicht Abbildung 47:

```
-----
[TESTCASE] Session ID: 7d5851367a3ebaa41d006c7dfee36ca6
[TESTCASE] Testcase File: C:\Users\Flo\Documents\MSE\2\MPP\framework\wdio-workflo-example\system_test\src\testcases\homepage.tc.ts
[TESTCASE] Running: chrome
[TESTCASE]
[TESTCASE] Homepage Suite
[TESTCASE]   1) display homepage
[TESTCASE]     ✓ open impressum
[TESTCASE]
[TESTCASE]
[TESTCASE]   1 passing (20s)
[TESTCASE]   1 broken
-----

1) Homepage Suite.display homepage:

Expected 'Google' to be 'Homepage'.
Error: Expected 'Google' to be 'Homepage'.
    at Spec.addExpectationResult (C:\Users\Flo\Documents\MSE\2\MPP\framework\wdio-workflo-jasmine-framework\build\adapter.js:377:42)
    at validate (C:\Users\Flo\Documents\MSE\2\MPP\framework\wdio-workflo-example\system_test\src\testcases\homepage.tc.ts:18:25)
    at Object.cb (C:\Users\Flo\Documents\MSE\2\MPP\framework\wdio-workflo-example\system_test\src\testcases\homepage.tc.ts:9:9)

Failed: An element could not be located on the page using the given search parameters ("//div[@id="asdfasdfasdf]").
Error: An element could not be located on the page using the given search parameters ("//div[@id="asdfasdfasdf]").
    at wdio_workflo_1.ParameterizedStep (C:\Users\Flo\Documents\MSE\2\MPP\framework\wdio-workflo-example\system_test\src\steps\homepa
    at testcase (C:\Users\Flo\Documents\MSE\2\MPP\framework\wdio-workflo-example\system_test\src\testcases\homepage.tc.ts:22:6)

-----
[SPEC] Spec File: C:\Users\Flo\Documents\MSE\2\MPP\framework\wdio-workflo-example\system_test\src\specs\homepage.spec.ts
[SPEC]
[SPEC] Homepage:
[SPEC]
[SPEC]   1.1 - Display Impressum
[SPEC]     ✓ Then 1: the legal disclaimer is displayed
[SPEC]
[SPEC]   1.2 - Failing story
[SPEC]     1) Then 1: the homepage title is shown
[SPEC]     2) Then 2: the user is redirected to the homepage
[SPEC]
[SPEC]
[SPEC]   1 passing (3s)
[SPEC]   1 failing
[SPEC]   1 unvalidated
-----

1) 1.2 Failing story [1 - the homepage title is shown]:

Expected 'Google' to be 'Homepage'.
Error: Expected 'Google' to be 'Homepage'.
    at Spec.addExpectationResult (C:\Users\Flo\Documents\MSE\2\MPP\framework\wdio-workflo-jasmine-framework\build\adapter.js:377:42)
    at validate (C:\Users\Flo\Documents\MSE\2\MPP\framework\wdio-workflo-example\system_test\src\testcases\homepage.tc.ts:18:25)
    at Object.cb (C:\Users\Flo\Documents\MSE\2\MPP\framework\wdio-workflo-example\system_test\src\testcases\homepage.tc.ts:9:9)

2) 1.2 Failing story [2 - the user is redirected to the homepage]:

Spec 1.2: Then 2 was not validated!
```

Abbildung 47: Ausgabe von Testcase und Spec Results im BDD-Format

Wie man in Abbildung 47 ebenso erkennen kann, erleichtert die farbige Ausgabe von Defects und Successes die Analyse des Outputs wesentlich. Dieser gliedert sich in die Kategorien „passing“ (grün), „failing“ (rot), „broken“ (gelb), „unvalidated“ (violett) und „skipped“ (hellblau – fehlt in diesem Beispiel).

Der Unterschied zwischen „failing“ und „broken“ besteht darin, dass „broken“ Testcases Defekte im Testcase selbst beschreiben, so dass dieser nicht bis zu seinem Ende ausgeführt werden konnte, während „failing“ auf Abweichungen zwischen den erwarteten und gemessenen Resultaten hinweist. „unvalidated“ Specs treten oft dann auf, wenn der Testcase, der diese validiert, unerwartet frühzeitig beendet wurde, also „broken“ ist, oder wenn das betroffene Akzeptanzkriterium (noch) von keinem Testcase validiert wird. In letzterem Fall weisen sowohl Spec Reporter als auch der im nächsten Unterabschnitt beschriebene Allure Reporter jedoch explizit auf die noch nicht eingebaute Validierung hin.

Am Ende der Testausgabe zeigt der Spec-Reporter die gesammelten Ergebnisse aller Specs und Testcases, deren Erfolgs- und Fehlerraten sowie die automatisierte und manuelle Testabdeckung an. Zudem werden Statistiken über die Anzahl der getesteten Artefakte und nochmals eine komplette Liste aller im Test aufgetretenen Fehler, gegliedert in Specs und Testcases, dargestellt. In Abbildung 48 ist der komplette Schlussbereich der Testausgabe des Spec Reporters, mit Ausnahme der Fehlerlisten, zu sehen:

```
=====
Number of Testcase Files: 1
Number of Suites: 1
Number of Testcases: 2

Testcases Duration: 27.20s
=====
Number of Spec Files: 1
Number of Features: 1
Number of Specs: 2

Specs Duration: 6.00s
=====
Criteria Coverage:

3 automated (~100%)
0 manual (~0%)
0 unvalidated (~0%)
=====
Testcase Results:

1 passing (50.0%)
1 broken (50.0%)
=====
Spec Criteria Results:

1 passing (33.3%)
1 unvalidated (33.3%)
1 failing (33.3%)
=====
```

Abbildung 48: Der Schlussbereich der Ausgabe des Spec Reporters

2.4.8.3 Allure Reporter

Während sich der Spec Reporter ausschließlich an die EntwicklerInnen der Testfälle richtet, erlaubt der Allure Reporter allen anderen Stakeholdern eines Projekts Einblick in die Testergebnisse zu nehmen. So kann etwa das Management strategische Entscheidungen aufgrund der Resultate treffen und EntwicklerInnen können die Ergebnisse von Smoke Tests und Regressionstests betrachten, um festzustellen, ob durch Änderungen am Code Fehler in die Anwendungen eingebaut wurden.

Der Allure Reporter verfügt über eine visuell ansprechende, graphische Browseroberfläche, die auf einem Webserver gehostet werden kann und dadurch permanent einsehbar ist. Der Rest dieses Abschnitts widmet sich der Beschreibung der einzelnen Seiten eines vom Allure Reporter erstellten Reports.

Um späterer Verwirrung vorzubeugen, sei an dieser Stelle jedoch zunächst auf das größte Manko des Allure Reporters hingewiesen: Dieser unterstützt die von WDIO-WORKFLO getroffene Unterscheidung zwischen Specs und Testcases nur mangelhaft, so dass die beiden in keinem der abgebildeten Diagrammen voneinander getrennt und in diesen immer nur als eine Einheit unter dem Begriff „test cases“ dargestellt werden.

Die Startseite des Allure Reports, welche in Abbildung 49 gezeigt wird, verschafft einen guten Überblick über die Ergebnisse eines Testlaufs:

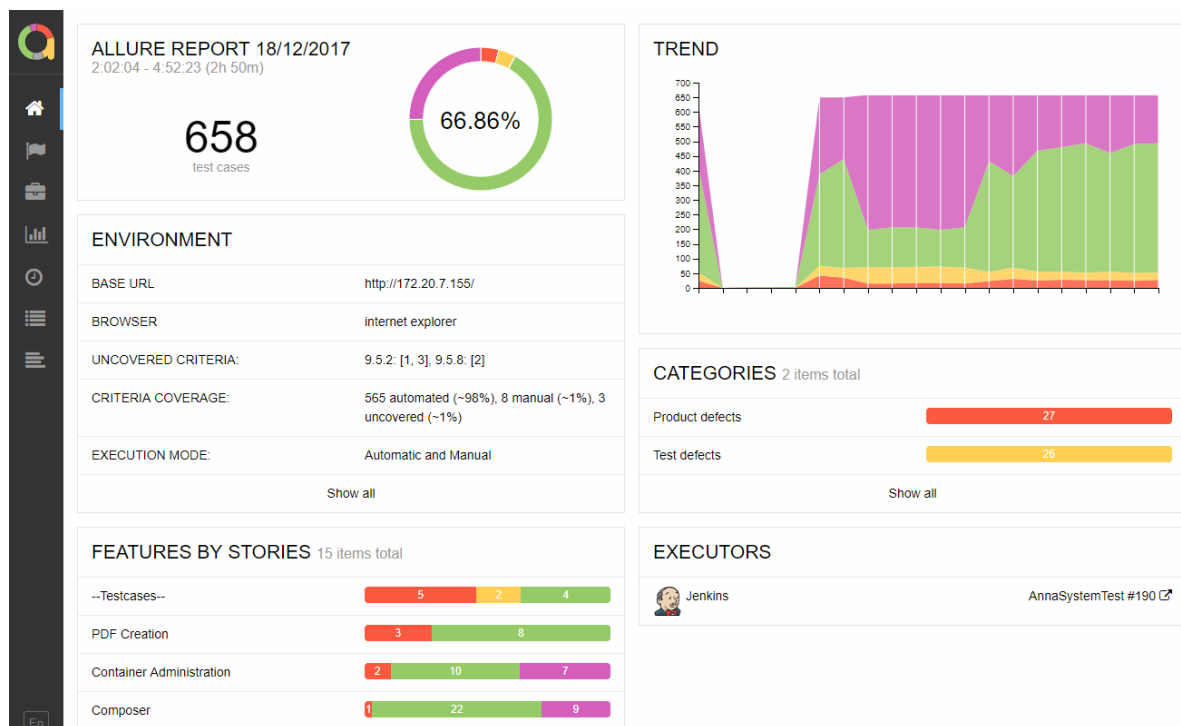


Abbildung 49: Die Startseite des Allure Reports

Neben der Dauer des Testlaufs zeigt sie die Rate an „passing“, „failing“, „broken“, „unvalidated“ und „skipped“ Testcases und Specs in einem Tortendiagramm an und erlaubt damit schnelle Rückschlüsse auf die „Gesundheit“ der getesteten Anwendung. Hilfreich ist zudem die Darstellung eines Trends, um die Entwicklung der letzten Testresultate zu bewerten.

In dem Environment Bereich können Angaben zur Testumgebung abgelesen werden und es finden sich hier auch Angaben zur Testabdeckung von automatischen, manuellen und nicht validierten Tests. Darüber hinaus können weitere Teststatistiken eingeblendet werden.

Weiters lässt sich die Testkonfiguration durch eine Verlinkung auf den getesteten Build des Continuous Integration Servers (in diesem Fall Jenkins) ermitteln.

Allure unterstützt zudem die Konfiguration von Defect Categories, wobei in diesem Fall zwischen „broken“ Testcases als „Test defects“ und „failing“ Specs oder Testcases als „Product defects“ unterschieden wird.

Der Bereich „Features By Stories“ ermöglicht schließlich eine Analyse des Status der validierten Stories, gegliedert nach Features, und bildet somit den Grad der Erfüllung der getesteten Anforderungen ab.

Alle weiteren Seiten, welche nicht ausschließlich Diagramme darstellen, sind nach einem einheitlichen Schema aufgebaut: Auf der linken Seite befindet sich eine Tabelle, in dem die betrachteten Artefakte gruppiert aufgelistet werden. Die einzelnen Gruppen können ein- und ausgeklappt werden und die Reihen der Tabelle können nach (zeitlicher) Reihenfolge, Name, Dauer und Status sortiert werden. Auch eine Suche innerhalb der Tabelle ist möglich. Auf der rechten Seite hingegen befindet sich ein Bereich, der die Details zum jeweiligen Testcase oder Spec betrachtet.

Abbildung 50 demonstriert das soeben beschriebene Layout-Prinzip am Beispiel der „Suites“ Seite, welche links eine Gruppierung der einzelnen Testcases nach deren Suites vornimmt:

The screenshot displays the Allure Reports interface. On the left, a sidebar lists various test suites: Proposal Settings Suite, Proposal Management Suite, Proposal Creation Suite, PDF Creation Suite, Composer Suite, and AdditionalRequirement Suite. The 'AdditionalRequirement Suite' is expanded, showing two test cases: '#1 load new calculation in proposal' and '#2 action buttons'. The main table lists these test cases with their respective durations and status indicators. The right panel provides detailed information for the selected test case '#1 load new calculation in proposal', including its status (Passed), severity (normal), duration (3m 36s), parameters, links, and execution steps.

order	name	duration	status
27	load new calculation in proposal	3m 36s	Passed
26	action buttons	3m 27s	Passed

Abbildung 50: Die Suites-Seite des Allure Reports

Der rechte Bereich enthält eine große Anzahl an Informationen zum ausgewählten Testcase, beginnend mit dessen ID. Nach der Anzeige von Status und Namen des Testcases erlauben drei Tabs, zwischen den Ansichten „Overview“, „History“ und „Retries“ zu wechseln.

Der „Overview“ Tab stellt zunächst die Schwere und Dauer des Testcases dar.

Danach listet er die Parameter und Links des Testcases auf, welche dessen Traceability gewährleisten. Die Parameter umfassen das Testcase File, in welchem der Testcase

implementiert wurde, sowie alle Specs, die durch den Testcase validiert wurden mit den zugehörigen Spec Files. Durch einen Mausklick auf die Links gelangt man direkt zu den in einem Issue Management Tool wie JIRA angelegten User Stories, Incidents oder Bugs. Der letzte Abschnitt im „Overview“ Tab stellt die durchlaufenen Schritte eines Testcases mit deren Übergabeparametern dar. Zu jedem Schritt werden zudem dessen Status in Form einer farbigen Markierung, sowie dessen Dauer angezeigt. Hierarchisch verschachtelte Schritte können ausgeklappt werden. Dadurch lässt sich die Stelle im Testcase, an der ein Defekt aufgetreten ist, exakt ermitteln, wie Abbildung 51 demonstriert:

▼ Test body		
➤	Given fast logged in as standard user 31 sub-steps, 52 attachments	19s 882ms
➤	and open containers page via url 8 sub-steps, 15 attachments	4s 385ms
▼	and create new container in content administration 140 sub-steps, 217 attachments	34s 571ms
➤	open container base dialog in content administration 8 sub-steps, 14 attachments	1s 936ms
➤	choose new container in base container dialog 13 sub-steps, 23 attachments	5s 720ms
▼	fill data in container settings dialog in content administration 116 sub-steps, 179 attachments	26s 912ms
➤	POST /wd/hub/session/e8a2599e-6b9a-460c-9468-a656231e7684/element 2 attachments	149ms

Abbildung 51: Hierarchisch dargestellte Steps und farblich hervorgehobene Fehler

Die Lokation des Ursprungs eines Defekts wird außerdem durch die Darstellung von Stacktraces unterstützt, wie in Abbildung 52 zu sehen ist:

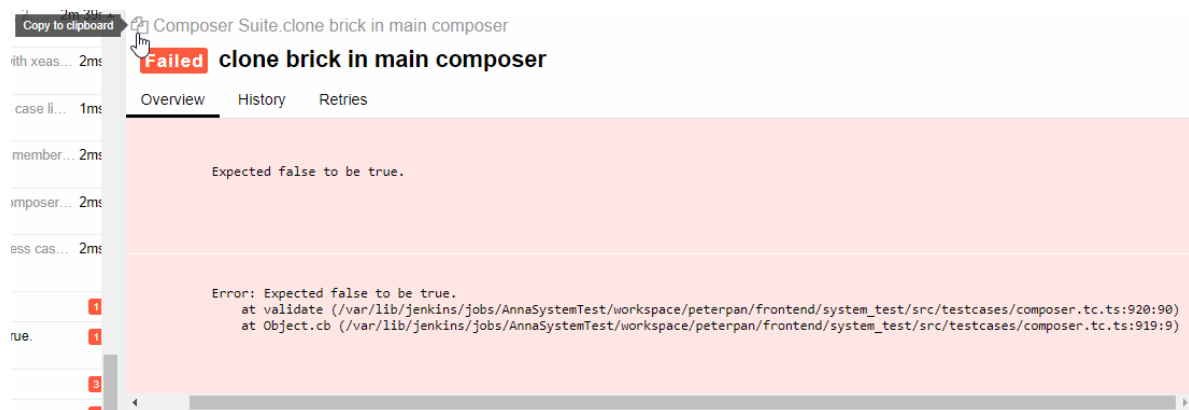


Abbildung 52: Stacktraces und Kopieren des Testcase-Namens in Allure Reports

Abbildung 52 zeigt ebenso, dass die ID des Testcases praktischerweise durch einen Mausklick direkt in die Zwischenablage kopiert werden und von dort bei Bedarf in die Konsole eingefügt werden kann. Dies ist nützlich, wenn der betroffene Testcase mithilfe des in Kapitel 2.4.10 beschriebenen Commandline Interfaces erneut ausgeführt werden soll, um aufgetretene Defekte nachzustellen oder zu korrigieren.

Schließlich erstellt WDIO-WORKFLO bei jedem Error oder Assertion Failure automatisch einen Screenshot der Browseroberfläche zum Fehlerzeitpunkt und fügt diese dem jeweiligen Step hinzu, was ebenso in großem Maße zur Analyse von Defekten beiträgt, wie Abbildung 53 veranschaulicht:

The screenshot shows the Allure Reporter interface for a test suite named 'Angebotsverwaltung'. The top section lists test steps with their durations and attachment counts:

- validate: {"8.1.4": [1, 8]} 4 sub-steps, 10 attachments (1s 287ms)
- POST /wd/hub/session/cf7c9623-68fd-470a-8611-af47d35b8f93/element 2 attachments (151ms)
- POST /wd/hub/session/cf7c9623-68fd-470a-8611-af47d35b8f93/elements 2 attachments (198ms)
- POST /wd/hub/session/cf7c9623-68fd-470a-8611-af47d35b8f93/element 2 attachments (733ms)
- GET /wd/hub/session/cf7c9623-68fd-470a-8611-af47d35b8f93/screenshot 1 attachment (165ms)
- Validation Failures (3.8 KB)
- Expected true to be false. (1) (34.6 KB)

Below this, a screenshot of a web application is displayed. The application has a header 'Angebotsverwaltung' and a search bar with the value '666666'. The main content area shows a table with columns: Angebotsbesch..., Opportunity Nr., Firmenname, VB(ID), VM (Name), Interessentenn..., Debitorennummer, Anlagennummer, Vertragsnummer, Erstellt von (ID), Erstellt von (Na..., and Erstellt am. The table contains one row with data: SearchFoundP..., 1234/1234, SearchCompany, 12, firstname Gerh..., -, 666666, 999999999, -, pp_standard_u..., PP Standard_u..., and 16.12.2017 04:33.

Abbildung 53: An fehlerhafte Steps im Allure Reporter angehängte Screenshots

Die „History“ Ansicht, welche in Abbildung 54 dargestellt wird, zeigt die Ergebnisse eines Testcases in den vergangenen Testläufen und erlaubt damit, Aussagen über die Stabilität des Testcases zu treffen:

The screenshot shows the 'History' view for the test case 'Ressourcen und Rubriken'. The top section indicates the test case is 'Passed' with a success rate of 63.15% (12 of 19). Below this, a table lists the results of individual test runs:

Result	Timestamp
passed	17/12/2017 at 3:11:17
passed	16/12/2017 at 2:57:25
passed	15/12/2017 at 3:10:28
passed	14/12/2017 at 3:09:46
broken	13/12/2017 at 3:13:58

Abbildung 54: History der Ergebnisse einzelner Tests

Die dritte und letzte Ansicht des Detailbereichs, „Retries“, dient der Veranschaulichung wiederholt aufgerufener Testfälle. Dieses Verhalten wird von WDIO-WORKFLO allerdings nicht unterstützt.

Analog zur „Suites“ Seite ist die „Behaviors“ Seite aufgebaut, nur, dass diesmal nicht die Suites und deren Testcases, sondern die Features mitsamt den Stories, aus denen sie sich zusammensetzen, abgebildet werden. Im Vordergrund der Betrachtung steht nunmehr die Aussage, welche Anforderungen der Anwendung erfüllt wurden und welche nicht. Um diese tätigen zu können, sind die Anforderungen nicht nur in Features und Stories gruppiert, sondern es wird zusätzlich die darunterliegende Ebene der einzelnen Akzeptanzkriterien einer Story betrachtet, wie man anhand von Abbildung 55 erkennen kann:

Behaviors

order name duration status

Filter by status: 27 26 440 0 165

- > --Testcases-- 5 26 45
- > Angebotseinstellungen 4 66 24
- > Composer 1 83 33
- > Composer Pricelist 5
- > Container Administration 3 37 64
- > Content Selector 1 2 2
- > Dashboard 13
 - > 6.1 - Uebersicht ueber Angebote 3
 - > 6.1.1 - Uebersicht ueber angepinnte Angebote 2
 - > 6.1.2 - Uebersicht ueber angepinnte Angebote 1
 - > 6.1.7 - Oeffnen eines Angebots oder Containers 1
 - ✓ #1 Then 1: the proposal is loaded in the main composer specs/dashboard.spec.ts, Dashboard Suite ope... 1ms
 - > 6.4 - Erstellen eines Angebots 2
 - > 8.4 - Anpinnen von Angeboten im Dashboard 4
- > Editors 4
- > Ergänzende Anforderungen 1 66 5
- > Infoboard 11

Dashboard: 6.1.7 - Oeffnen eines Angebots oder Containers.Then 1: th...

Passed Then 1: the proposal is loaded in the main composer

Overview History Retries

Severity: blocker

Duration: 1ms

Parameters

Spec File: specs/dashboard.spec.ts

Validated in Testcases: Dashboard Suite.open a proposal from dashboard prime group tab (t...

Links

KB CPP-22

Execution

Test body

- > Given the user is logged in as standard user 0s
- > When the user opens the dashboard 0s
- > and the user selects a proposal 0s
- > and the user hits the 'open proposal' button in the selected proposal's details view 0s
- > Then the proposal is loaded in the main composer 1ms

Abbildung 55: Behaviors-Seite von Allure Reports

Besonders praktisch für die Wartung von Testcases ist die „Categories“ Seite, welche die Fehlermeldungen defekter Testcases und Specs, gegliedert nach „Product defects“ und „Test defects“, auflistet, wie Abbildung 56 zeigt. So kann schnell zu den fehlerhaften Testcases oder Specs gesprungen werden, und im Falle ähnlicher Fehlermeldungen kann auf Probleme geschlossen werden, die mehrere Testcases betreffen und deren Behebung daher mit einer besonders hohen Priorität eingestuft werden sollte.

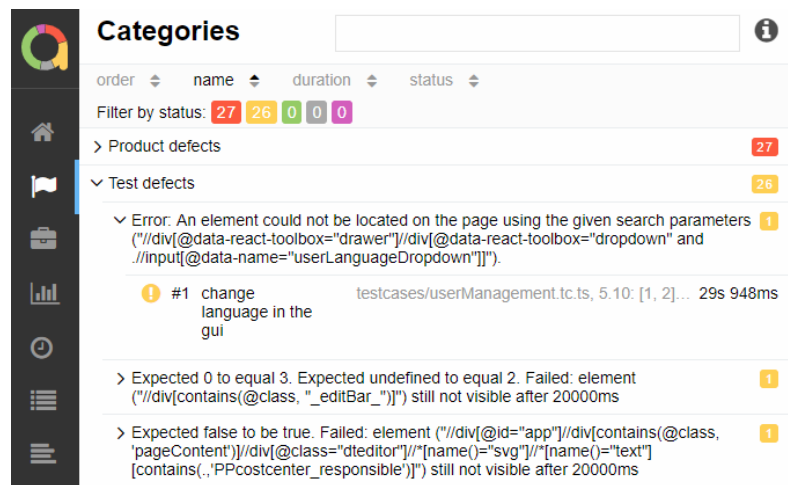


Abbildung 56: Categories Seite von Allure Reports

Eine weitere, in Abbildung 57 veranschaulichte Seite, stellt das Zeitverhalten der einzelnen Testcases dar. Neben der zeitlichen Abfolge, welche für die Bestimmung der Vorbedingungen nachfolgender Testcases von Interesse sein kann, sowie der Dauer von Testcases, können dadurch auch „Störfaktoren“, die die Testausführung beeinträchtigt haben, auf einen Blick erkannt werden. In einem konkreten Fall etwa öffnete ein am Testsystem installierter PDF Viewer unerwartet einen Dialog, der zum Akzeptieren geänderter Lizenzbedingungen aufforderte. Dadurch verlor der Testrunner die Verbindung zum Selenium Server, und alle weiteren Testfälle derselben Suite schlugen fehl. Dieser Störfaktor schlug sich in einer außergewöhnlich langen Dauer der betroffenen Testcases nieder, da diese in ein „Socket Connection Timeout“, welches auf 30 min gesetzt war, liefen.

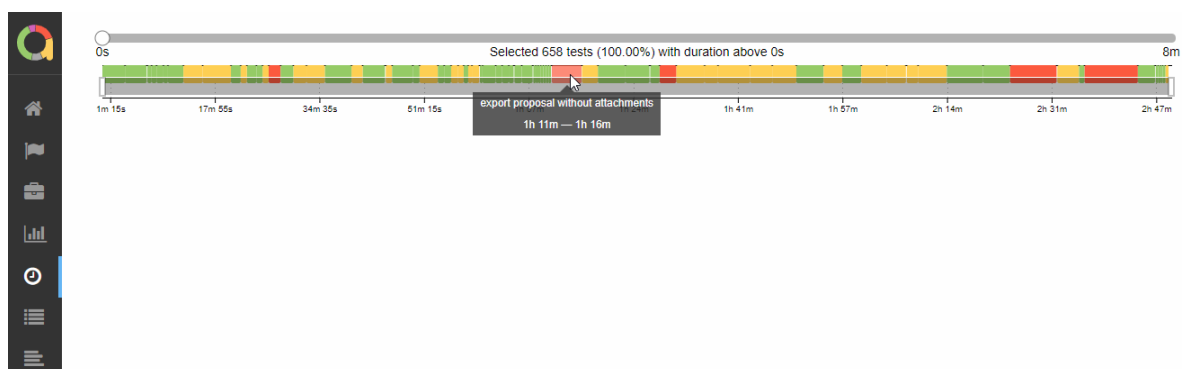


Abbildung 57: Zeitliche Abfolge und Dauer von Tests

Die in Abbildung 58 gezeigte Seite ist schließlich vor allem für das Testmanagement interessant. Der Status der einzelnen Testcases und Steps wird nicht nur, wie bereits auf der Startseite, in einem Tortendiagramm dargestellt, sondern auch nach Schwereklassen gegliedert. Dies deutet beispielsweise auf einen dringenden Handlungsbedarf hin, wenn besonders viele als „critical“ eingestufte Testcases fehlgeschlagen sind.

Außerdem kann die Aufgliederung nach Schwereklassen für die Bestimmung der Testdekriterien von Interesse sein, wenn diese etwa die Einstellung der Testaktivitäten definieren, sobald über einen bestimmten Zeitraum keine kritischen oder schweren Fehler mehr aufgetreten sind.

Zusätzlich werden die Testcases nach deren Ausführungsdauer gruppiert, wodurch sich besonders lange Testcases, die womöglich besser in kleinere Testcases zerteilt werden sollten, schneller auffinden lassen.

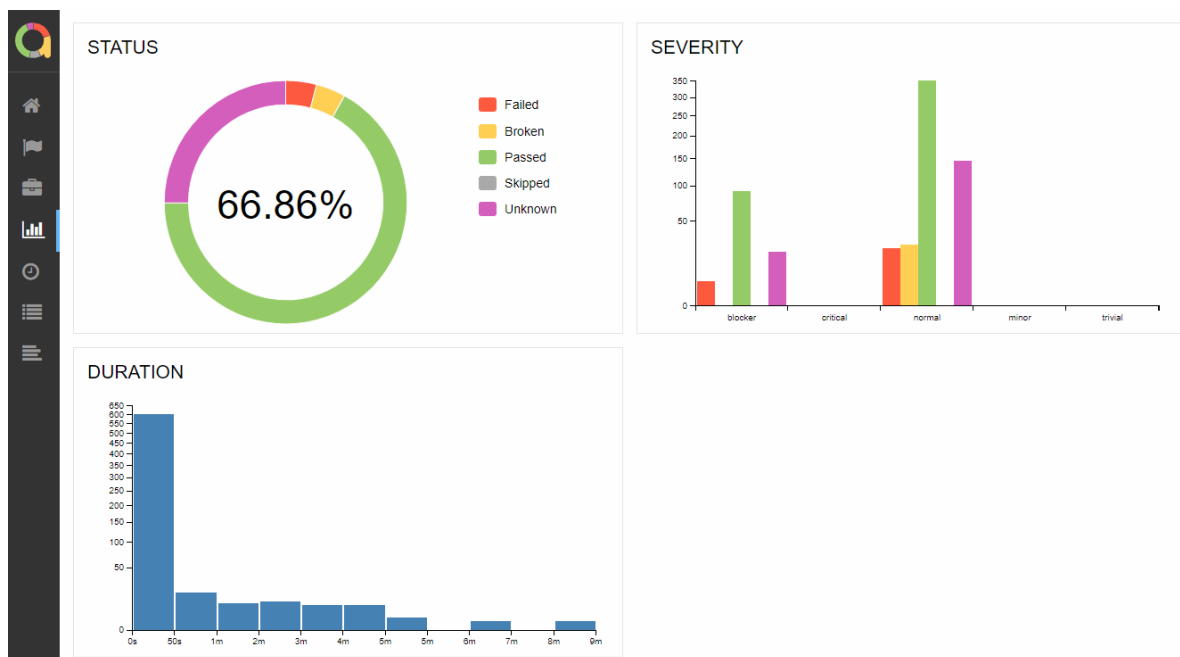


Abbildung 58: Diagramme für überblicksmäßige Informationsvermittlung

2.4.9 Customization von WebdriverIO und dessen Dependencies

Wie bereits in Kapitel 2.3.2 erläutert wurde, besteht WDIO-WORKFLO aus einem in TypeScript entwickelten Core und einigen externen Werkzeugen, welche in JavaScript geschrieben sind. Während der Core vollständig neu entwickelt wurde, waren die externen Werkzeuge zwar bereits vorhanden, mussten jedoch auf die Bedürfnisse von WDIO-WORKFLO abgestimmt werden. Dieses Kapitel beschreibt die größten erforderlichen Anpassungen in den externen Werkzeugen, ohne dabei allzu sehr ins Detail zu gehen.

Betroffen von den Anpassungen waren die npm Packages „webdriverio“, „wdio-jasmine-framework“, „wdio-spec-reporter“ und „wdio-allure-addons-reporter“. Da diese allesamt als Open Source Software zur Verfügung stehen, wurde zunächst deren Git-Repository geklont, um auch in Zukunft von Weiterentwicklungen und Bugfixes in dem jeweiligen Package profitieren zu können. Die geklonten Repositories „webdriverio-workflo“, „wdio-workflo-jasmine-framework“, „wdio-workflo-spec-reporter“ und „wdio-workflo-allure-reporter“ wurden daraufhin den besonderen Bedürfnissen von WDIO-WORKFLO entsprechend modifiziert.

Die meisten Änderungen waren am npm Package „webdriverio“ erforderlich, denn das Konzept einer Trennung zwischen Specs und Testcases ist in dessen Architektur nicht vorgesehen. WebdriverIO unterstützt in seinem ursprünglichen Zustand nämlich nur Testcases. Dementsprechend musste der Support für Specs im Rahmen der Entwicklung von WDIO-WORKFLO erst noch hinzugefügt werden. Die von Änderungen betroffenen Dateien im Package „webdriverio“ waren „launcher.js“, „runner.js“, „baseReporter.js“ und „reporterStats.js“. Deren Aufgaben seien einem besseren Verständnis halber zunächst kurz umrissen:

„launcher.js“ dient als Einstiegspunkt in den Testrunner und fungiert als dessen Schaltzentrale. Es instanziert die notwendigsten Klassen und implementiert das Messaging System von „webdriverio“, welches eine Kommunikation zwischen unterschiedlichen Kind-Prozessen ermöglicht. Pro Testcase File startet es eine Instanz von „runner.js“ als neuen Kindprozess, welcher für die Abwicklung der in der Datei definierten Testfälle zuständig ist. „runner.js“ baut zunächst eine Verbindung zum Selenium Server auf und öffnet auf dem Testsystem ein Fenster des gewünschten Browsers, in dem die Testfälle ausgeführt werden können. Während der Testfallausführung und dem Assertion Handling arbeitet „runner.js“ mit dem Package „wdio-jasmine-framework“ zusammen. Dieses nimmt die eigentliche Abarbeitung eines Testfalls vor und stellt „runner.js“ die Resultate des Testfalls zur Verfügung. „runner.js“ schickt Informationen über alle wesentlichen Ereignisse, wie etwa den Beginn oder das Ende eines neuen Testfalls sowie dessen Ergebnisse, über das Messaging System an „launcher.js“.

Der Message Handler in „launcher.js“ bereitet die empfangenen Nachrichten auf und kann diese bei Bedarf auch filtern, bevor sie an die/den Empfänger – in den meisten Fällen „baseReporter.js“, weitergeleitet werden.

„baseReporter.js“ stellt allen spezifischen Reportern wie „wdio-spec-reporter“ grundlegende Funktionalitäten für das Reporting zur Verfügung und speichert dazu etwa die Ergebnisse der Testfälle in der Klasse „reporterStats.js“ ab.

Die spezifischen Reporter können sich ebenfalls für den Empfang von Nachrichten anmelden, welche sie immer nach dem Base Reporter erhalten. Dadurch können sie einerseits die Informationen des Base Reporters verarbeiten, und andererseits auch auf Nachrichten reagieren, von denen der Base Reporter selbst gar nichts weiß – im Falle des Allure Reporters etwa alle Nachrichten bezüglich der einzelnen Steps eines Testcases. Jeder spezifische Reporter erstellt schließlich anhand der gesammelten Informationen einen Report im jeweils gewünschten Output Format, wie dem Spec Format für die Konsole oder einem Allure Report.

Im Rahmen der Entwicklung von WDIO-WORKFLO wurde neben „runner.js“, welches sich weiterhin um die Abwicklung der Testcases kümmert, die Klasse „verifier.js“ implementiert, welche für die Validierung der Specs zuständig ist. „verifier.js“ wird erst gestartet, nachdem die Ausführung aller Testcases in allen Instanzen von „runner.js“ abgeschlossen ist. Im Gegensatz zu „runner.js“ muss „verifier.js“ keine Kommunikation zum Selenium Server herstellen und auch nicht mit „wdio-jasmine-framework“ zusammenarbeiten, sondern lediglich die während der Testfallausführung gesammelten Validierungsergebnisse und aufgetretenen Fehler mit den verknüpften Specs in Verbindung bringen.

Dazu musste der Message Handler in „launcher.js“ um die Funktionalität erweitert werden, die Ergebnisse der Testcases auf die durch diesen validierten Specs zu mappen, was sehr umfangreiche Adaptionen zur Folge hatte.

Einen weiteren großen Anpassungsschritt stellte die einheitliche Umsetzung der Teststatus „passing“, „failing“, „broken“, „unvalidated“ und „skipped“ dar. Dafür musste der Message Handler in „launcher.js“ erneut verfeinert werden, um etwa zwischen „failing“ und „broken“ unterscheiden zu können. Zusätzlich mussten sowohl Base Reporter als auch Spec und Allure Reporter um die Möglichkeit erweitert werden, auf die einzelnen Statustypen zu reagieren und auch die Klasse „reporterStats.js“ musste dahingehend angepasst werden, dass diese die Resultate der Testcases und Specs kategorisiert nach den definierten Status abspeichern konnte.

Da WDIO-WORKFLO zahlreiche Filtermöglichkeiten bietet, um die Auswahl der ausgeführten Testcases und der validierten Specs einzuschränken (siehe Kapitel 2.4.10), WebdriverIO und dessen Dependencies jedoch Filter nur auf Dateiebene unterstützen, mussten auch in diesem Bereich Erweiterungen vorgenommen werden. Die vom Commandline Interface ausgewerteten „ExecutionFilters“ werden mit dem „options“ Objekt von WebdriverIO gemerged, über „launcher.js“ und „runner.js“ an „wdio-jasmine-framework“ weitergegeben und stehen von da an im Ausführungskontext von Jasmine zur Verfügung.

So wie beispielsweise auf die „expect()“ Assertions von Jasmine kann dadurch nun auch auf die „ExecutionFilters“ global zugegriffen werden. Dies macht sich die Core-API von WDIO-WORKFLO zunutze und entscheidet anhand der gesetzten Filter innerhalb der Funktionen „suite()“, „testcase()“, „Feature()“ und „Story()“, ob die jeweiligen Function Bodies ausgeführt werden sollen oder nicht.

Ein letzter Bereich, in dem viele Anpassungen an den ursprünglichen externen Dependencies von WebdriverIO vorgenommen wurden, sind die Reporter Packages „wdio-spec-reporter“ und „wdio-allure-addons-reporter“. Es wurden hierbei neben der Unterstützung der definierten Teststatus vor allem Verbesserungen an der bestehenden Formatierung sowie Erweiterungen des Report Inhalts vorgenommen. So waren die erstellten Teststatistiken, welche die Testabdeckung automatisch, manuell oder gar nicht validierter Akzeptanzkriterien, sowie die Anzahl getesteter Features, Stories, Suites und Testcases beinhalten, in den ursprünglichen Packages nicht enthalten. Im „wdio-spec-reporter“ wurden außerdem ein paar vorhandene Ausgabefehler (fehlerhafte Namen von Suites und Testcases) behoben.

2.4.10 Commandline Interface

Die Steuerung von WDIO-WORLFLO findet ausschließlich über die Kommandozeile statt. Dementsprechend verfügt WDIO-WORKFLO über ein umfangreiches Commandline Interface (CLI), dessen Interaktionsmöglichkeiten im Verlauf dieses Kapitels beschrieben werden. Da eine Darstellung der kompletten Funktionalität des CLI den Rahmen dieser Arbeit sprengen würde, konzentriert sich dieses Kapitel auf dessen wichtigste Einsatzmöglichkeiten. Eine vollständige Auflistung der unterstützten Interaktionsmöglichkeiten kann über die in Abbildung 59 dargestellte Option „--help“ angezeigt werden (der Screenshot zeigt nur einen Ausschnitt aller Optionen):

```
$ ./node_modules/wdio-workflo/bin/wdio-workflo --help
wdio-workflo CLI runner

Usage: wdio-workflo [configFile] [options]
The [options] object will override values from the config file.

Options:
  --help, -h           prints wdio-workflo help menu
  --version, -v        prints wdio-workflo version
  --host              Selenium server host address
  --port              Selenium server port
  --logLevel, -l       level of logging verbosity (default: silent)
  --coloredLogs, -c    if true enables colors for log output (default: true)
  --bail              stop test runner after specific amount of tests have failed (default: 0 - don't bail)
  --baseUrl, -b       shorten url command calls by setting a base url
  --waitForTimeout, -w timeout for all waitForXXX commands (default: 5000ms)
  --info              shows static information about testcases and specs
  --testcases          restricts test execution to these testcases
                      '["Suite1", "Suite2.Testcase1"]' => execute all testcases of Suite1 and Testcase1 of Suite2
                      '["Suite2", "-Suite2.Testcase2"]' => execute all testcases of Suite2 except for Testcase2
  --features           restricts test execution to these features
                      '["Login", "Logout"]' => execute all testcases which validate specs defined within these features
                      '["-Login"]' => execute all testcases except those which validate specs defined within these features
  --specs              restricts test execution to these specs
                      '["3.2"]' => execute all testcases which validate spec 3.2
                      '["1.1", "-1.1.2.4"]' => 1.1* includes spec 1.1 and all of its sub-specs (eg. 1.1.2), -1.1.2.4 excludes spec 1.1.2.4
                      '["1.*"]' => 1.* excludes spec 1 itself but includes all of its sub-specs
```

Abbildung 59: Die "--help" Option des CLI

Der größte und häufigste Einsatzbereich des Commandline Interfaces ist die Ausführung der Tests, bei Bedarf in Kombination mit Ausführungsfiltern („Execution Filters“). Die verfügbaren Ausführungsfiler sind sehr vielseitig. Am häufigsten wird während der Testfallentwicklung nach Testcases gefiltert, um nur den soeben entwickelten Testcase auszuführen. Die Filterung nach Testcase veranschaulicht Abbildung 60:

```
$ ./node_modules/wdio-workflo/bin/wdio-workflo --testcases '["Menu Suite.navigation test"]'
```

Abbildung 60: Filterung der Testausführung nach Testcases

Ebenso ist es möglich, wie in Abbildung 61 alle Testcases einer Suite auszuführen:

```
--testcases '["Menu Suite"]'
```

Abbildung 61: Filterung der Testausführung nach Suiten

Oft soll auch ermittelt werden, ob die Akzeptanzkriterien aller Stories eines bestimmten Features erfüllt wurden. Dies wird in Abbildung 62 dargestellt:

```
--features '["User Management"]'
```

Abbildung 62: Filterung der Testausführung nach Features

Um die Filtereinstellung nach Features zu verfeinern, kann auch auf Story Ebene gefiltert werden. Wie auch für die Filtertypen „testcases“ und „features“ können bei den Stories mehrere Einträge im Filter aufgelistet werden, welche ebenso Wildcards (*) und Negationen (-) unterstützen, wie Abbildung 63 zeigt:

```
--specs '["1*", "-1.2", "2.5.3"]'
```

Abbildung 63: Kombination mehrerer Filtereinträge mittels Wildcards und Negationen

In diesem Fall etwa werden alle Stories ausgeführt, deren ID mit der Ziffer „1“ beginnt. Einzige Ausnahme hiervon ist die Story mit der ID „1.2“. Zusätzlich wird auch noch die Story mit der ID „2.5.3“ zur Ausführung gebracht.

Auf die gleiche Weise kann die Ausführung auch nach den Dateien, in welchen die Testcases implementiert und die Stories definiert sind, gefiltert werden. Hierfür stehen die Optionen „--testcaseFiles“ und „--specFiles“ zur Verfügung.

WDIO-WORKFLO ermittelt automatisch, welche Specs durch welche Testcases validiert werden. Wählt man eine Filterung nach Testcases oder Suites, werden nur jene Features und Stories ausgewertet, die von diesen auch tatsächlich überprüft werden. Dasselbe Prinzip wird genauso in die Gegenrichtung unterstützt, also, wenn nach Features oder Stories gefiltert wird.

Die gesetzten Filter gelten immer zunehmend restriktiv. Das bedeutet, zur Ausführung gelangen nur jene Testcases und ausgewertet werden nur jene Stories, die alle Filterkriterien erfüllen. Am Beispiel von Abbildung 64 wird der Testcase „Menu Suite.navigation test“ also nur ausgeführt und die Story mit der ID „1.2“ nur dann ausgewertet, wenn der angegebene Testcase diese Story auch wirklich validiert. Ist dies nicht der Fall, so werden gar keine Tests gestartet, da die angegebene Filterkombination „zu restriktiv“ ist.

```
--specs ["1.2"] --testcases '["Menu Suite.navigation test"]'
```

Abbildung 64: Ausführungsfilter in WDIO-WORKFLO

Die bisher beschriebenen Filtermöglichkeiten können zudem allesamt in Listen (Kapitel 2.4.6) vordefiniert und dann über die Angabe der Liste im CLI ausgeführt werden, was in Abbildung 65 veranschaulicht wird:

```
$ ./node_modules/wdio-workflo/bin/wdio-workflo --listFiles '["smokeTests"]'
```

Abbildung 65: Ausführung von Listen mittels CLI

Ebenso wie Listen ist auch die in Abbildung 66 präsentierte Filtermöglichkeit nach Testcases und Specs eines bestimmten Schweregrades eine interessante Unterstützung für Smoke Tests, wobei die Stufen „blocker“, „critical“, „normal“, „minor“ und „trivial“ unterstützt werden:

```
--specSeverity '["blocker"]' --testcaseSeverity '["blocker", "critical"]'
```

Abbildung 66: Filterung der Testausführung nach Schweregrad

Besonders für Fehlernachtests bietet sich zudem die Option „--rerunFaulty“ an, die standardmäßig alle defekten Testcases und Specs des letzten Testlaufs erneut ausführt. Durch explizite Angabe eines Result-Ordner Namens in einem zusätzlichen Parameter (etwa „2017-10-24_20-38-13“) können auch die defekten Testcases und Specs älterer Testläufe wiederholt werden.

Für manuell validierte Specs bietet WDIO-WORKFLO darüber hinaus die Möglichkeit, sich all jene Specs anzeigen zu lassen, die innerhalb eines bestimmten Zeitrahmens und seitdem nicht mehr überprüft wurden, wie Abbildung 67 demonstriert:

```
$ ./node_modules/wdio-workflo/bin/wdio-workflo --printStatus --manualOnly --dates '["(2017-08-01,2017-10-01)"]'
```

```
=====
[SPEC] Story
[SPEC]
[SPEC]      5.5 - Manual story
[SPEC]      ✓ [1m] (2017-08-16_00-00-00)
[SPEC]      F [2m] (2017-08-16_00-00-00)
=====
```

Abbildung 67: Anzeige aller länger nicht geprüften manuellen Specs in CLI

Die in Abbildung 67 eingesetzte „--dates“ Option kann zudem als Testausführungsfiler verwendet werden. In diesem Fall werden nur Testcases und Specs ausgeführt, deren letzte bekannte Resultate im angegebenen Zeitraum liegen. Die Optionen „--automaticOnly“ und „--manualOnly“ hingegen bringen entweder ausschließlich durch Testcases automatisch validierte Specs oder nur manuell überprüfte Specs zur Ausführung.

WDIO-WORKFLO speichert zu jedem Testcase und jedem Spec den letztbekannten Status ab. Auch nach diesem kann mit Hilfe von „--specStatus“ und „--testcaseStatus“ gefiltert werden. Außerdem kann durch die Option „--printStatus“ der letztbekannte Status für alle Testcases und Specs mitsamt dem Datum deren letzter Ausführung ausgegeben werden, wie Abbildung 68 darstellt. Dies bietet sich vor allem für sehr umfangreiche Testsuiten an, die in einem Testlauf gar nicht komplett abgearbeitet werden können. Im Falle, dass ein Testcase noch nie ausgeführt wurde, oder eine Spec noch nie validiert wurde, wird der jeweilige Status als „unknown“ markiert.

```

$ ./node_modules/wdio-workflo/bin/wdio-workflo --printStatus

=====
[TESTCASE] Chaining Suite
[TESTCASE]   x test chaining (2017-12-18_20-33-06)
[TESTCASE]
[TESTCASE]     sub suite
[TESTCASE]
[TESTCASE]       sub sub suite
[TESTCASE]         x test subs (2017-12-18_20-33-06)
[TESTCASE]
[TESTCASE] Homepage Suite 2
[TESTCASE]   F visit homepage 2 (2017-12-18_20-33-06)
[TESTCASE]
[TESTCASE] Homepage Suite
[TESTCASE]   ? visit homepage
[TESTCASE]   ? visit homepage other
=====
[SPEC] Homepage
[SPEC]
[SPEC]   1.1 - Display correct title
[SPEC]     ✓ [1] (2017-12-18_20-44-11)
[SPEC]
[SPEC]   1.2 - Failing story
[SPEC]     U [2] (2017-12-18_20-44-11)
[SPEC]     F [1] (2017-12-18_20-44-11)
[SPEC]
[SPEC]   2.2 - last failing story
[SPEC]     ? [1]
[SPEC]
[SPEC] Story
[SPEC]
[SPEC]   4.4 - Another story
[SPEC]     U [1] (2017-12-18_20-33-06)
[SPEC]
[SPEC]   5.5 - Manual story
[SPEC]     ✓ [1m] (2017-12-18_20-33-06)
[SPEC]     F [2m] (2017-12-18_20-33-06)
[SPEC]
[SPEC] Test
[SPEC]
[SPEC]   7.7 - Display correct title
[SPEC]     U [1] (2017-12-18_20-33-06)
=====
Number of Testcase Files: 3
Number of Suites: 5
Number of Testcases: 5
=====
Number of Spec Files: 3
Number of Features: 3
Number of Specs: 6
=====
Testcase Results:

0 passing (0.0%)
1 failing (20.0%)
2 broken (40.0%)
2 unknown (40.0%)
=====
Spec Criteria Results:

2 passing (25.0%)
3 unvalidated (37.5%)
2 failing (25.0%)
1 unknown (12.5%)
=====

```

Abbildung 68: Anzeige des letztbekannten Status mittels "--printStatus"

Um die Verknüpfung zwischen Testcases und validierten Specs, sowie den Files, in denen diese implementiert beziehungsweise definiert sind, zu ermitteln, kann das CLI von WDIO-WORKFLO mithilfe der Optionen „--traceTestcase“ und „--traceSpec“ Traceability Informationen in der Konsole ausgeben, ohne dass dafür Tests ausgeführt werden müssen. Im Hintergrund werden dazu alle Spec und Testcase Files geparsed und die Traceability Informationen während des Parsens gesammelt. Abbildung 69 zeigt exemplarisch die Ausgabe der Traceability Informationen zur Story mit der ID „1.2“ im CLI:

```
$ ./node_modules/wdio-workflo/bin/wdio-workflo --traceSpec 1.2

Trace information for spec '1.2':

Spec File:                specs/homepage.spec.ts
validated by Testcases:    Homepage Suite.visit homepage: [1] (testcases/homepage.tc.ts)
                           Homepage Suite 2.visit homepage 2: [2] (testcases/otherHomepage.tc.ts)
validated in Manual Results: []
```

Abbildung 69: Ausgabe von Traceability Informationen in der CLI

Schließlich wird die CLI auch zum Erzeugen von Allure Reports mit der Option „--generateReport“ und deren Anzeige mittels „--openReport“ verwendet. Beide Optionen werden durch „--report“ kombiniert. Wie bereits bei der Option „--rerunFaulty“ betreffen die Report Interaktionsmöglichkeiten standardmäßig immer den Result-Ordner des letzten Testlaufs. Durch explizite Angabe des Result-Ordner Namens, welcher dem Zeitpunkt der zugehörigen Testausführung entspricht, kann jedoch ein beliebiger Testlauf im Allure Report aufgerufen werden.

Alte Testresultate können durch die Option „--consoleReport“ jedoch ebenso im Spec Reporter Format betrachtet werden. Dabei gibt das CLI jene Zeilen, die während der ursprünglichen Testausführung in der Konsole angezeigt wurden, komplett ident erneut aus.

2.4.11 Konfiguration

Die Konfiguration aller Einstellungsmöglichkeiten von WDIO-WORKFLO findet in der Datei „workflo.conf.js“ statt, welche ein JavaScript Objekt enthält, innerhalb dessen die gewünschten Einstellungen vorgenommen werden können. Abbildung 70 stellt eine exemplarische Konfigurationsdatei von WDIO-WORKFLO dar:

```
1  const testDir = __dirname + '/system_test'
2
3  module.exports = {
4    testDir: testDir,
5    logLevel: 'verbose',
6    baseUrl: 'http://www.google.com/',
7    windowSize: {
8      width: 1280,
9      height: 800
10   },
11   webdriver: {
12     host: '127.0.0.1',
13     port: 4444
14   },
15   selenium: {
16     version: '3.4.0',
17     baseUrl: 'http://selenium-release.storage.googleapis.com'
18   },
19   capabilities: {
20     maxInstances: 1,
21     browserName: 'chrome',
22     requireWindowFocus: true,
23     nativeEvents: true,
24     unexpectedAlertBehaviour: "accept",
25     ignoreProtectedModeSettings: true,
26     "disable-popup-blocking": true,
27     enablePersistentHover: true,
28   },
29   allure: {
30     issueTrackerPattern: "https://mydomain.com/jira/browse/%s"
31   },
32   uidStorePath: `${testDir}/data/uidStore.json`,
33   reportErrorsInstantly: true
34 }
```

Abbildung 70: Die Konfigurationsdatei von WDIO-WORKFLO

Viele Einstellungsmöglichkeiten der Konfigurationsdatei können auch als Optionen an das CLI übergeben werden und überschreiben dann die Werte in der Konfigurationsdatei.

2.4.12 Ordnerstruktur

Das letzte Kapitel der Framework Beschreibung widmet sich der in Abbildung 71 dargestellten Ordnerstruktur von WDIO-WORKFLO. Diese ist aus architektonischen Gründen durch das Framework vorgegeben und kann nicht verändert werden. Einzig der „Einstiegspunkt“ in Form des Testverzeichnisses kann in der Konfigurationsdatei, wie in Abbildung 70 ersichtlich, angegeben werden.

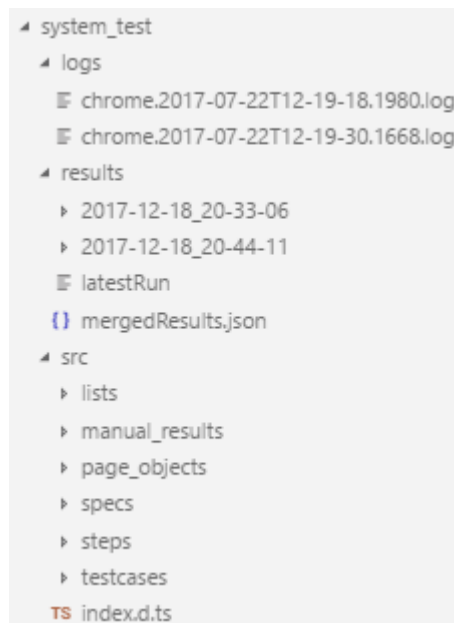


Abbildung 71: Ordnerstruktur von WDIO-WORKFLO

Wie man anhand der Abbildung 71 erkennen kann, lauten die Namen der Hauptordner in der Ordnerstruktur von WDIO-WORKFLO „logs“, „results“ und „src“.

„logs“ enthält alle von Selenium erstellten Logdateien und gibt im jeweiligen Dateinamen Auskunft darüber, welcher Browser hier zum Einsatz kam.

In „results“ wird für jeden Testlauf ein neuer Ordner mit dem Zeitpunkt des Teststarts als Name angelegt, worin alle Ergebnisse und Artefakte des Testlaufs wie etwa Screenshots abgespeichert werden. Zusätzlich befinden sich in „results“ die Datei „latestRun“, welche den Namen des zuletzt ausgeführten Testlaufordners enthält, sowie die Datei „mergedResults.json“, welche die jeweils aktuellsten Ergebnisse von Testcases und Specs über alle Testläufe hinweg speichert.

Der größte Hauptordner, „src“, gliedert sich in die Unterordner „lists“, „specs“, „steps“, „testcases“, „page_objects“ sowie „manual_results“ und spiegelt somit die wesentlichen Bausteine des WDIO-WORKFLO Cores wieder.

2.5 Evaluierung des Frameworks

2.5.1 Evaluierungsstrategie und Deployment-Umgebung

Nachdem in Kapitel 2.2 die Anforderungen an ein Framework zur Automatisierung funktionaler Systemtests in Webapplikationen recherchiert und formuliert wurden, erfolgte durch die Entwicklung von WDIO-WORKFLO die Schaffung eines Testframeworks, das auf den während der Literaturrecherche gewonnenen Erkenntnissen basiert. Kapitel 2.3 widmete sich der Beschreibung des Entwurfs und Kapitel 2.4 der detaillierten Schilderung der Implementierung von WDIO-WORKFLO.

Die folgenden Kapitel sollen nunmehr klären, in welchem Ausmaß die definierten Anforderungen im Rahmen von WDIO-WORKFLO umgesetzt werden konnten und wie sich dies auf die Qualität, sowie die Kosten der automatisierten, funktionalen Systemtests auswirkt.

Dazu muss zunächst beschrieben werden, auf welche Weise das Ausmaß der Umsetzung der Anforderungen gemessen werden kann und wie die Begriffe Kosten und Qualität zu verstehen sind.

Während die Kosten relativ simpel durch den zeitlichen und personellen Ressourcenaufwand, welcher für die Planung, Erstellung, Ausführung und Wartung der automatisierten funktionalen Systemtests erforderlich ist, definiert werden können, ist der Begriff Qualität schon viel offener für unterschiedliche Auslegungen.

Klaus Franz [4, p. 22] definiert Qualität als das Maß der Nichtabweichung des Ists vom Soll. In diesem Sinne kann die Qualität des implementierten Frameworks gleichgesetzt werden mit dem Ausmaß, in welchem dieses die an sich gestellten Anforderungen erfüllt.

Diese Definition ist vor allem für funktionale Anforderungen relevant, da diese nur individuell beurteilt werden können und die meisten in Kapitel 2.2 definierten Anforderungen sind funktionale Anforderungen. Sie entspricht zudem im Wesentlichen der in der Norm ISO/IEC 25010:2011 [34] getroffenen Definition von Funktionalität als Ausmaß, in welchem ein Produkt oder System Funktionen anbietet, um die gestellten oder implizierten Anforderungen unter bestimmten Bedingungen zu erfüllen.

Da das Testframework WDIO-WORKFLO jedoch genauso auf nicht funktionale Qualitätseigenschaften hin untersucht werden soll, werden auch die restlichen Qualitätskriterien der Norm ISO/IEC 25010:2011, nämlich Effizienz, Kompatibilität, Benutzbarkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit und Portabilität, in die Evaluierung der Qualität des Frameworks miteinbezogen.

Zur Bewertung der Erfüllung der Anforderungen werden diese, ähnlich wie in Kapitel 2.2, in einem vordefinierten Schema betrachtet, das die Felder ID, Name, Bewertung, Begründung und Nutzen enthält.

Das Feld „ID“ ist vor allem für die Referenzierung der evaluierten Anforderungen erforderlich. Die „Bewertung“ erfolgt bei booleschen Aussagen mittels der Begriffe „Erfüllt“ und „Nicht Erfüllt“ und bei gleitenden Werten anhand der aufsteigenden Skala --, -, ~, + und ++, wobei ++ für eine Erfüllung der Anforderung im höchstmöglichen Ausmaß steht und -- dafür, dass die Anforderung gar nicht oder nur in äußerst geringem Umfang umgesetzt wurde.

Die „Begründung“ erläutert die Beweggründe, wieso die Bewertung wie angegeben ausgefallen ist und das Feld „Nutzen“ beschreibt die positiven Auswirkungen der umgesetzten Anforderung auf die Qualität der Tests. Im Falle eines besonders hohen Kostensenkungspotenzials durch eine bestimmte Anforderung wird darauf ebenfalls im „Nutzen“ hingewiesen.

Um die Bewertung der Anforderungsumsetzung zudem anschaulicher zu gestalten, wurde das Testframework WDIO-WORKFLO in einer konkret vorliegenden, React-basierten Webapplikation für Angebotserstellung und -verwaltung, sowie im Rahmen eines Fallbeispiels zur Entwicklung automatisierter funktionaler Systemtests eingesetzt. Als Entwicklungsumgebung kam dabei Visual Studio Code [38] zum Einsatz, welches als Betrachtungsgrundlage sämtlicher Evaluierungen (vor allem in Bezug auf statische Codeüberprüfungen) dient.

Abschließend sei an dieser Stelle darauf hingewiesen, dass die vorgenommene Bewertung der Framework-spezifischen Anforderungen den Anspruch einer gänzlich objektiven Messung nicht erfüllt. Nach reiflicher Überlegung wurde zudem beschlossen, auch die durch WDIO-WORKFLO erzielte Kostenersparnis nicht direkt zu messen.

Diese bewussten Entscheidungen wurden aufgrund mehrerer Faktoren getroffen: So müssten, um die Qualitätssteigerung und Kostensenkung, die durch den Einsatz von WDIO-WORKFLO in einem Projekt erzielt werden, festzustellen, ausgewählte Metriken über einen langfristigen Zeitraum und in mehreren unterschiedlichen Projekten erhoben werden, um wissenschaftlich signifikante Aussagen zuzulassen. Dies ist in dem begrenzten Zeitrahmen dieser Masterarbeit, sowie aufgrund der schier Menge an zu bewertenden Anforderungen, jedoch nicht möglich.

Stattdessen wird eine semi-formale Bewertung der Anforderungserfüllung durchgeführt, die durch wissenschaftlich fundierte, in der Literaturrecherche erhobene Argumente untermauert ist. Die potenzielle Reduktion der Kosten wiederum wird indirekt durch eine Verbesserung ausgewählter Qualitätskriterien wie etwa Wartbarkeit, sowie durch die Erfüllung funktionaler Anforderungen, welche eine besonders positive Auswirkung auf die Kosten haben, abgeleitet.

2.5.2 Evaluierung des Bereichs JavaScript und React-Webapplikationen

ID	Name	Bewertung
RW1	Unterstützung für Abbildung von Website-Komponenten	++
<p>Begründung</p> <p>WDIO-WORKFLO erzwingt den Einsatz des Page Object Patterns, um die Komponenten einer Website als Testobjekte abzubilden. Page Objects können eine vollständige und identische „Schattenkopie“ ihres Vorbilds in der Webapplikation darstellen und bei Änderungen in der ursprünglichen Komponente müssen auf Testseite auch nur im jeweiligen Page Object, sowie an Stellen, wo dieses verwendet wird, Anpassungen vorgenommen werden. Durch Kapselung des internen Aufbaus und der inneren Abläufe hinter einer applikationsspezifischen API sind Page Objects zudem von außen vergleichsweise einfach bedienbar.</p> <p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ4 Benutzbarkeit:</i> Steigerung der Erlernbarkeit und Bedienbarkeit • <i>SQ7 Wartbarkeit:</i> Steigerung der Modularität, Änderbarkeit, Wiederverwendbarkeit und Analysierbarkeit 		

ID	Name	Bewertung
RW2	Unterstützung struktureller Selektoren	Erfüllt
<p>Begründung</p> <p>WDIO-WORKFLO verwendet XPath Selektoren, die eine Identifikation von Webseitenelementen sowohl über deren HTML-Attribute als auch über deren Position in der Struktur der Webseite ermöglichen.</p> <p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Erhöhung der Testbarkeit 		

ID	Name	Bewertung
RW3	Kompatibilität mit geläufigsten Browsern	+
<p>Begründung</p> <p>Gemäß der Website „netmarketshare.com“ sind die meistgebrauchten Browser im Jahr 2017 Chrome, Firefox, Internet Explorer, Edge und Safari [45]. Diese werden dank Selenium Webdriver allesamt von WDIO-WORKFLO unterstützt. Allerdings hängt der Grad der Zusammenarbeit stark vom jeweiligen Selenium-Treiber des Browserherstellers ab, und ist in manchen Ausnahmefällen nicht gegeben. Um den verwendeten Browser zu wechseln, muss schließlich nur eine Zeile in der Konfigurationsdatei angepasst werden.</p> <p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ3 Kompatibilität:</i> Das Testframework arbeitet mit unterschiedlichsten Browsern zusammen. • <i>SQ8 Portabilität:</i> Gute Anpassbarkeit an unterschiedliche Browser 		

ID	Name	Bewertung
RW4	Funktionalität und Verlinkungen der Web-Applikation müssen über den Presentation Layer effizient testbar sein	++
<p>Begründung</p> <p>Bis auf sehr vereinzelte Ausnahmen (etwa Dateiupload) können sämtliche Funktionalitäten, die auch den BenutzerInnen einer Webanwendung zur Verfügung stehen, von WDIO-WORKFLO getestet und Verlinkungen innerhalb und außerhalb der getesteten Webanwendungen überprüft werden.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ1 Funktionalität:</i> Aufgabe von WDIO-WORKFLO ist es, die Funktionalität von Webanwendungen zu testen. Da es diese erfüllt, ist somit auch die eigene Funktionalität von WDIO-WORKFLO gewährleistet. 		

ID	Name	Bewertung
RW5	Performance der Web-Applikation soll überprüfbar sein*	~
<p>Begründung</p> <p>WDIO-WORKFLO misst die Dauer von Testcases und Steps und zeigt diese in den Allure und Spec Reports an. Dadurch sind grundlegende Performance Analysen möglich. Allerdings bietet WDIO-WORKFLO keinerlei integrierte Funktionalität, um im Rahmen von Last- und Stresstests zu überprüfen, wie sich die Performance des getesteten System in Abhängigkeit der Nutzlast verhält.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Gewährleistung der Testbarkeit nicht-funktionaler Anforderungen 		

ID	Name	Bewertung
RW6	Tests sollen Bedienschritte der BenutzerInnen nachahmen	++
<p>Begründung</p> <p>Die „Steps“ von WDIO-WORKFLO können die Bedienschritte von BenutzerInnen auf unterschiedlichen Abstraktionsebenen nachbilden. So können sowohl einzelne Mausklicks, als auch das Ausfüllen eines ganzen Formulars durch einen „Step“ beschrieben werden. Über Page Objects interagieren sie, genauso wie die BenutzerInnen, mit der graphischen Benutzeroberfläche der Webanwendung. Ihnen stehen dabei exakt dieselben Mittel zur Verfügung wie auch den BenutzerInnen.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ4 Benutzerbarkeit:</i> Erlernbarkeit und Bedienbarkeit werden begünstigt, indem die Tests die Aktionen realer BenutzerInnen nachstellen 		

ID	Name	Bewertung
RW7	Usability der Web-Anwendung soll geprüft werden können*	--
Begründung WDIO-WORKFLO verfügt über keine expliziten Möglichkeiten, um die Usability einer Webanwendung objektiv zu beurteilen.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Gewährleistung der Testbarkeit nicht-funktionaler Anforderungen 		

ID	Name	Bewertung
RW8	Security der Web-Applikation soll prüfbar sein*	~
Begründung Durch Nachstellen von Login-Vorgängen sowie das Ausführen unterschiedlicher Aktionen mit bestimmten Accounts kann das Autorisierungssystem der Webanwendung von der GUI aus getestet werden. Auch weitere simple Security-Checks wie das Filtern von HTML-Code in Formulareingabefeldern kann durch WDIO-WORKFLO überprüft werden. „Hacker-Angriffe“, „Denial of Service“ Attacken und weitere komplexe Bedrohungsszenarien sind durch WDIO-WORKFLO jedoch nicht nachstellbar.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Gewährleistung der Testbarkeit nicht-funktionaler Anforderungen 		

ID	Name	Bewertung
RW9	Überprüfung der Interoperabilität soll möglich sein*	~
Begründung Die Beurteilung, inwieweit eine Webanwendung mit externen Systemen zusammenarbeitet, erfolgt in WDIO-WORKFLO nur indirekt dadurch, dass auf der GUI alle Inhalte so angezeigt werden wie erwartet. Explizite Überprüfungen an den Systemschnittstellen sind nicht möglich.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Gewährleistung der Testbarkeit nicht-funktionaler Anforderungen 		

* Performance, Usability, Security und Interoperabilität sind nicht-funktionale Anforderungen, und daher eigentlich keine zentralen Bestandteile von WDIO-WORKFLO, einem Testframework für automatisierte, **funktionale** Systemtests. Aufgrund ihrer großen Bedeutung für Webanwendungen wurde jedoch trotzdem evaluiert, inwiefern WDIO-WORKFLO bei der Überprüfung dieser Bereiche behilflich sein kann.

2.5.3 Evaluierung des Bereichs Grundzüge des Softwaretests

ID	Name	Bewertung
ST1	Trennung zwischen logischen und konkreten Testfällen	++
Begründung In WDIO-WORKFLO beschreiben Specs die logischen und Testcases die konkreten Testfälle. Specs können zeitlich und logisch unabhängig von Testcases definiert werden.		
Nutzen <ul style="list-style-type: none"> • Logische Testfälle können bereits während der Designphase, also vor der Implementierung, und bei Bedarf auch von anderen Personen als den TesterInnen, geschrieben werden. Dadurch können Fehler in den Anforderungen bereits vor der Implementierung des Source Codes aufgedeckt werden. Die Fehlerkosten sinken gemäß der „Rule of ten of defect cost“ [31, pp. 89-92] damit drastisch. • Logische Testfälle dienen EntwicklerInnen der Webanwendung als Orientierungshilfe. 		

ID	Name	Bewertung
ST2	Beschreibung der Vorbedingungen	++
Begründung Das Keyword „Given“, in Kombination mit dem Keyword „And“ in den Specs von WDIO-WORKFLO erlaubt es, Vorbedingungen als boolesche überprüfbare Konstrukte zu formulieren.		
Nutzen <ul style="list-style-type: none"> • Eine lückenlose und klar formulierte Beschreibung der Vorbedingungen ist Voraussetzung für die Wiederholbarkeit und Reproduzierbarkeit der Tests. 		

ID	Name	Bewertung
ST3	Definition des Testorakels vor Testdurchführung erzwingen	Erfüllt
Begründung Zwar ist in WDIO-WORKFLO während der Testfallentwicklung auch die Ausführung von Testcases möglich, die keine Specs validieren, sobald jedoch ein Testcase mittels der „validate()“ Funktion mit einer Spec verlinkt wird, sind die EntwicklerInnen der Testfälle gezwungen, die erwarteten Ergebnisse mittels Assertions logisch prüfbar festzulegen.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit</i>: Erhöhung der Testbarkeit • „False positives“ werden von vornherein ausgeschlossen 		

ID	Name	Bewertung
ST4	Minimierung des Aufwands für die Einrichtung der Testinfrastruktur	+
Begründung WDIO-WORKFLO stellt eine aufeinander abgestimmte Auswahl von Testrunner, Assertion Library, Testdriver (Selenium Webdriver) und Reportern zur Verfügung, die TestautomatisiererInnen mit dem simplen Befehl „npm install wdio-workflo“ out-of-the-box		

verwenden und in einer einzelnen Konfigurationsdatei anpassen können. Allerdings müssen Testdatenverwaltung, Hardware und Betriebssystem weiterhin eigenmächtig eingerichtet werden.

Nutzen

- *SQ8 Portabilität*: Verbesserte Installierbarkeit und Anpassbarkeit
- Testen kann möglichst zeitnah mit der Implementierung begonnen werden – qualitatives Feedback ist somit schon früh in dem Projekt möglich.
- Senkung der Kosten für Einrichtung der Testinfrastruktur

2.5.4 Evaluierung des Bereichs Funktionale Systemtests

ID	Name	Bewertung																												
FS1	Verifizierung der Anforderungen im Rahmen der externen Spezifikation	+																												
Begründung																														
<p>Die in Gherkin-Language verfasste Formalisierungsschicht im Rahmen der Specs von WDIO-WORKFLO macht Anforderungen logisch überprüfbar und dank der Verwendung natürlicher Sprache für alle Stakeholder verständlich. Dadurch können fehlerhafte, inkonsistente oder unvollständige Anforderungen im Rahmen der Verifizierung der externen Spezifikation frühzeitig aufgedeckt werden. Wie von Rupp et al. [18, pp. 311-319] vorgeschlagen, kommen in WDIO-WORKFLO die Formulierung von Abnahmekriterien (Synonym für Akzeptanzkriterien) sowie Prototyping (die Beschreibung der Schritte der BenutzerInnen stellt eine Art „mentalen Prototyp“ dar) als Prüftechniken für die in der externen Spezifikation verifizierten Anforderungen zum Einsatz. Wie Abbildung 72 zu entnehmen ist, sind diese beiden Methoden bei der Prüfung von Anforderungen äußerst effizient, jedoch auch aufwendig:</p>																														
<div><div><div>Effektivität</div><div>Aufwand</div><div>Formalität</div></div><table><tr><td>Stellungnahme</td><td>-</td><td>++</td><td>--</td></tr><tr><td>Inspektion</td><td>++</td><td>--</td><td>++</td></tr><tr><td>Agile SQC</td><td>++</td><td>-</td><td>+</td></tr><tr><td>Walkthrough</td><td>-</td><td>+</td><td>-</td></tr><tr><td>Prototyp</td><td>++</td><td>--</td><td>-</td></tr><tr><td>Abnahmekriterien</td><td>++</td><td>-</td><td>+</td></tr><tr><td>Modell</td><td>++</td><td>-</td><td>+</td></tr></table></div>			Stellungnahme	-	++	--	Inspektion	++	--	++	Agile SQC	++	-	+	Walkthrough	-	+	-	Prototyp	++	--	-	Abnahmekriterien	++	-	+	Modell	++	-	+
Stellungnahme	-	++	--																											
Inspektion	++	--	++																											
Agile SQC	++	-	+																											
Walkthrough	-	+	-																											
Prototyp	++	--	-																											
Abnahmekriterien	++	-	+																											
Modell	++	-	+																											
<p>Abbildung 72: Eigenschaften von Anforderungs-Prüftechniken [18, p. 319]</p>																														
<p>In Abbildung 72 wird die Effektivität durch die Anzahl der gefundenen Fehler und der Aufwand durch die benötigten Arbeitsstunden definiert. Die Skala reicht von – (sehr wenige gefundene Fehler/sehr geringer Aufwand) bis ++ (sehr viele gefundene Fehler/sehr hoher Aufwand). Da die Anforderungen in WDIO-WORKFLO, um überhaupt prüfbar zu sein, jedoch sowieso in der bereits beschriebenen Form von Specs definiert werden müssen, und Abnahmekriterien sowie Prototyping quasi ein „Nebenprodukt“ der Spec-Formulierung darstellen, ist der in Abbildung 72 als hoch eingestufte Aufwand für diese beiden Prüftechniken vernachlässigbar.</p> <p>Die ebenso erwähnten Prüftechniken Stellungnahme, Inspektion, Agile SQC, Walkthrough und Modelling bleiben jedoch außen vor, weshalb ein Punkt von der Bewertung dieser Anforderung abgezogen wird.</p>																														
Nutzen																														
<ul style="list-style-type: none">Durch frühzeitig entdeckte Fehler, Inkonsistenzen oder Unvollständigkeiten in den Anforderungen können die Fehlerkosten gemäß der „Rule of ten of defect cost“ [31, pp. 89-92] signifikant reduziert werden.																														

ID	Name	Bewertung
FS2	Validierung der externen Spezifikation	++
<p>Begründung</p> <p>Die externe Spezifikation wird in WDIO-WORKFLO durch die Specs ausgedrückt. Deren Validierung in Form der „validate()“ Funktion ist fixer Bestandteil des Testframeworks und erlaubt eine genaue Aussage darüber, welche Bedingungen zur Erfüllung welcher Akzeptanzkriterien eingehalten werden müssen. Es können pro Testcase mehrere Specs und jede Spec durch mehrere Testcases validiert werden, so dass insgesamt weniger Testcases entwickelt und ausgeführt werden müssen.</p> <p>Wie von Dustin et al. [46] gefordert, können so die in den Testcases überprüften Status den zugrundeliegenden Anforderungen zugeordnet werden. Da WDIO-WORKFLO zudem jene Akzeptanzkriterien, die durch gar keinen Testcase validiert werden, mit dem Status „unvalidated“ versieht, ist eine Aussage darüber, ob für jede Anforderung ein Testfall existiert, sehr einfach möglich.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ1 Funktionalität:</i> Gewährleistung von Vollständigkeit und Korrektheit, da vollständige und exakte Validierung der in der externen Spezifikation definierten Anforderungen sichergestellt wird. • <i>SQ2 Effizienz:</i> Der Aufwand für die Validierung sinkt durch die reduzierte Testcase-Anzahl. 		

ID	Name	Bewertung
FS3	Korrektheit, Kompatibilität und Interaktion zwischen Interfaces der einzelnen Komponenten prüfen	~
<p>Begründung</p> <p>WDIO-WORKFLO kann die Korrektheit, Kompatibilität und Interaktion zwischen Interfaces der einzelnen Komponenten nur indirekt prüfen, indem die in der GUI dargestellten Inhalte den erwarteten Resultaten entsprechen. Für ein gezielteres Testen sollten jedoch Techniken des Integration Tests gewählt werden, welche den direkten Aufruf einzelner Schnittstellenfunktionen ermöglichen.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ1 Funktionalität:</i> Gewährleistung von Vollständigkeit und Korrektheit 		

ID	Name	Bewertung
FS4	Reduzierung der Komplexität funktionaler Systemtests	++
<p>Begründung</p> <p>Beim Design von WDIO-WORKFLO wurden zahlreiche architektonische Prinzipien beachtet, die eine Reduzierung der Komplexität der funktionalen Systemtests zur Folge haben. Eine detailliertere Beschreibung dieser Prinzipien findet sich in der Evaluierung der Anforderungen FD2 Anwendung von Entwurfsmustern, FD4 Abstraktion und Separation of Concerns und FD5 funktionale Unabhängigkeit in Kapitel 2.5.10.</p>		

Nutzen	
<ul style="list-style-type: none"> • <i>SQ4 Benutzbarkeit:</i> Erhöhung von Erlernbarkeit und Bedienbarkeit • <i>SQ7 Wartbarkeit:</i> Steigerung von Modularität, Analysierbarkeit und Änderbarkeit 	

ID	Name	Bewertung
FS5	Unterstützung von Release Testing	++
Begründung Da in WDIO-WORKFLO die Abnahmekriterien in Form der in den Specs definierten Akzeptanzkriterien durch die Testcases validiert werden, bietet das Framework eine ideale Unterstützung bei der Durchführung von Release Testing an.		
Nutzen <ul style="list-style-type: none"> • <i>SQ1 Funktionalität:</i> Gewährleistung von Vollständigkeit und Korrektheit • Vermeidung „böser“ Überraschungen während den mit Kunden durchgeführten Abnahmetests 		

ID	Name	Bewertung
FS6	Berücksichtigung von Black-Box-Testfallentwurfsverfahren	+
Begründung Im Fokus der Testfallentwurfsverfahren, welche in WDIO-WORKFLO zum Einsatz kommen, steht eindeutig das anforderungsbasierte Testen, welches sich in der Definition von Specs und deren Validierung manifestiert. Da die GUI der getesteten Webanwendung jedoch im Grunde nichts Anderes ist als eine State Machine, und die Steps, die die Interaktionen von BenutzerInnen mit der GUI des Systems nachstellen, im Grunde Beschreibungen von Zustandsübergängen darstellen, spielt auch zustandsbasiertes Testen in WDIO-WORKFLO eine zentrale Rolle. Anwendungsfälle können durch die logischen und konkreten Testfälle beschrieben werden, jedoch bietet WDIO-WORKFLO darüber hinaus keine Unterstützung für anwendungsfallbasiertes Testen – als einzige Ausnahme könnte noch die Verlinkung von Issues, welche eine detaillierte Schilderung der Anwendungsfälle enthalten, in den Metadaten von Specs und Testcases gesehen werden. Äquivalenzklassenbildung und Grenzwertanalyse können für die Definition der Testdaten zum Einsatz kommen, allerdings bietet WDIO-WORKFLO hierfür ebenso wenig explizit unterstützende Funktionalitäten, wie etwa einen Testdatengenerator, an.		
Nutzen <ul style="list-style-type: none"> • <i>SQ4 Benutzbarkeit:</i> Erhöhung der Erlernbarkeit durch Einsatz bewährter Testfallentwurfsverfahren • <i>SQ7 Wartbarkeit:</i> Steigerung der Testbarkeit • Vereinfachter Entwurf von Testfällen und Testdaten • Einsatz von Entwurfsmethoden, die sich bereits zahlreich bewährt haben 		

ID	Name	Bewertung
FS7	Unterstützung von Smoke Testing	++
Begründung		
Durch Listen kann eine Auswahl jener Specs und Testcases definiert werden, die zum Aufdecken der bedeutendsten Defekte in der Anwendung beitragen sollen.		
Nutzen		
<ul style="list-style-type: none"> • Gesteigerte Qualität der Anwendung durch rasches Feedback der Smoke Tests • Die bedeutendsten Defekte der Anwendung werden zeitnah aufgedeckt und Kosten durch eine verzögerte Entdeckung der Defekte werden somit vermieden. 		

2.5.5 Evaluierung des Bereichs Testwerkzeuge

ID	Name	Bewertung
TW1	Steigerung der Testeffizienz und -abdeckung	+
<p>Begründung</p> <p>Im Vergleich zu manuell ausgeführten funktionalen Systemtests werden für die mit WDIO-WORKFLO automatisierten Tests keine menschlichen Ressourcen zur Ausführung benötigt. Je öfter funktionale Systemtests durchgeführt werden, desto effizienter ist daher deren automatisierte Ausführung. WDIO-WORKFLO misst während der Testausführung die Dauer von Testcases und Steps und erstellt automatisch Screenshots beim Auftreten von Fehlern, was in manuell durchgeführten Tests zusätzliche Aufwände mit sich brächte. Durch den Einsatz von TypeScript können zudem mittels „static checking“ Fehler in den Testfällen bereits vor deren Ausführung behoben werden, wodurch die Kosten für die Entwicklung automatisierter Tests weiter sinken.</p> <p>Andererseits ist die Automatisierung der Tests mit einem erheblichen initialen Mehraufwand verbunden und auch die in Kapitel 2.2.9 beschriebenen, architektonischen Design Prinzipien schlagen sich anfänglich als „Cost of Quality“ nieder. Daher wurde die ansonsten äußerst positive Bewertung dieser Anforderung auch um einen Punkt abgestuft.</p> <p>Die initialen Mehrausgaben rentieren sich jedoch langfristig. So können automatisierte Tests etwa jede Nacht ausgeführt werden, um den aktuellen Stand der Software zu prüfen, was manuell gar nicht oder nur mit erheblichem Mehraufwand möglich wäre. Neu in die Software eingebrachte Fehler werden somit wesentlich zeitnäher entdeckt und behoben. Einmal erstellte Page Objects und Steps können zudem beliebig oft ohne zusätzliche Kosten wiederverwendet werden, was die Testeffizienz drastisch erhöht. Steps können darüber hinaus ohne großen Aufwand mit unterschiedlichen Parametern, die in der Äquivalenzklassenbildung von Testfällen „anfallen“, aufgerufen werden, was zu einer erhöhten Testabdeckung der mittels WDIO-WORKFLO automatisierten funktionalen Systemtests führt.</p> <p>Insgesamt schlagen sich die von WDIO-WORKFLO verfolgten, architektonischen Design Prinzipien gemäß <i>SQ7 Wartbarkeit</i> in einer deutlichen Erhöhung der Änderbarkeit, Analysierbarkeit und Wiederverwendbarkeit nieder. Bedenkt man, dass Capers Jones [47, p. 4] zufolge mehr als 77 Prozent der gesamten Softwareentwicklungskosten in der Wartung der Software entstehen, bedeutet eine Investition in qualitativ hochwertigere, wartbarere automatisierte Tests langfristig gesehen eine signifikante Reduktion der durch funktionale Systemtests anfallenden Kosten.</p> <p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ2 Effizienz</i>: Reduzierung des Verbrauchs „menschlicher“ Ressourcen • Reduzierung der langfristigen Kosten für Ausführung, Entwicklung und Wartung automatisierter Tests • Gesteigerte Qualität der getesteten Software durch erhöhte Testabdeckung 		

ID	Name	Bewertung
TW2	Steigerung der Zuverlässigkeit von Tests	++
<p>Begründung</p> <p>Die als Testskripte ausgeführten, manuellen Systemtests sind immun gegenüber Gewöhnungseffekten, die bei manuellen TesterInnen häufig dazu führen, dass diese Fehler in der Software nicht mehr wahrnehmen. Ebenso sind automatisierte Tests selbst weniger fehleranfällig als manuell durchgeführte Tests und halten sich immer strikt an die in den Testskripten definierten Schritte und deren Parameter, wohingegen menschliche TesterInnen oft geringfügig vom definierten Vorgehen abweichen.</p> <p>Schließlich sind auch die mittels WDIO-WORKFLO durchgeführten Messungen, etwa jene der Dauer von Testcases oder jene der Rate fehlgeschlagener Tests, im Gegensatz zu menschlich erhobenen Werten objektiver und daher zuverlässiger.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • Höhere Rate entdeckter Fehler während der Testausführung • Vermeidung von Kosten, die durch nicht entdeckte Fehler entstehen 		

ID	Name	Bewertung
TW3	Ermöglichung von Tests, die manuell nicht durchführbar sind	--
<p>Begründung</p> <p>Abgesehen von der Messung der Dauer von Testcases und Steps bietet WDIO-WORKFLO keinerlei Unterstützung für Tests an, die aus technischer Hinsicht manuell gar nicht oder nur mit unverhältnismäßigem Aufwand durchführbar sind (etwa der Simulation von „Denial of Service“ Attacken).</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit</i>: Erhöhte Testbarkeit nicht-funktionaler Anforderungen 		

ID	Name	Bewertung
TW4	Verknüpfungsmöglichkeit zu Incident Management Tool	Erfüllt
<p>Begründung</p> <p>In den Metadaten von Specs und Testcases können Links zu Incident und Issue Management Tools wie JIRA definiert werden. Diese werden in den generierten Allure Reports angezeigt und können durch einen Mausklick geöffnet werden.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit</i>: Erhöhung der Analysierbarkeit durch bessere Traceability 		

ID	Name	Bewertung
TW5	Möglichkeit, fehlgeschlagene oder „korrigierte“ Testfälle erneut auszuführen	++
<p>Begründung</p> <p>Das CLI von WDIO-WORFKLO erlaubt durch die Option „--rerunFaulty“, fehlerhafte Testfälle eines Testlaufs erneut auszuführen. Alternativ können korrigierte Testfälle in Listen eingetragen oder als Ausführungsfilter direkt dem CLI übergeben werden.</p>		

Nutzen
<ul style="list-style-type: none"> • Reduzierter Aufwand für Fehlernachtests

ID	Name	Bewertung
TW6	Unterstützung für Konfigurationsmanagement	~
<p>Begründung</p> <p>In den Allure Reports wird auf den Build, der die Testausführung im Rahmen des Continuous Integration Prozesses ausgelöst hat, verlinkt. Dieser stellt üblicherweise genaue Informationen über die Version der einzelnen Testobjekte bereit. In WDIO-WORKFLO selbst werden die Versionen jedoch nicht aufgelistet.</p> <p>Allerdings bieten Allure Reports durch Bereitstellung eines „Trends“ und einer „History“ der Ergebnisse einzelner Testfälle die Möglichkeit, den Status von Testfällen über mehrere Versionen eines Testobjekts hinweg zu verfolgen.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • Fehler können einzelnen Versionen eines Testobjekts zugeordnet werden. 		

ID	Name	Bewertung
TW7	Statische Überprüfung des Source-Codes	Erfüllt
<p>Begründung</p> <p>Durch den Einsatz von TypeScript kann bei der Entwicklung von Tests mit WDIO-WORKFLO auf „static checking“ zurückgegriffen werden. Die Überprüfung der Syntax des Source Codes, sowie der korrekten und typengerechten Verwendung von Klassen, Funktionen und Variablen, vermeidet im Sinne einer konstruktiven Qualitätssicherung Fehler, noch bevor sich diese im Source Code „einnisten“ können. Zusätzlich werden Cross References, also Verwendungsnachweise von Variablen und Funktionen, erzeugt und Visual Studio Code's IntelliSense unterstützt die Testfallentwicklung durch zahlreiche Komfortfunktionen wie Autocomplete oder eine Vorschau der Codestelle, an welcher eine später referenzierte Variable definiert wurde.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Erhöhung der Änderbarkeit und Analysierbarkeit durch Cross References und Refactoring-Features • Reduzierung der Fehlerkosten in Testfällen selbst durch konstruktive statt analytischer Qualitätssicherung 		

ID	Name	Bewertung
TW8	Unterstützung einer Daten- und Keyword-getriebenen Testfallarchitektur	+
<p>Begründung</p> <p>Eine Daten- und Keyword-getriebene Testfallarchitektur wird in WDIO-WORKFLO durch Steps umgesetzt. Wie auch in der Keyword-getriebenen Testfallarchitektur ermöglichen Steps, geskriptete Interaktionen mit der Benutzeroberfläche der getesteten Webanwendung durch Referenzierung dieser „Code-Blöcke“ in natürlicher Sprache</p>		

wiederverwenden – allerdings kann der Name eines Steps wesentlich ausschweifender ausfallen als reine Keywords. Der datengetriebene Ansatz wird einerseits durch die Parametrisierungsmöglichkeit der Steps, und andererseits durch die in Kapitel 2.4.5.6 näher erläuterten PageElementGroups umgesetzt.

Wünschenswert wäre jedoch noch die integrierte Unterstützung eines „Testdatenrepositories“, welche WDIO-WORKFLO nicht beinhaltet.

Nutzen

- *SQ4 Benutzbarkeit:* Das Testframework ist leichter erlernbar und bedienbar, da die in natürlicher Sprache formulierten Steps den Sinn eines Testfalls durch vereinfachte Lesbarkeit besser verständlich machen.
- *SQ7 Wartbarkeit:* Erhöhung von Analysierbarkeit, Wiederverwendbarkeit und Änderbarkeit
- Steigerung der Testabdeckung durch Daten-getriebenen Ansatz
- Reduktion der Aufwände für die Testfallerstellung durch hohe Wiederverwendbarkeit der Steps

ID	Name	Bewertung
TW9	Unterstützung für Debugging	~
<p>Begründung</p> <p>Die Debugging-Features von WDIO-WORFKLO beschränken sich auf den WebdriverIO-Befehl „browser.debug()“, welcher die Ausführung eines Testfalls pausiert und es ermöglicht, über die von WebdriverIO bereitgestellten Selenium Webdriver Bindings mit der gerade im Browser getesteten Webanwendung zu kommunizieren. Die Möglichkeit, dabei einen Schritt nach vorn oder in eine Funktion zu springen, und den Code so zeilenweise auszuführen, fehlt jedoch.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ1 Funktionalität:</i> Die Unterstützung von Debugging stellt eine wesentliche Anforderung an ein Testframework dar • Bessere Analysierbarkeit von Fehlerzuständen 		

ID	Name	Bewertung
TW10	Einsatz von Komparatoren	Erfüllt
<p>Begründung</p> <p>Die in WDIO-WORKFLO eingesetzten Komparatoren entstammen der Testing Library Jasmine [21], welche zahlreiche Varianten von Assertion Matchern beinhaltet.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • Kostenersparnis und erhöhte Zuverlässigkeit, da Vergleich von erwarteten und tatsächlichen Ergebnissen automatisiert stattfindet 		

2.5.6 Evaluierung des Bereichs Testautomatisierungslösungen

ID	Name	Bewertung
AU1	Konsistenz und Wiederholbarkeit der automatisierten Tests	Erfüllt
<p>Begründung</p> <p>Werden in WDIO-WORKFLO Tests mit denselben Ausführungsfiltren und Parametern mehrmals hintereinander ausgeführt, liefern diese in der Regel dieselben Resultate. Die Automatisierung der Tests gewährleistet, dass die vorgesehenen Aktionen immer auf dieselbe Art und Weise und in derselben Reihenfolge von der Maschine ausgeführt werden und das extensive und vollständige Logging der einzelnen Aktionen mit ihren Parametern und Ergebnissen macht die Testausführung transparent und nachvollziehbar. Zusätzlich stellt die Funktion „getUid()“ sicher, dass Testdaten, die eindeutig sein müssen, nicht irrtümlicherweise denselben Wert annehmen, was zu abweichenden Ergebnissen bei mehrmaliger Ausführung führen könnte.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ1 Funktionalität:</i> Konsistenz und Wiederholbarkeit sind essentielle Anforderungen an ein Testframework. • Resultate und Fehler können bei erneuter Testausführung reproduziert werden. 		

ID	Name	Bewertung
AU2	Erhöhung der Testabdeckung	+
<p>Begründung</p> <p>Die Wiederverwendbarkeit und Parametrisierungsmöglichkeit von Steps im Sinne eines Daten- und Keyword-getriebenen Ansatzes führen dazu, dass mit geringem Aufwand Tests mit Testdaten unterschiedlicher Äquivalenzklassen durchgeführt werden können, was eine Erhöhung der Testabdeckung zur Folge hat.</p> <p>Nichtsdestotrotz müssen für jede „Variante“ einer Überprüfung immer noch unterschiedliche Testfälle geschrieben werden. Eine Möglichkeit, für ein und denselben Testfall mehrere Sets an Eingabedaten und erwarteten Resultaten zu definieren, um unnötige Schreiarbeit zu ersparen, wäre wünschenswert, ist jedoch nicht gegeben.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • Erhöhung der Qualität der getesteten Software, da mehr Testszenarien abgedeckt werden können 		

ID	Name	Bewertung
AU3	Vollständige und verständliche Dokumentation	-
<p>Begründung</p> <p>Zwar verbessern klingende, und in natürlicher Sprache formulierte Step-Namen, ausgereifte architektonische Designprinzipien und Typisierung in Kombination mit IntelliSense Features von Visual Studio Code die „Selbsterklärungsfähigkeit“ des Source Codes, allerdings können diese Maßnahme nicht die zum Zeitpunkt der Verfassung dieser Arbeit fehlende, externe Dokumentation von WDIO-WORKFLO kompensieren.</p>		

Nutzen	
<ul style="list-style-type: none"> • <i>SQ4 Benutzbarkeit:</i> Verbesserung der Erlernbarkeit 	

ID	Name	Bewertung
AU4	Senkung des menschlichen Anteils an der Ergebnisanalyse	+
Begründung		
<p>Der Vergleich der erwarteten und der tatsächlichen Ergebnisse von Testfällen geschieht dank den von Jasmine [21] bereitgestellten Assertion Matchers vollautomatisch. Zudem stellt WDIO-WORKFLO im Fehlerfall Screenshots und Stacktraces zur Verfügung. Allerdings ist zur Analyse eines Fehlers weiterhin menschliches Handeln nötig, was jedoch kaum vermeidbar ist.</p>		
Nutzen		
<ul style="list-style-type: none"> • Erhöhte Zuverlässigkeit und verringerter Aufwand für die Analyse der Testergebnisse durch Automatisierung 		

ID	Name	Bewertung
AU5	Kompensation von künftigem Technologie-Rückstand	++
Begründung		
<p>Die Interaktion mit der getesteten Webanwendung findet mittels Selenium Webdriver ausschließlich über den HTML Content einer Webseite statt. Da im Jahr 2017 selbst die neuesten Frontend Webtechnologien auf dem HTML Standard basieren, und auch keine Anzeichen dafür existieren, dass sich an diesem Umstand in absehbarer Zukunft etwas ändert, kann WDIO-WORKFLO als in hohem Maße zukunftsicher bezeichnet werden.</p>		
Nutzen		
<ul style="list-style-type: none"> • Technologische Lebenszeit des Testframeworks wird erhöht • Geringer zukünftiger Änderungsbedarf am Testframework 		

ID	Name	Bewertung
AU6	Erhöhung der Robustheit automatisierter Tests	+
Begründung		
<p>Durch die Typisierung mittels TypeScript werden Syntaxfehler von vornherein vermieden. Der „implicit wait“ Mechanismus stellt sicher, dass die überall gegenwärtigen Wartebedingungen einer Webanwendung berücksichtigt werden und „rescue callbacks“ wie der Parameter „postcondition()“ der „click()“ Funktion sorgen dafür, dass unzuverlässige Aktionen wie ein Klick auf Buttons, welcher in manchen Fällen nicht von der Webanwendung registriert wird, wiederholt aufgerufen werden, bis ein erwarteter Zustand eintritt. Zusätzlich steigert auch die Kapselung von erwiesenermaßen funktionstüchtigem Verhalten in Steps und Page Objects die Robustheit automatisierter Tests. Allerdings ist die erwartungsgemäße Ausführung der Testfälle auch stark von der Ausgereiftheit der Browser-Treiber für Selenium Webdriver abhängig, auf welche TesterInnen im Normalfall keinen Einfluss nehmen können.</p>		
Nutzen		

- *SQ4 Benutzbarkeit:* AnwenderInnen werden durch typensicheren Framework-Code vor Fehlbedienung geschützt
- *SQ5 Zuverlässigkeit:* Erhöhung von Fehlertoleranz und Wiederherstellbarkeit

ID	Name	Bewertung
AU7	Unterstützung von Regressionstests	++
<p>Begründung</p> <p>WDIO-WORKFLO ermöglicht durch die Automatisierung monotoner oder mühsamer, repetitiver Tätigkeiten, des Vergleichs großer Datenmengen mittels Assertion Matchers oder der Durchführung objektiver Messungen von Testmetriken eine ideale Unterstützung bei der Durchführung von Regressionstests. Einmal automatisierte Tests können etwa jede Nacht erneut ausgeführt werden, um zu prüfen, ob Änderungen an der Software neue Fehler in diese eingebracht haben. Manuell durchgeführt wären Regressionstests in einem vergleichbaren Ausmaß nicht möglich.</p> <p>In dem Falle, dass ein kompletter Regressionstest zu aufwendig wäre, bietet WDIO-WORKFLO zudem die Möglichkeit, über Testausführungsfilter des CLI die Ausführung auf bestimmte (Arten von) Testcases und Steps zu beschränken, etwa all jene, die einen bestimmten Schweregrad aufweisen. Zusätzlich können diese Ausführungsfilter auch in Listen gespeichert werden.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • Verringerung der Kosten von Regressionstests durch Automatisierung • Erhöhung der Qualität der getesteten Software durch rasches Feedback bei Änderungen an der Software 		

ID	Name	Bewertung
AU8	Einsatz einer bekannten Skriptsprache für Testskripte	Erfüllt
<p>Begründung</p> <p>TypeScript, welches in WDIO-WORKFLO als Skriptsprache zum Einsatz kommt, liegt im Jahr 2017 bereits auf Platz 17 der populärsten Programmiersprachen weltweit. TypeScript wird in JavaScript kompiliert, was wiederum die weitverbreitetste Programmiersprache im Jahr 2017 darstellt. Nachzulesen ist dies im Ranking der populärsten Programmiersprachen [48] auf der Website der Firma RedMonk.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ4 Benutzbarkeit:</i> Leichtere Erlernbarkeit, da viele EntwicklerInnen bereits mit den weitverbreiteten Sprachen TypeScript und JavaScript vertraut sein dürften 		

2.5.7 Evaluierung des Bereichs Gestaltung von Anforderungen

ID	Name	Bewertung
AN1	Anforderungen müssen niedergeschrieben sein	Erfüllt
Begründung Anforderungen können in WDIO-WORKFLO von Testcases nur dann validiert werden, wenn sie als in Form von Specs schriftlich ausformuliert wurden.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Erhöhung der Testbarkeit, da mündliche überlieferte oder nur im Geiste vorhandene Anforderungen nicht von anderen Personen testbar sind • Jeder Stakeholder hat eine eigene Vorstellung davon, welche Funktionalitäten ein System beherrschen soll. Das Niederschreiben der Anforderungen beugt somit Missverständnissen vor. 		

ID	Name	Bewertung
AN2	Konsistenz innerhalb der Anforderungen	~
Begründung Durch die genaue Beschreibung der Zustände und Zustandsänderungen des Systems in den Specs, sowie dadurch, dass ähnliche Anforderungen physikalisch nahe zueinander definiert und wenn möglich innerhalb desselben Testcases validiert werden, steigt die Chance, Inkonsistenzen innerhalb der Anforderungen zu erkennen.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Erhöhung der Testbarkeit, da inkonsistente Anforderungen kaum testbar sind • Vermeidung von Fehlern infolge inkonsistenter Anforderungen 		

ID	Name	Bewertung
AN3	Verfolgbarkeit der Anforderungen	++
Begründung User Stories oder ähnliche Formen der Anforderungsdefinition sind mit den Specs über deren Metadaten verlinkbar. Durch die „validate()“ Funktion und die „--traceXXX“ Optionen des CLI wird zudem eine Verknüpfung zwischen Testcases und Specs hergestellt. Sämtliche Traceability Informationen werden darüber hinaus in den Details eines Testcases oder Specs im Allure Report ausgegeben.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Erhöhung der Analysierbarkeit • Verschiedene Formen der Anforderungsdefinition können aufeinander verweisen. • Bei Fehlschlag (einzelner Validierungen) eines Testcases kann rasch Auskunft gegeben werden, welche Anforderung davon betroffen ist. • Verringerter Aufwand bei Rückverfolgung von Anforderungsinformationen 		

ID	Name	Bewertung
AN4	Eindeutigkeit der Anforderungen	++
Begründung Durch die Formalisierung der Specs mittels Gherkin Language, welche eine boolesche Überprüfbarkeit der Anforderungen anhand von Systemzuständen erlaubt und die einzelnen Zustandsübergänge detailliert beschreibt, können Mehrdeutigkeiten auf ein Minimum reduziert werden.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit</i>: Erhöhung der Testbarkeit, da mehrdeutige Anforderungen nicht testbar sind 		

ID	Name	Bewertung
AN5	Semantische Korrektheit der Anforderungen	--
Begründung Das Framework selbst kann zur Überprüfung der semantischen Korrektheit der Anforderungen keinen Beitrag leisten. Stattdessen müssen mit dem Fachbereich vertraute Stakeholder die Specs gegenlesen oder gleich selbst verfassen.		
Nutzen <ul style="list-style-type: none"> • Vermeidung von durch inkorrekte Anforderungen verursachte Fehlerkosten 		

ID	Name	Bewertung
AN6	Vollständigkeit der Anforderungen	~
Begründung Die verschachtelte Struktur, in welcher Specs formuliert werden, hilft dabei, fehlende Handlungsstränge (siehe Kapitel 2.4.1) leichter zu erkennen, als in Anforderungen, welche in freier Prosa ausgedrückt wurden. Jedoch kann WDIO-WORKFLO die Vollständigkeit der Anforderungen nicht garantieren.		
Nutzen <ul style="list-style-type: none"> • Vermeidung von durch unvollständige Anforderungen verursachte Fehlerkosten 		

ID	Name	Bewertung
AN7	Gültigkeit der Anforderungen	--
Begründung Die Gültigkeit der Anforderungen muss durch Stakeholder mit Fachbereichskenntnissen und kann nicht durch das Testframework selbst beurteilt werden.		
Nutzen <ul style="list-style-type: none"> • Vermeidung von durch ungültige Anforderungen verursachte Fehlerkosten 		

ID	Name	Bewertung
AN8	Verständlichkeit der Anforderungen	++
<p>Begründung</p> <p>Sowohl Dustin et al. [46] als auch Loucopoulos und Karakostas [19, pp. 77-78, 83] betrachten die Verständlichkeit für alle beteiligten Stakeholder als größte Herausforderung bei der Formulierung von Anforderungen. Beide empfehlen, die Anforderungen in einer Mischung aus natürlicher Sprache und logischen Regeln zu paraphrasieren. Dadurch sollen sowohl AnalystInnen und TesterInnen bei der Entwicklung eines formalen Anforderungsmodells unterstützt werden, als auch die Anforderungen gleichzeitig für alle anderen Stakeholder verständlich gehalten werden. WDIO-WORKFLO setzt diese Idee in Form der Specs um. Im Detail erlauben die Schlüsselworte der Gherkin Language, ein Akzeptanzkriterium, sowie dessen Vorbedingungen und alle Zustandsübergänge zum Erreichen des finalen Zustandes in einer logisch überprüfbaren Weise zu formulieren, während die in natürlicher Sprache beschriebenen Inhalte der Schritte gleichzeitig vom gesamten Team verstanden werden können. Zudem trägt auch die verschachtelte Bauweise der Specs dazu bei, die Struktur der Handlungsstränge einer Anforderung deutlich zu vermitteln und erhöht damit ebenfalls die Verständlichkeit.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ4 Benutzbarkeit:</i> Bessere Bedienbarkeit in allen Bereichen, die auf Anforderungen basieren • <i>SQ7 Wartbarkeit:</i> Erhöhung der Analysierbarkeit • Aufwandsreduktion infolge verringerter Klarstellungen von Anforderungen • Verringerung der durch falsch verstandene Anforderungen verursachten Fehler 		

ID	Name	Bewertung
AN9	Bewertbarkeit und Gewichtbarkeit der Anforderungen	Erfüllt
<p>Begründung</p> <p>Durch Angabe der „Severity“ in den Spec Metadaten können die Anforderungen gewichtet werden.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • Anhand der Gewichtung können die Auswirkungen nicht erfüllter Anforderungen auf das System eingeschätzt und die Behebung von Fehlern priorisiert werden. 		

ID	Name	Bewertung
AN10	Prüfbarkeit der Anforderungen	++
<p>Begründung</p> <p>Die Formulierung der Specs, welche natürliche Sprache mit logischen Regeln erweitert, um boolesche Aussagen über die Erfüllung einer Anforderung zuzulassen, führt zu einer idealen Prüfbarkeit der Anforderungen.</p>		
<p>Nutzen</p>		

- *SQ7 Wartbarkeit*: Gewährleistung der Testbarkeit

ID	Name	Bewertung
AN11	Umsetzbarkeit der Anforderungen	~
Begründung Der in den Specs formulierte, „mentale Prototyp“ der GUI kann Probleme in der Umsetzbarkeit aufzeigen. Allerdings ist eine zusätzliche Einschätzung der technischen Umsetzbarkeit durch das Entwicklungsteam in jedem Fall erforderlich.		
Nutzen <ul style="list-style-type: none"> • Vermeidung von Kosten infolge von nicht umsetzbaren Anforderungen 		

ID	Name	Bewertung
AN12	Minimalität der Anforderungen	+
Begründung Die Beschreibung der Schritte logischer Testfälle innerhalb der Specs kann auf beliebigen abstrakten Ebenen stattfinden und eine Überspezifikation dadurch vermieden werden. Wie Kapitel 2.4.1 näher erläutert, führt die verschachtelte Form der Specs zudem zu erheblichen Einsparungen des für die Anforderungsbeschreibung benötigten Textes.		
Nutzen <ul style="list-style-type: none"> • <i>SQ2 Effizienz</i>: Reduzierter Aufwand für die Formulierung von Specs • EntwicklerInnen werden in ihrer technischen Kreativität nicht unnötig eingeschränkt 		

ID	Name	Bewertung
AN13	Vermeidung von Redundanzen in den Anforderungen	~
Begründung Dadurch, dass ähnliche Anforderungen innerhalb von Specs in physikalischer Nähe zueinander definiert werden und da diese meist innerhalb derselben Testcases validiert werden, können Redundanzen in den Anforderungen leichter entdeckt werden. Ein weiteres Indiz stellen Akzeptanzkriterien mit einem Status von „unvalidated“ dar. Im Endeffekt obliegt es jedoch der Beobachtungsgabe der TestautomatisiererInnen, und nicht dem Testframework selbst, Redundanzen in den Anforderungen aufzudecken.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit</i>: Geringere Kosten für Analysierbarkeit und Änderbarkeit, da redundante Anforderungen nicht mehr berücksichtigt werden müssen 		

ID	Name	Bewertung
AN14	Anforderungsmodell soll Vorstellung von Zuständen und Zustandsübergängen ermöglichen	++
Begründung In den Specs von WDIO-WORKFLO drücken die Keywords „When“ und „And“ Zustandsübergänge und das Keyword „Then“ Zustände des Systems aus.		
Nutzen		

- *SQ7 Wartbarkeit:* Steigerung der Analysierbarkeit
- Schaffung eines „mentalen“ Prototyps zur Überprüfung von Anforderungen

ID	Name	Bewertung
AN15	Auf implizierte Anforderungsinformationen soll rückgeschlossen werden können	-
<p>Begründung</p> <p>In Testcases ist das Schließen auf implizierte Informationen innerhalb von Steps relativ einfach möglich, da Steps sich aus beliebigen anderen Steps zusammensetzen können und dies auch durch den Source Code abgebildet und mittels „static checking“ geprüft werden kann.</p> <p>Specs hingegen können im Grunde frei formuliert werden, solange sie sich an die Gherkin-Grundstruktur halten. Eine technische Referenzierung auf konkretere Teilschritte innerhalb von abstrakten Schritten ist bei Specs nicht möglich und es hängt von der Auffassungsgabe der LeserInnen von Specs und deren Formulierung ab, ob Rückschlüsse auf implizierte Anforderungsinformationen möglich sind.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • Reduzierter Aufwand für Formulierung von Anforderungen innerhalb der Specs • Gesteigerte Lesbarkeit, da die Beschreibung jedes kleinsten Details durch eigene Steps oftmals nicht nötig ist und diese zu abstrakten Steps zusammengefasst werden können 		

2.5.8 Evaluierung des Bereichs Traceability

ID	Name	Bewertung
TR1	Traceability zwischen Anforderungen unterschiedlicher Ebenen	Erfüllt
Begründung Durch Links in den Metadaten der Specs kann Traceability zwischen Specs und anderen Ebenen der Anforderungsdefinition, etwa User Stories in JIRA, hergestellt werden.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Verbesserung der Analysierbarkeit und Änderbarkeit 		

ID	Name	Bewertung
TR2	Eindeutige Identifikation von Artefakten muss möglich sein	Erfüllt
Begründung Specs sind über die Story ID eindeutig identifizierbar. Zur eindeutigen Identifikation von Testcases werden deren Namen, sowie die Namen aller umschließenden Suites, verkettet.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Verbesserung der Analysierbarkeit und Änderbarkeit 		

ID	Name	Bewertung
TR3	Das Ziel eines Verweises soll ersichtlich sein	+
Begründung Die „validate()“ Funktion führt in ihrem ersten Parameter die Story IDs und die IDs der Akzeptanzkriterien auf, auf welche der jeweilige Testcase verlinkt. In den Metadaten von Specs und Testcases sind die Arten eines Verweises durch die Namen der Properties („issue“ oder „bug“) ersichtlich und die verlinkte Issue- oder Bug-ID ist jene, die auch im verknüpften Issue oder Incident Management System verwendet wird. Jedoch werden die Namen/Titel der verlinkten Issues und Bugs in WDIO-WORKFLO nicht angezeigt.		
Nutzen <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Verbesserung der Analysierbarkeit 		

ID	Name	Bewertung
TR4	Der Aufwand für Traceability soll gering gehalten werden	++
Begründung Der Aufwand für die Sicherstellung von Traceability fällt in WDIO-WORKFLO sehr gering aus. So ist die Verlinkung zwischen Specs und Testcases ohnehin Grundvoraussetzung dafür, dass die Specs überhaupt validiert werden können und die Traceability zwischen Specs und Testcases erfordert daher keinen zusätzlichen Aufwand. Zuordnungen zwischen Specs und Features, Testcases und Suites, sowie den Dateien, in welchen diese definiert sind, ermittelt WDIO-WORKFLO zudem automatisch. Einzig Verknüpfungen zu externen Tools in Form von Issues und Bugs stellen einen Mehraufwand dar, wobei die Links zu Issues im Regelfall nur einmalig definiert werden		

müssen und daher vernachlässigbar sind. Ein aktives Verlinken von Bugs mit Specs und Testcases erfordert bei konsequenter Durchführung jedoch einen erheblichen Mehraufwand – ob diese Maßnahme ergriffen wird, liegt jedoch im Ermessen der TesterInnen und kann dem Testframework nicht angelastet werden.

Nutzen

- *SQ2 Effizienz:* Es werden weniger Ressourcen für die Herstellung von Traceability benötigt

ID	Name	Bewertung
TR5	Traceability zwischen Anforderungen und Testfällen	Erfüllt
<p>Begründung</p> <p>Die Traceability zwischen Anforderungen und Testfällen wird durch die „validate()“ Funktion innerhalb von Testcases gewährleistet, welche in ihrem ersten Parameter angibt, welche Specs von den im Body der Funktion definierten Assertion Matchers validiert werden. Diese Verknüpfung ist im Code selbst nur unidirektional vorhanden. Das Framework ermittelt jedoch automatisch auch die Traceability Informationen, welche in Richtung der Specs zu den Testcases, die diese validieren, führen, und zeigt diese in den Allure Reports an.</p> <p>Weiters sind die in den Reports ausgegebenen Stacktraces nützliche Informationsquellen für Traceability, denn sie zeigen bei nicht erfüllten Akzeptanzkriterien innerhalb der Specs an, in welcher Codezeile eines Testcases die jeweilige Validierung fehlgeschlagen ist.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Verbesserung der Analysierbarkeit und Änderbarkeit • Bei Fehlschlag eines Testcases können sofort Aussagen darüber getroffen werden, welche Specs dadurch nicht validiert werden konnten. • Genauso kann bei nicht erfüllten Akzeptanzkriterien in den Specs leicht ausgemacht werden, in welchem Testcase und an welcher Stelle innerhalb dieses Testcases das betroffene Akzeptanzkriterium validiert wurde. 		

ID	Name	Bewertung
TR6	Traceability zwischen Testfällen und Defects	Erfüllt
<p>Begründung</p> <p>Traceability Links zwischen Testcases und Defects können in den Metadaten der Testcases definiert werden.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ7 Wartbarkeit:</i> Verbesserung der Analysierbarkeit und Änderbarkeit 		

2.5.9 Evaluierung des Bereichs Reports

ID	Name	Bewertung
RE1	Sammeln von Maßen zur Teststeuerung	+
<p>Begründung</p> <p>WDIO-WORKFLO erhebt die Dauer von Testcases und einzelner Steps, den Abdeckungsgrad automatisierter/manueller/nicht validierter Akzeptanzkriterien, den Status von Testcases und Specs, sowie den Schweregrad von Specs und Testcases, um eine Grundlage für die grundlegendsten Entscheidungen des Testmanagements zu bieten. AnwenderInnen des Testframeworks können jedoch keine eigenen, ebenfalls zu erhebende Metriken definieren.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ2 Effizienz:</i> Keine menschlichen Ressourcen für Sammeln von Maßen zur Teststeuerung erforderlich (beispielsweise keine manuellen Zeitmessungen nötig) • Unterstützung des Testmanagements durch Bereitstellung von Entscheidungsgrundlagen 		

ID	Name	Bewertung
RE2	Unterstützung der Fehleranalyse sowie der Festlegung von Schwere und Priorität der Behebung von Fehlern	++
<p>Begründung</p> <p>Zur Unterstützung der Analyse von Fehlern erstellt WDIO-WORKFLO automatisch Screenshots von der GUI zum Fehlerzeitpunkt und fügt den Reports die Stacktraces der Fehler hinzu. Zudem sind alle Steps, die bis zum Eintreten eines Fehlers durchlaufen werden, mitsamt deren Parametern und Ergebnissen, in den Allure Reports einsehbar, wodurch die Vorbedingungen eines Fehlers genau untersucht werden können. Auch die Möglichkeit, durch den Klick auf eine Zeile eines Stack Traces im Spec Report direkt zur betroffenen Stelle im Testcase zu springen, unterstützt die Fehleranalyse erheblich.</p> <p>Um die Priorität der Behebung von Fehlern festzulegen, können Specs und Testcases in deren Metadaten mit Schweregraden bewertet und Links zu den zugrunde liegenden Issues (etwa User Stories in JIRA) definiert werden.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ2 Effizienz:</i> Schnellere Fehleranalyse • Unterstützung agiler Entwicklungsmethoden, bei denen die höchstpriorisierten Issues zuerst bearbeitet werden • <i>SQ7 Wartbarkeit:</i> Bessere Analysierbarkeit, da von Fehlern betroffene Anforderungen direkt im Testreport ersichtlich sind und nicht erst händisch ermittelt werden müssen 		

ID	Name	Bewertung
RE3	Möglichkeit zur Erstellung kompletter Testdokumentation	+
<p>Begründung</p> <p>Eine komplette Testdokumentation umfasst Testplan, Testspezifikation und Testabschlussbericht. Die Testspezifikation wird durch die Darstellung von Namen und Metadaten der Specs und Testcases, sowie deren Steps mitsamt Parametern und Resultaten in den Allure Reports ausführlich abgebildet. Zum Testabschlussbericht tragen die Diagramme des Allure Reports, welche einen schnellen Überblick über die Ergebnisse eines Testlaufs verschaffen, bei. Zudem können die Resultate auf den Ebenen von Features und Suites, Testcases und Specs, und sogar einzelnen Steps detailliert „navigiert“ und betrachtet werden. Einzig zum Testplan liefert WDIO-WORKFLO keine Informationen.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ2 Effizienz:</i> Reduktion menschlicher Ressourcen für Erstellung von Testdokumentation • Reduzierter Aufwand, da Testdokumentation nicht separat erstellt werden muss • Nachvollziehbarkeit und Auskunft über Aktivitäten und Ergebnisse des Testprozesses für alle Stakeholder 		

ID	Name	Bewertung
RE4	Exaktes und vollständiges Logging der Testausführung	+
<p>Begründung</p> <p>Die Hauptaufgabe des exakten und vollständigen Loggings der Testausführung ist eine lückenlose Nachvollziehbarkeit und Reproduzierbarkeit der einzelnen Testfälle. Diese erfüllt WDIO-WORKFLO vorbildlich, indem die Abfolge der einzelnen Steps mit deren Parametern, Resultaten und Status vollständig erfasst und zusätzlich die Dauer der Testcases und Steps aufgezeichnet wird. Darüber hinaus werden im Fehlerfall Screenshots und Stacktraces zu den fehlerhaften Steps im Allure Report hinzugefügt. Jedoch bietet WDIO-WORKFLO keine Möglichkeit, im Falle einer manuellen Auslösung eines Testlaufs den Namen der Person anzugeben, die diesen gestartet hat. Nur im Rahmen eines Continuous Integration Prozesses, bei dem die Testausführung etwa durch Jenkins Builds angestoßen wird, verlinkt der Allure Report auf diese. Ebenso kann auf die betesteten Teile des Systems nur indirekt durch Angabe der ausgeführten Testcases und validierten Specs rückgeschlossen werden. Auch die getestete Konfiguration kann nur indirekt ermittelt werden, indem diese Informationen aus dem verlinkten Buildvorgang gewonnen werden. Die Testumgebung wird in WDIO-WORKFLO durch den verwendeten Browser und die URL der gehosteten Webanwendung angegeben, weiterführende Informationen, wie etwa das verwendete Betriebssystem, fehlen jedoch.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • Exakte Reproduzierbarkeit und Nachvollziehbarkeit eines (fehlerhaften) Testfalls 		

2.5.10 Evaluierung des Bereichs Framework Design

ID	Name	Bewertung
FD1	Förderung von Wiederverwendbarkeit	++
<p>Begründung</p> <p>WDIO-WORKFLO wurde aus der Idee heraus geboren, die Kosten für die Entwicklung automatisierter Testfälle von funktionalen Systemtests durch Wiederverwendung von Komponenten zu senken. Dementsprechend hoch ist auch die Förderung von Wiederverwendbarkeit durch das Framework - vor allem dank dem Einsatz von Steps und Page Objects. Wurden diese einmal implementiert, können sie in beliebig vielen Testcases verwendet werden.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ4 Benutzbarkeit:</i> Bessere Erlernbarkeit, da wiederverwendete Steps und Page Objects immer gleich funktionieren • <i>SQ7 Wartbarkeit:</i> Erhöhung von Wiederverwendbarkeit • Sehr großes Einsparungspotenzial bei den Entwicklungskosten von Tests 		

ID	Name	Bewertung
FD2	Anwendung von Entwurfsmustern	~
<p>Begründung</p> <p>Zwar kommen in WDIO-WORKFLO einige Entwurfsmuster zur Anwendung, meist jedoch treten diese situationsbedingt in etwas abgewandelter Form auf, was die Verständlichkeit und Wiedererkennung verschlechtert und zu einer Abwertung dieser Anforderung führt. Das einzige Entwurfsmuster, dessen Anwendung in WDIO-WORKFLO exakt dem ursprünglichen Vorbild entspricht, ist das in Kapitel 2.4.5.1 behandelte Page Object Pattern [39]. Die PageElementStores hingegen stellen eine Mischung aus Factory Methods [44] und dem Singleton Pattern dar, wie in Kapitel 2.4.5.7 näher erläutert wird. Pages wiederum können, wie in Kapitel 2.4.5.2 beschrieben, zum Teil als Anwendung des Facade Patterns [41] gesehen werden. Schließlich stellen die in Kapitel 2.4.5.6 betrachteten PageElementGroups eine Form des Composite Design Patterns [43] dar, wobei die Traversierung nicht durch das Composite selbst, sondern durch den PageElementGroupWalker vorgenommen wird.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • <i>SQ2 Effizienz:</i> Die verwendeten Entwurfsmuster konnten sich bereits über lange Zeit als effiziente Lösungen für eine bestimmte Problemstellung bewähren. • <i>SQ4 Benutzbarkeit:</i> Verbesserte Erlernbarkeit und Bedienbarkeit 		

ID	Name	Bewertung
FD3	Inversion of Control	++
<p>Begründung</p> <p>Im Sinne von „Inversion of Control“ können Frameworks zwar erweitert werden, um den Bedürfnissen einer bestimmten Anwendung zu entsprechen. Dabei werden jedoch keine</p>		

Änderungen am Framework Code selbst vorgenommen und die Kontrolle über Aufbau und Ablauf eines Programms obliegen dem Framework, und nicht dem applikationsspezifischen Code.

In WDIO-WORKFLO kann dieses Prinzip in zwei großen Bereichen beobachtet werden:

Der erste Bereich betrifft das Zusammenspiel von Specs, Testcases und Steps. Für diese drei Komponenten stellt WDIO-WORKFLO eine Art Skelett zur Verfügung, das die grundlegende Struktur und die (Abfolge der) Interaktionen zwischen den Komponenten definiert. So ruft WDIO-WORKFLO die einzelnen Testcases auf, verknüpft deren Validierungsergebnisse mit den Specs, und kümmert sich ebenso um die Ausführung und das Logging der einzelnen Steps. Die AnwenderInnen von WDIO-WORKFLO füllen dagegen dieses Grundgerüst mit applikationsspezifischem Inhalt, ohne dabei jedoch auf die zugrundeliegende Basisfunktionalität Einfluss nehmen zu können oder zu müssen.

Besonders augenscheinlich ist dies am Beispiel der Callbacks von Steps. Während das Framework den Step mit den übergebenen Parametern aufruft, das Resultat des Steps an das Callback übergibt und damit die Abfolge der einzelnen Phasen eines Steps kontrolliert, können die AnwenderInnen im Callback applikationsspezifische Logik bereitstellen, welche stets Zugriff auf das Resultat des Steps hat und immer im Anschluss an diesen ausgeführt wird.

Der zweite Bereich bezieht sich auf die Page Object Basisklassen (PageElement, PageElementList, PageElementMap, PageElementGroup, PageElementGroupWalker und PageElementStore), welche vom Testframework zur Verfügung gestellt werden und bereits alle grundlegenden internen Funktionalitäten und Formen der Zusammenarbeit zwischen diesen Klassen bereitstellen. Dazu zählen etwa das Caching von PageNode Instanzen im PageElementStore oder die Verwaltung einer Sammlung von PageElements durch PageElementLists und PageElementMaps. AnwenderInnen von WDIO-WORKFLO schreiben Page Object Subklassen, welche von den Basisklassen erben und diese um applikationsspezifische Funktionalitäten erweitern, während die grundlegenden Funktionalitäten weiterhin durch das Framework selbst verwaltet werden.

Beispielsweise könnte für ein Online-Forum ein applikationsspezifischer Store, welcher von PageElementStore ableitet, erzeugt werden. Dieser „ForumStore“ könnte eine FactoryMethod zum Erzeugen des PageElements „ForumPost“ über seine API anbieten. Während die applikationsspezifische Klasse „ForumPost“ die Vergabe der Default-Typen und -Parameter beim Erzeugen von „ForumPost“ festlegen würde, wäre das Caching aller Instanzen von „ForumPost“ und sämtlicher anderer PageNode Instanzen jedoch weiterhin unter der Kontrolle von PageElementStore selbst.

Schließlich stellt auch die „click()“ Funktion von PageElement ein äußerst deutliches Anwendungsbeispiel von „Inversion of Control“ dar: Während AnwenderInnen im „postCondition()“ Parameter eine applikationsspezifische Bedingung angeben, welche

erfüllt sein muss, damit ein Klick als erfolgreich ausgeführt anerkannt wird, wertet das Framework die Erfüllung dieser Bedingung in regelmäßigen Abständen aus, ohne dafür über applikationsspezifisches Wissen verfügen zu müssen.

Nutzen

- *SQ4 Benutzbarkeit:* Möglichkeiten der Fehlbedienung reduziert, da Hauptlogik allein vom Framework übernommen wird
- *SQ7 Wartbarkeit:* Verbesserte Änderbarkeit, da AnwenderInnen keine Fehler in die Hauptlogik des Testframeworks einbringen können

ID	Name	Bewertung
FD4	Abstraktion und Separation of Concerns	++
<p>Begründung</p> <p>Abstraktion erlaubt, Probleme auf verschiedenen Ebenen zu betrachten und dadurch deren Komplexität zu reduzieren. Pressman und Maxim [6, p. 232] unterscheiden zwischen prozeduraler Abstraktion und der Abstraktion von Daten.</p> <p>WDIO-WORKFLO fördert vor allem die Anwendung prozeduraler Abstraktion, indem Steps in beliebigen Abstraktionsebenen definiert werden können. Für den Login-Vorgang im System könnte ein Step auf niederer, technischer Abstraktionsebene etwa „Klicke den Login Button“ lauten, während „Login als Admin User“ einen Step auf hoher, domänenspezifischer Abstraktionsebene darstellt.</p> <p>Die Abstraktion von Daten hingegen obliegt zum Großteil den AnwenderInnen von WDIO-WORKFLO, denn das Framework selbst verwaltet kaum Daten, da der Zustand des Systems in der GUI abgebildet ist und nicht im Framework selbst gespeichert wird.</p> <p>Im Allgemeinen bemüht sich WDIO-WORKFLO, die Entwicklung von Testfällen zu möglichst großen Teilen in einer domänenspezifischen, hohen Abstraktionsebene zu gestalten. Dies wird dadurch erreicht, dass die „low-level“ Kommunikation mit Selenium Webdriver über die von WebdriverIO bereitgestellten NodeJS Bindings innerhalb der Klassen PageElement, PageElementList und PageElementMap, sowie deren Kindern, gekapselt wird. Interaktionen mit der getesteten Webseite finden somit über die applikationsspezifische API der Page Objects auf einer hohen Abstraktionsebene statt, was den Test-Code wesentlich leichter verständlich macht.</p> <p>Als weiteres Mittel zur Reduktion von Komplexität setzt WDIO-WORKFLO „Seperation of Concerns“ ein. Wie man anhand der Skizze der Systemumgebung in Abbildung 15 und der zugehörigen Beschreibung in Kapitel 2.3.2 erkennen kann, übernehmen der Core von WDIO-WORKFLO sowie dessen externe Dependencies alle klar voneinander abgetrennte Aufgaben, die jeweils einen Teilbereich des komplexen Prozesses automatisierter funktionaler Systemtests abdecken. So sorgt der Core für das Zusammenspiel von Specs, Testcases und Steps auf der einen, sowie der unterschiedlichen Klassen von Page Objects auf der anderen Seite. WebdriverIO fungiert als synchroner Testrunner, der mithilfe seiner Bindings in NodeJS geschriebene Befehle</p>		

an Selenium Webdriver übermittelt. Selenium Webdriver wiederum ermöglicht eine geskripte Interaktionen mit diversen Browsern. Jasmine stellt Assertion Matchers zur Verfügung und Allure Reporter sowie Spec Reporter kümmern sich um das Logging des Testvorgangs sowie um die Präsentation der Resultate.

Auch innerhalb des Cores wendet WDIO-WORKFLO „Seperation of Concerns“ an, wie sich anhand des Klassendiagramms in Abbildung 18 (Kapitel 2.3.3) beobachten lässt. Dieser setzt sich demgemäß aus den Packages Specs, ManualResults, Testcases, Steps und Page Objects zusammen, welche sich wiederum in mehrere Klasse zerteilen, deren jeweilige Aufgabe in den einzelnen Unterkapiteln des Kapitels 2.4 näher erläutert wird. Allen gemein ist der Umstand, dass sie sich auf die Lösung eines ganz speziellen Teilproblems konzentrieren.

Eine „Konsequenz“ der im Rahmen von „Seperation of Concerns“ geschaffenen Modularität stellt das Prinzip „Information Hiding“ dar, bei dem Interaktionen einzelner Komponenten mit der Außenwelt über eine Schnittstelle stattfinden. Dadurch werden die Kenntnisse, die zur Anwendung einer Komponente von außen nötig sind, auf das Notwendigste reduziert. In WDIO-WORKFLO ist „Information Hiding“ allgegenwärtig.

Am augenscheinlichsten tritt das Prinzip jedoch im Rahmen der Page Objects zu Tage, welche die Interaktionen mit der getesteten Webseite nach außen hin kapseln. So könnte etwa die Auswahl einer Option in einem Dropdown-Menü durch die API-Funktion „chooseOption()“ beschrieben werden, welche als Übergabeparameter einzig den Wert der gewünschten Option benötigt. Alles Wissen über die intern erforderlichen Abläufe zur Selektion der Option (Klick auf Dropdown, um Menü zu öffnen; Identifizierung der Option mit dem gewünschten Wert über ein spezielles HTML-Attribut; Klick auf die besagte Option) werden vor der Außenwelt verborgen.

Den AnwenderInnen von WDIO-WORKFLO dürfte darüber hinaus gar nicht bewusst sein, wie viele interne Vorgänge durch Aufruf der API-Funktionen „Story()“, „Testcase()“ oder „step[xxx]()“ ausgelöst werden, denn sämtliche Aktivitäten des Testrunners und der Testreporter werden durch diese „verursacht“. In dieser Hinsicht wurde „Information Hiding“ in WDIO-WORKFLO besonders gründlich umgesetzt.

Insgesamt sorgt „Seperation of Concerns“ für eine erhöhte Wiederverwendbarkeit der Module, da sie zur Lösung voneinander unabhängiger Teilprobleme herangezogen werden können. In Kombination mit der Verwendung von Schnittstellen aufgrund des „Information Hiding“ Prinzips steigt zudem die Analysierbarkeit, da keine Kenntnisse über den inneren Aufbau und die internen Abläufe eines Moduls erforderlich sind. Solange die Schnittstellen gleichbleiben, kann die Funktionsweise von Modulen zudem geändert werden, ohne dass dadurch Komponenten, die diese Module verwenden, mitangepasst werden müssen.

Nutzen

- *SQ4 Benutzbarkeit:* Teilprobleme können von Menschen leichter verstanden werden und sind damit auch leichter erlernbar.
- *SQ7 Wartbarkeit:* Verbesserte Modularität, Wiederverwendbarkeit, Analysierbarkeit und Änderbarkeit

ID	Name	Bewertung
FD5	Funktionale Unabhängigkeit	++
<p>Begründung</p> <p>Da die Evaluierung der Anforderung zum Konzept der funktionalen Unabhängigkeit in WDIO-WORKFLO vergleichsweise umfangreich ist, erfolgt diese in mehreren Phasen: Zunächst werden Kohäsion und Kopplung im Allgemeinen anhand des Klassendiagramms von WDIO-WORKFLO betrachtet. Im Anschluss wird die Umsetzung der einzelnen Ausprägungsarten von Kohäsion und Kopplung beurteilt und zum Schluss noch die Einhaltung von „Open/Closed Principle“, „Interface Segregation Principle“ und „Dependency Inversion“ als Vertreter der SOLID Prinzipien anhand von Beispielen überprüft.</p> <p>Anhand des Klassendiagramms von WDIO-WORKFLO in Abbildung 18 lassen sich Kohäsion und Kopplung durch Zählen der Verbindungen von Klassen zwischen und innerhalb von Modulen abschätzen.</p> <p>Im Idealfall ist die Zahl der Verbindungen zwischen Klassen unterschiedlicher Module möglichst gering, denn dies bedeutet eine niedrige Kopplung. Außerdem sollten Verbindungen bestenfalls auch nur in eine Richtung erfolgen. In WDIO-WORKFLO sind diese beiden Voraussetzungen für niedrige Kopplung ideal erfüllt, denn zwischen den fünf Packages Specs, ManualResults, Testcases, Steps und PageObjects existieren lediglich fünf und damit die Mindestanzahl an Verbindungen, wobei sämtliche dieser Verbindungen unidirektional sind. Auch innerhalb von Packages sollten möglichst viele Verbindungen nur in eine Richtung erfolgen, und mit Ausnahme der Verbindungen „Feature-Story“ und „Suite-Testcase“ ist dies immer der Fall.</p> <p>Eine hohe Kohäsion wiederum ist dann erreicht, wenn die Anzahl an Verbindungen zwischen Klassen innerhalb eines Packages möglichst hoch im Vergleich zur Anzahl an „Package-externen“ Verbindungen ist. Dies deutet nämlich darauf hin, dass alle Klassen eines Packages auf die Erreichung eines gemeinsamen Ziels hinarbeiten. In WDIO-WORKFLO ist dies vor allem im PageObjects Package deutlich erkennbar. Während nur eine externe Verbindung (Page-Steps) existiert, gibt es zwischen den Klassen innerhalb des Packages eine Vielzahl an Verbindungen.</p> <p>Pressman und Maxim [6, pp. 296-298] unterscheiden mit „Layer Cohesion“, „Functional Cohesion“ und „Communicational Cohesion“, sowie „Content Coupling“, „Control Coupling“ und „External Coupling“, jeweils drei Arten von Kohäsion und Kopplung.</p>		

„Layer Cohesion“ besagt, dass höhere abstrakte Ebenen die Dienste von niedrigeren abstrakten Ebenen verwenden dürfen, aber nicht umgekehrt. In WDIO-WORKFLO zeigt sich dies beispielsweise anhand von Steps und Page Objects. Steps stellen Klassen einer hohen Abstraktionsebene dar, die in domänenspezifischer Sprache beschrieben werden. Page Objects hingegen kommunizieren über die WebDriverIO Bindings direkt mit Selenium WebDriver und gehören somit einer niedrigen, technik-nahen Abstraktionsebene an. Steps greifen auf die API Funktionen der Page Objects zu, während Page Objects keine Kenntnisse von Steps haben.

„Communicational Cohesion“ äußert sich in WDIO-WORKFLO etwa darin, dass so gut wie alle Operationen, die den Zugriff auf Daten, welche den Zustand der getesteten Webseite beschreiben, ermöglichen, innerhalb der Klasse PageElement definiert sind.

Bei der „Functional Cohesion“ konzentriert sich ein gesamtes Modul auf die Berechnung des Ergebnisses einer ganz bestimmten Funktion oder Operation. Die Klasse PageElementGroupWalker stellt in WDIO-WROFKLO ein Beispiel für „Functional Cohesion“ dar, denn sie beschäftigt sich allein damit, den aus beliebig verschachtelten PageNodes bestehenden Content von PageElementGroups zu traversieren.

Auf Seite der Kopplung ist „Content Coupling“ in WDIO-WORKFLO so gut wie nicht vorhanden, da einerseits die meisten Klassen zustandslos sind und daher auch keine internen Daten verwalten. Andererseits verfügen jene Klassen, die nicht zustandslos sind, über eine klar definierte Schnittstelle nach außen, welche die einzige Möglichkeit darstellt, den Zustand der internen Daten zu ändern, denn alle internen Properties sind mit den Visibility-Werten „private“ oder „protected“ versehen.

„Control Coupling“ tritt in WDIO-WORKFLO in geringem Maße auf. Zum größten Teil kann es jedoch verhindert werden, indem anstelle von Flags, die einer Funktion übergeben werden, um den Kontrollfluss innerhalb der Funktion umzuleiten, gleich zwei unterschiedliche Funktionen für jeden der beiden Zweige im Kontrollflussgraph definiert werden. Beispiele hierfür stellen etwa die beiden Funktionen „waitVisible()“ sowie „waitHidden()“ in der Klasse PageElement dar. Die Funktion „waitHidden()“ hätte im Falle von „Control Coupling“ auch über den Aufruf der Funktion „waitVisible()“ mit einem „reverse“ Flag umgesetzt werden können.

Zur Vermeidung von „External Coupling“ wird die Kommunikation mit „low-level“ Infrastrukturkomponenten in WDIO-WORKFLO auf jeweils eine oder wenige Klassen reduziert. So finden beispielsweise sämtliche Zugriffe auf die NodeJS Bindings für Selenium WebDriver innerhalb der Klassen PageElement, PageElementList und PageElementMap statt.

Schließlich wird nun die Einhaltung ausgewählter SOLID Prinzipien beurteilt, welche positive Auswirkungen auf die funktionale Unabhängigkeit haben:

Im Sinne des „Open/Closed“ Prinzips etwa kann die Funktionalität der Page Object Basisklassen durch AnwenderInnen von WDIO-WORKFLO nicht verändert werden, allerdings ist diese durch abgeleitete Subklassen erweiterbar. Auch bei den Callbacks von Steps tritt dieses Prinzip deutlich zu Tage: Während der Code eines einmal definierten Steps nicht zur ständigen Veränderung vorgesehen ist, kann im Callback im Anschluss an die Ausführung des Steps beliebig oft unterschiedlichster Code zur Reaktion auf das Step Ergebnis verwendet werden.

Das „Interface Segregation Principle“ wiederum zeigt sich in WDIO-WORKFLO etwa anhand der von spezifischen PageElementGroups verwendeten Interfaces, die die Fähigkeiten eines PageNodes beschreiben, welche von der jeweiligen Gruppe zur Lösung eines bestimmten Problems benötigt werden. Diese können von konkreten PageNode Klassen nach Belieben kombiniert werden. So könnte etwa die von PageElement abgeleitete Klasse „Dropdown“ sowohl das Interface der „ValueGroup“ mit den Methoden „setValue()“ und „getValue()“ als auch jenes der „TextGroup“ mit der Methode „getText()“ implementieren.

„Dependency Inversion“ besagt schließlich, dass sowohl „high-level“ als auch „low-level“ Klassen von Abstraktionen abhängen sollten. Ein Beispiel hierfür stellt das Zusammenspiel von PageElementGroupWalker und den einzelnen PageNodes des Contents einer PageElementGroup dar. PageElementGroupWalker ist eine „high-level“ Klasse und verwendet zur Traversierung der diversesten konkreten Implementierungen von PageNode ausschließlich das Interface der abstrakten Klasse PageNode selbst. Eine beispielsweise von PageElement abgeleitete Klasse namens „Input“ hingegen erbt von der abstrakten PageNode Klasse. Auf diese Weise ist PageElementGroupWalker im Traversierungsvorgang mit allen Klassen kompatibel, die von der abstrakten Klasse PageNode ableiten.

Alles in Allem führt eine hohe Kohäsion dazu, dass Module wesentlich einfacher verständlich sind, da sie ausschließlich auf ein Ziel hinarbeiten. Gleichzeitig steigt damit auch deren Wiederverwendbarkeit.

Eine geringe Kopplung hingegen wirkt sich vor allem auf die Analysierbarkeit und Änderbarkeit positiv aus, da bei Änderungen an einer Klasse weniger von dieser abhängige Komponenten auf Anpassungen hin untersucht und bei Bedarf adaptiert werden müssen.

Nutzen

- *SQ4 Benutzbarkeit:* Hoch kohäsive Module mit geringer Kopplung sind besser verständlich und daher leichter erlernbar.
- *SQ7 Wartbarkeit:* Verbesserung der Modularität, Wiederverwendbarkeit, Analysierbarkeit und Änderbarkeit
- Die Kosten für die Wartung der Testfälle sinken, da Änderungen weniger „Nebenwirkungen“ mit sich bringen.

2.5.11 Evaluierung der Softwarequalität gemäß ISO/IEC 25010

Die Bewertung der Softwarequalität gemäß ISO/IEC 25010:2011 [34] nimmt in der Evaluierung von WDIO-WORKFLO eine Sonderrolle ein, denn sie bezieht sich auf nicht-funktionale Anforderungen, während im Fokus der vorhergehenden Unterkapitel vorwiegend funktionale Anforderungen standen. Allerdings konnten durch die meisten funktionalen Anforderungen positive Auswirkungen auf die nicht-funktionalen Qualitätskriterien erzielt werden, was sich sehr deutlich in der Bewertung der Anforderungen dieses Unterkapitels niederschlägt.

Einen Überblick über den Beitrag der funktionalen Anforderungen zu den Qualitätskriterien nach ISO/IEC 25010:2011 liefert Tabelle 3:

	SQ1	SQ2	SQ3	SQ4	SQ5	SQ6	SQ7	SQ8
Σe	1	0	0	1	0	0	9	0
$\Sigma ++$	3	4	0	6	0	0	11	1
$\Sigma +$	0	3	0	3	1	0	3	1
$\Sigma \sim$	2	1	0	1	0	0	6	0
$\Sigma -$	0	0	0	1	0	0	0	0
$\Sigma --$	0	0	0	0	0	0	2	0
Σn	0	0	0	0	0	0	0	0
Σ	6	8	0	12	1	0	31	2

Tabelle 3: Beitrag funktionaler Anforderungen zu Qualitätskriterien

Die Spalten von Tabelle 3 repräsentieren dabei die einzelnen Qualitätskriterien. Die Zeilen hingegen bilden die Summen der Bewertungen aller funktionalen Anforderungen, die zur Erfüllung eines Qualitätskriteriums beigetragen haben, ab und sind nach den Bewertungsstufen der gleitenden Skala ($\Sigma --$ bis $\Sigma ++$) sowie nach den beiden booleschen Werten „Erfüllt“ (Σe) und „Nicht Erfüllt“ (Σn) gruppiert. Die letzte Zeile spiegelt die Gesamtanzahl aller Anforderungen wieder, die zur Verbesserung eines bestimmten Qualitätskriteriums beitrugen.

Vor der detaillierten Betrachtung der einzelnen Qualitätskriterien lässt sich damit bereits feststellen, dass WDIO-WORKFLO einen besonderen Fokus auf die Verbesserung von Wartbarkeit, Bedienbarkeit und Effizienz der automatisierten funktionalen Systemtests legt.

Im Folgenden werden nun zu jedem Qualitätskriterium überblicksmäßig die wichtigsten Eigenschaften von WDIO-WORKFLO aufgeführt, die zu einer Verbesserung der jeweiligen nicht-funktionalen Anforderung beitragen. Diese können, müssen jedoch nicht zwingend einer funktionalen Anforderung entstammen. Im Nutzen Feld der Qualitätskriterien werden überwiegend deren Auswirkungen auf die Kosten der Entwicklung automatisierter funktionaler Systemtests beschrieben.

ID	Name	Bewertung																
SQ1	Funktionalität	++																
Begründung																		
<p>Gemäß ISO/IEC 25010:2011 lässt sich die Funktionalität von WDIO-WORKFLO direkt daraus ableiten, wie gut die in den vorhergehenden Kapiteln evaluierten, funktionalen Anforderungen vom Testframework umgesetzt wurden. Hierzu müssen die Teilaspekte Vollständigkeit, Korrektheit und Angemessenheit berücksichtigt werden.</p> <p>Die Grundlage für die Einhaltung dieser drei Teilaspekte bildete eine umfangreiche und detaillierte Literaturrecherche, welche jene Erkenntnisse lieferte, die zur Formulierung der funktionalen Anforderungen an WDIO-WORKFLO erforderlich waren.</p> <p>Die Umsetzung der funktionalen Anforderungen lässt sich anhand von Tabelle 4 beurteilen. Diese summiert die Bewertungen der einzelnen Anforderungen und gruppiert diese nach der gleitenden Skala ($\Sigma--$ bis $\Sigma++$) sowie nach den beiden booleschen Werten „Erfüllt“ (Σe) und „Nicht Erfüllt“ (Σn):</p> <table><tr><td></td><td>e</td><td>++</td><td>+</td><td>~</td><td>-</td><td>--</td><td>n</td></tr><tr><td>Σ</td><td>13</td><td>26</td><td>13</td><td>10</td><td>3</td><td>4</td><td>0</td></tr></table> <p>Tabelle 4: Bewertung der funktionalen Anforderungen</p> <p>Wie Tabelle 4 bestätigt, konnten alle booleschen Anforderungen von WDIO-WORKFLO erfolgreich umgesetzt werden. Von den Anforderungen, die anhand der gleitenden Skala von -- bis ++ bewertet wurden, wurde der mit Abstand größte Teil mit der Bestnote ++ beurteilt. Jeweils etwa ein Fünftel der funktionalen Anforderungen wurden gut oder mittelmäßig umgesetzt, und nur ein paar wenige funktionale Anforderungen konnten lediglich zu einem geringen Ausmaß oder gar nicht erfüllt werden.</p> <p>Weist man den Bewertungen der gleitenden Skala Schulnoten von 1 (++) bis 5 (--) zu, ergibt sich ein Durschnitt von rund 2 (+). In Anbetracht der Tatsache, dass alle booleschen Anforderungen erfüllt werden konnten, wird für das Qualitätskriterium Funktionalität insgesamt eine Bewertung von ++ vergeben.</p>				e	++	+	~	-	--	n	Σ	13	26	13	10	3	4	0
	e	++	+	~	-	--	n											
Σ	13	26	13	10	3	4	0											
Nutzen																		
<ul style="list-style-type: none">AnwenderInnen von WDIO-WORKFLO stehen alle Funktionalitäten zur Verfügung, die zum automatisierten Testen von funktionalen Systemtests erforderlich sind.																		

ID	Name	Bewertung
SQ2	Effizienz	+
<p>Begründung</p> <p>Um die Verbesserung des Qualitätskriteriums Effizienz durch WDIO-WORKFLO zu beurteilen, werden einerseits das Zeitverhalten und der Ressourcenverbrauch des Testframeworks selbst zur Laufzeit betrachtet und andererseits auch der Verbrauch menschlicher Ressourcen beim Entwickeln und Analysieren der automatisierten Tests berücksichtigt.</p> <p>In Sachen Zeitverhalten verfolgt WDIO-WORKFLO zwei Strategien:</p> <p>Einerseits werden in den Page Object Basisklassen Performance-steigernde Mechanismen implementiert. Dazu zählen etwa die Funktion „identifyBy()“ in der PageElementList und die PageElementMap, welche die Identifizierung eines bestimmten aus einer Vielzahl gleichartiger PageElements anhand des Selektors selbst ermöglichen – dies ist effizienter, als wie von WebdriverIO vorgeschlagen zunächst alle gleichartigen Elemente zurückzuliefern und diese erst daraufhin mithilfe weiterer Selenium Abfragen zu filtern. Zudem bieten auch die zahlreichen „waitXXX()“ Funktionen, so paradox dies erscheinen mag, eine effizientere Art des Wartens. Durch „waitXXX()“ können initial höhere maximale Timeouts angegeben werden, was die Robustheit der Tests fördert – da die Wartebedingungen jedoch in kurzen Intervallen überprüft werden, ist die Gesamtwartezeit in der Regel viel geringer als beim banalen verstreichen lassen einer vordefinierten Zeitspanne.</p> <p>Andererseits fördert WDIO-WORKFLO das Abprüfen ähnlicher Specs innerhalb möglichst weniger Testcases, wodurch einerseits die Gesamtdurchlaufzeit der meist sehr langatmigen funktionalen Systemtests abnimmt und andererseits auch die Kosten für die Wartung aufgrund einer geringeren Anzahl an Testcases gesenkt werden.</p> <p>Hinsichtlich Ressourcenverbrauch stellt WDIO-WORKFLO durch das Cachen von PageNode Instanzen in PageElementStores sicher, dass der Speicherverbrauch durch das Testframework geringgehalten wird. Gleichzeitig steigt auch die Performance, da die Instanzen nicht bei jedem Aufruf neu erzeugt werden müssen.</p> <p>Auch die menschlichen Aufwände beim Entwickeln und Analysieren der automatisierten Systemtests werden von WDIO-WORKFLO beispielsweise dadurch gesenkt, dass die Auswertung der Testergebnisse, das Sammeln von Testmetriken, die Ermittlung einiger Arten von Traceability Links und das Erstellen der Testfalldokumentation zum größten Teil automatisiert stattfinden. Zudem können der für die Formulierung von Specs benötigte Code durch deren verschachtelte Bauweise deutlich reduziert und Specs somit erheblich schneller definiert werden.</p>		

Jedoch könnte man kritisieren, dass bei sehr kleinen Testaufwänden die Architektur von WDIO-WORKFLO, welche eine Definition von Specs, Steps, Testcases und ein ganzes Ökosystem an Page Objects vorsieht, einen gewissen „Overkill“ darstellt.

In Kombination der oben aufgeführten Eigenschaften mit den die Effizienz betreffenden Ergebnissen in Tabelle 4 erhält das Qualitätskriterium Effizienz eine Bewertung von +.

Nutzen

- Reduktion der Kosten für Testausführung in Form reduzierter Testlaufzeiten
- Reduktion der Kosten für Entwicklung und Analyse automatisierter Testfälle durch geringeren Verbrauch menschlicher Ressourcen

ID	Name	Bewertung
SQ3	Kompatibilität	+
<p>Begründung</p> <p>Grundsätzlich konnte durch die Verwendung von WDIO-WORKFLO keine negative Beeinträchtigung anderer auf dem System installierter Software festgestellt werden. Lediglich der Rechner, auf welchem der von Selenium verwendete Browser läuft, ist im Falle eines Tests mittels Internet Explorer während des Testvorgangs nicht anderweitig benutzbar, da Internet Explorer einen ständigen Fokus auf das Browserfenster benötigt.</p> <p>Insgesamt arbeitet WDIO-WORKFLO dank Selenium Webdriver jedoch mit den weitverbreitetsten Browsern bis auf wenige Einzelfälle sehr gut zusammen. Jedoch hängt der Grad der Unterstützung stark von den Selenium-Treibern des jeweiligen Browser-Herstellers ab.</p> <p>Da WDIO-WORKFLO wie praktisch jedes Web-Frontend in JavaScript entwickelt wurde, können zudem vergleichsweise einfach Informationen zwischen den beiden Systemen ausgetauscht und Synergien genutzt werden, etwa in Form von Helper Klassen oder Dateien mit Übersetzungen.</p> <p>Nutzen</p> <ul style="list-style-type: none"> • WDIO-WORKFLO ist universal für Webanwendungen einsetzbar. • Aufwände infolge von Inkompatibilitäten können vermieden werden – etwa, wenn Helper-Klassen, die im Frontend der Webanwendung bereits existieren, erneut entwickelt werden müssten. 		

ID	Name	Bewertung
SQ4	Benutzbarkeit	+
<p>Begründung</p> <p>Viele der funktionalen Anforderungen von WDIO-WORKFLO zielen darauf ab, die Benutzbarkeit der automatisierten Testfälle zu steigern. Die größten Nutznießer sind dabei die Aspekte Bedienbarkeit und Erlernbarkeit.</p>		

So bemüht sich das Framework, das Testen auf eine Art und Weise zu gestalten, die das Vorgehen und damit auch die Sprache der EndbenutzerInnen kopiert. Hierzu tragen vor allem die Abbildung von Website-Elementen als Page Objects, sowie die Nachahmung der Interaktionen von BenutzerInnen in natürlichsprachigen Steps bei. Diese verbergen die technischen Vorgänge hinter einer applikationsspezifischen API und machen den Test-Code dadurch verständlicher und einfacher zu bedienen und zu erlernen. Zudem tragen die Anforderungen „FD4 Abstraktion und Separation of Concerns“ sowie „FD5 Funktionale Unabhängigkeit“ in großem Maße dazu bei, den komplexen Testvorgang für Menschen einfacher darzustellen und damit „anwendungsfreundlicher“ zu gestalten. Nicht zuletzt hilft auch der Einsatz bewährter und bekannter Entwurfsmuster den TestfallentwicklerInnen in ihrem Unterfangen, den Aufbau und die Funktionsweise des Testframeworks zu verstehen.

Ein großer Minuspunkt in Sachen Erlernbarkeit ist jedoch die, mit Ausnahme eines für sich selbst sprechenden Codes, fehlende Dokumentation von WDIO-WORKFLO.

Dafür bietet auf Seiten der Bedienbarkeit der Einsatz von TypeScript in Kombination mit Visual Studio Code zahlreiche IntelliSense Komfortfunktionen (Anzeige aller Referenzen einer Variablen oder Funktion, Autocompletion, Code-Vorschau bei „MouseOver“ etc.), die die Entwicklung von automatisierten Testfällen wesentlich vereinfachen.

TypeScript schafft zudem Typensicherheit, was Fehler in der Handhabung des Frameworks durch dessen AnwenderInnen verhindert. Gemeinsam mit der Funktion „getUid()“, welche sicherstellt, dass einzigartige Testdaten auch wirklich einzigartig bleiben, können die Testfälle auf diese Weise robuster gestaltet werden.

Ein letztes Manko in der Benutzbarkeit von WDIO-WORKFLO stellt die Tatsache dar, dass dieses nicht über eine graphische Benutzeroberfläche für die Testfallentwicklung verfügt, was die Barrierefreiheit und Ästhetik des Testframeworks verringert. Da jedoch von Anfang an EntwicklerInnen und TesterInnen mit Programmierkenntnissen als Zielgruppe des Frameworks definiert wurden, führt dies in der Bewertung der Anforderung „SQ4 Benutzerbarkeit“ nicht zu einer Abwertung.

, Nutzen

- Die gesteigerte Erlernbarkeit und Bedienbarkeit von WDIO-WORKFLO äußern sich in geringeren Kosten für die Framework-Einführung und die Testfallentwicklung.

ID	Name	Bewertung
SQ5	Zuverlässigkeit	+
Begründung		
In Form der „waitXXX()“ Funktionen und durch den „postCondition“ Parameter der „click()“ Funktion beinhalten die PageElements von WDIO-WORKFLO Mechanismen, welche die Zuverlässigkeit der automatisierten Testfälle erhöhen, indem „Fallbacks“ und „sichere“ Wartebedingungen Abhilfe für ansonsten eher unzuverlässige Vorgänge schaffen.		

Andererseits können, je nach eingesetztem Browser, mangelhaft umgesetzte Selenium-Treiber zu unerwartetem Testverhalten und damit instabilen Tests führen, die jedoch in der Regel reproduzierbar und daher korrigierbar sind.

Nutzen

- „Fallback“-Mechanismen steigern die Robustheit von Testfällen.
- Durch die verwendeten Selenium-Treiber verursachtes, unerwartetes Fehlverhalten der Testfälle erfordert zusätzliche Kosten für Analyse und Testfallentwicklung.

ID	Name	Bewertung
SQ6	Sicherheit	-
<p>Begründung</p> <p>Die Sicherheit stellt jenes Qualitätskriterium automatisierter funktionaler Systemtests dar, welches durch WDIO-WORKFLO am geringsten gesteigert wird.</p> <p>Das Testframework selbst verfügt über keinerlei Sicherheitsmechanismen und das Testen der Sicherheit in der Webanwendung ist auf simple Authentifizierungsvorgänge und Überprüfungen autorisierter Aktionen beschränkt.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • Die negativen Auswirkungen mangelhafter Sicherheit in einem Testframework sind als gering anzusehen, da dieses im Normalfall immer in eigenen Testumgebungen operiert. Zudem steht den funktionalen Systemtests keine gesonderte Testschnittstelle, sondern lediglich die graphische Benutzeroberfläche der Anwendung zur Verfügung, sodass der Angriffsvektor von WDIO-WORKFLO äußerst klein gehalten werden kann. 		

ID	Name	Bewertung
SQ7	Wartbarkeit	++
<p>Begründung</p> <p>Die Wartbarkeit stellt jenes Qualitätskriterium dar, das durch WDIO-WORKFLO am stärksten gefördert wird. Wie Tabelle 3 belegt, sind mehr als die Hälfte aller Förderungen von Qualitätsmerkmalen durch funktionale Anforderungen von WDIO-WORKFLO auf Steigerungen der Wartbarkeit zurückzuführen.</p> <p>Den größten Beitrag zur Steigerung der Wartbarkeit bilden dabei jene Anforderungen, die die Architektur des Frameworks betreffen. Darunter fallen insbesondere die Anforderungen „FD1 Förderung der Wiederverwendbarkeit“, „FD4 Abstraktion und Separation of Concerns“ und „FD5 Funktionale Unabhängigkeit“. Umgesetzt wurden diese generellen Anforderungen an das Framework-Design vor allem durch die konkreten Anforderungen „RW1 Unterstützung für Abbildung von Website-Komponenten“ und „TW8 Unterstützung einer Daten- und Keyword-getriebenen Testfallarchitektur“. So sind Page Objects und Steps in außerordentlich hohem Maße wiederverwendbar, nachdem sie einmal implementiert wurden. Mit deren Wiederverwendung steigt gleichzeitig die</p>		

Änderbarkeit, da Änderungen, die die selbe Funktionalität betreffen, nur mehr an einer einzigen Stelle im Test-Code vorgenommen werden müssen. Darüber hinaus steigt damit auch die Analysierbarkeit, da mithilfe von Visual Studio Code's „Alle Verweise suchen“ Feature sehr rasch festgestellt werden kann, in welchen Testcases geänderte Steps und in welchen Steps geänderte Page Objects verwendet werden. Gemäß der Forderung „Seperation of Concerns“ hat in WDIO-WORKFLO zudem jede Komponente eine klar definierte Aufgabe und, wie von der funktionalen Unabhängigkeit vorgeschrieben, eine hohe Kohäsion, was sich wiederum sehr positiv auf deren Wiederverwendbarkeit und Änderbarkeit auswirkt. Außerdem fällt durch die geringe Kopplung zwischen den einzelnen Komponenten von WDIO-WORKFLO der Aufwand für die Abhängigkeitsanalyse (Analysability) im Zuge von Änderungen sehr gering aus.

Ein weiterer großer Fokus von WDIO-WORKFLO, welcher der Wartbarkeit ebenso zugutekommt, ist die in Kapitel 2.2.6 behandelte und in Kapitel 2.5.7 evaluierte Gestaltung von Anforderungen. Diese trägt durch die bessere Nachvollziehbarkeit von Zuständen und Zustandsänderungen in der GUI, sowie durch die erhöhte Verständlichkeit der Anforderungen, zu einer gesteigerten Analysierbarkeit der automatisierten funktionalen Systemtests bei, wirkt sich durch die Vermeidung von Redundanzen positiv auf die Änderbarkeit aus und stellt zudem die Testbarkeit der Anforderungen sicher.

Die Gestaltung der Anforderungen beinhaltet ebenso einen weiteren wichtigen Schwerpunkt von WDIO-WORKFLO, nämlich Traceability. Diese ist jedoch nicht nur in der Anforderungsgestaltung durch die Verknüpfungen von Anforderungen unterschiedlicher Ebenen über Links zu Issue Management Tools vorhanden, sondern wird vom Testframework auch zwischen Specs und Testcases, sowie zwischen Testcases und Defects ermöglicht. Zudem wertet das Framework in Reports automatisch aus, in welchen Dateien bestimmte Testcases und Specs definiert wurden. Insgesamt trägt Traceability zu einer wesentlichen Verbesserung der Analysierbarkeit bei, da die Auswirkungen von Änderungen zwischen unterschiedlichen Software-Artefakten effizient nachvollzogen werden können.

Ein Faktor, der die Testbarkeit zusätzlich fördert, ist die Unterstützung der Identifikation eines Website-Elements über die Struktur der Webseite in Form von XPath Selektoren.

Schließlich erlaubt der Einsatz von TypeScript in Zusammenarbeit mit Visual Studio Code, zahlreiche Komfortfunktionen zu nutzen, die nur mittels „static checking“ möglich sind. Dazu zählen beispielsweise ein sehr effizientes Refactoring und das Zurückverfolgen referenzierter Variablen und Methoden, sowie Typenchecks zur Kompilierzeit. Insgesamt tragen diese Komfortfunktionen ebenso zu Verbesserungen in den Bereichen Analysierbarkeit und Änderbarkeit bei.

Nutzen
<ul style="list-style-type: none"> • Erhebliche Reduktion der Entwicklungskosten durch Wiederverwendbarkeit • Drastische Senkung der Wartungskosten durch gesteigerte Änderbarkeit und Analysierbarkeit

ID	Name	Bewertung
SQ8	Portabilität	+
<p>Begründung</p> <p>Die Anpassbarkeit von WDIO-WORKFLO ist als äußerst hoch anzusehen, da die mit dem Testframework entwickelten Testfälle sowohl auf Windows, Linux als auch MacOS ausgeführt werden können und zudem nur eine einzige Zeile in der Konfigurationsdatei von WDIO-WORKFLO angepasst werden muss, um die Tests in einem anderen Browser zu starten.</p> <p>Auch in Sachen Installierbarkeit überzeugt WDIO-WORKFLO: Zur Installation des Frameworks muss lediglich der Einzeiler „npm install wdio-workflo“ ausgeführt werden. Die Konfiguration des Frameworks erfolgt durch einige wenige Einträge in einer einzelnen Konfigurationsdatei.</p> <p>Lediglich bei der Austauschbarkeit enttäuscht WDIO-WORKFLO, denn die einzelnen Komponenten von WDIO-WORKFLO sind in vielen Fällen aufeinander abgestimmt und können daher nicht ohne Anpassungen durch Alternativen ersetzt werden. Zudem liegen die meisten erzeugten Test-Artefakte nicht in einem standardisierten Format vor und können daher ausschließlich von WDIO-WORKFLO „verstanden“ werden.</p>		
<p>Nutzen</p> <ul style="list-style-type: none"> • Die hohe Anpassbarkeit und Installierbarkeit von WDIO-WORKFLO senken die Kosten zur Einrichtung der Testumgebung auf ein Minimum. • Andererseits ist das Framework, sobald es einmal in Verwendung ist, nur schwer durch alternative Lösungen ersetzbar. 		

2.5.12 Auswirkungen auf die Kosten der Testautomatisierung

2.5.12.1 Überblick über die Wirtschaftlichkeit von Testautomatisierung

Wie bereits in Kapitel 2.5.12 angekündigt, erfolgt die Evaluierung erzielter Kostensenkungen automatisierter funktionaler Systemtests aufgrund der Verwendung von WDIO-WORKFLO nicht mittels direkter Messungen, sondern indirekt durch die Bewertung von Anforderungen, welche großen Einfluss auf die Testkosten haben. Dazu wurde in den vorhergehenden Kapiteln, welche sich mit der Evaluierung einzelner Anforderungsbereiche befassten, bereits an mehreren Stellen auf Faktoren hingewiesen, die positive Auswirkungen auf die Kosten automatisierter funktionaler Systemtests haben.

Dieses Kapitel untersucht zunächst, auf welchen Ebenen die Automatisierung funktionaler Systemtests mittels WDIO-WORKFLO zu Kostenreduktionen führen kann, indem die oben erwähnten Faktoren, welche zu einer Senkung der Testkosten beitragen, den jeweiligen Ebenen zugeordnet und bei Bedarf näher begründet werden. Dustin et al. [15] unterscheiden hierfür zwischen den Ebenen Testsetup, Testentwicklung, Testausführung und Testanalyse, wie man Abbildung 73 entnehmen kann, welche eine beispielhafte Berechnung der Ersparnisse einer Testautomatisierung im Vergleich zu manueller Testausführung vornimmt:

Overall Test Automation Savings Worksheet			
Tasks	Automated Testing Time Savings (hours)	Tasks	Automated Testing Cost Savings (\$)
Test setup time savings	xxx	Test setup time savings	xxx
Test development time savings	-250	Test development cost savings	-\$25,000
Test execution time savings	1,583.33	Test execution cost savings	\$158,333
Test evaluation/ diagnostics time savings	2,250	Test evaluation/ diagnostics cost savings	\$225,000
		Other automation costs	-\$25,000
Hours	3,583.33	Dollars	\$333,333
Days (8-hour days)	447.9		
Months (20-day months)	22.4		

Abbildung 73: Berechnung der Ersparnisse durch Testautomatisierung (Quelle: [15])

Im Anschluss wird darauf eingegangen, wie durch frühzeitig entdeckte Fehler Kostenvorteile erzielt werden können. Schließlich erfolgt am Ende des Kapitels eine Erklärung, warum Testautomatisierung als langfristiges Investment gesehen werden muss und welche Faktoren bei der Entscheidung, ob und in welchem Ausmaß funktionale Systemtests automatisiert werden sollen, zusätzlich berücksichtigt werden müssen.

2.5.12.2 Einsparungen im Testsetup

Während gegenüber manueller Testausführung keine Einsparungen im Testsetup erzielt werden können, bietet der Einsatz von WDIO-WORKFLO bei einer automatisierten Testausführung vor allem durch die gute Umsetzung des Qualitätskriteriums „SQ8 Portabilität“ die Möglichkeit, die Kosten für die Testautomatisierung im Bereich Testsetup zu senken. Dies wird insbesondere durch die hohe Anpassbarkeit und Installierbarkeit von WDIO-WORKFLO sichergestellt. Zudem besitzt WDIO-WORKFLO den großen Vorteil, dass alle Bestandteile des Testframeworks nach reiflicher Überlegung und Recherche ausgewählt und genau aufeinander abgestimmt wurden. Es muss also kaum Zeit in den Aufbau eines „Testing Stacks“ investiert werden und die wohl überlegte Architektur des Testframeworks erleichtert darüber hinaus den Einstieg in die Testautomatisierung funktionaler Systemtests von Webanwendungen.

2.5.12.3 Einsparungen in der Testentwicklung

Eine Vielzahl an Qualitätskriterien und Anforderungen, die diese umsetzen, führen in WDIO-WORKFLO zu einer Senkung der Kosten für Testentwicklung. Hierzu zählen etwa die verschachtelte Bauweise der Specs und die damit einhergehende Code-Ersparnis zur Steigerung des Qualitätskriteriums „SQ2 Effizienz“, die Möglichkeit, im Rahmen von „SQ3 Kompatibilität“ Synergien in der Entwicklung von Tests und dem Frontend zu nutzen, sowie die durch „SQ4 Benutzbarkeit“ gesteigerte Erlernbarkeit und Bedienbarkeit von WDIO-WORKFLO.

Die Einsparungen in der Testentwicklung beruhen jedoch nicht allein auf Kostensenkungen in der initialen Entwicklung von Testfällen und Specs, sondern zu einem erheblichen Teil auch auf reduzierten Aufwänden für deren Wartung. Laut Capers Jones [47, p. 4] stellten Aufwände für Wartung im Jahr 2017 sogar rund 77 Prozent der gesamten Softwareentwicklungskosten dar.

Aus diesem Grund ist „SQ7 Wartbarkeit“ jenes Qualitätskriterium, das durch WDIO-WORKFLO am meisten gesteigert wird, da hier ein enormes Potential an Kostenreduktionen gegeben ist. Während dieses Qualitätskriterium ebenso die Wiederverwendbarkeit einzelner Softwarebestandteile umfasst, welche durch Steps und Page Objects stark gefördert wird, konzentrieren sich die Aspekte Änderbarkeit und Analysierbarkeit vorwiegend auf die Anpassung bestehender Software. Alle drei Teilaspekte tragen in außergewöhnlich hohem Maße zu einer Senkung der Entwicklungskosten bei und wurden gezielt in die Architektur von WDIO-WORKFLO miteinbezogen und im Rahmen der Evaluierung von „SQ7 Wartbarkeit“ in Kapitel 2.5.11 mit der Bestnote „++“ beurteilt.

2.5.12.4 Einsparungen in der Testausführung

Die Ausführungszeiten funktionaler Systemtests sind bei automatisierten Tests erfahrungsgemäß in einigen Bereichen, etwa dem Ausfüllen eines Formulars, niedriger als jene manueller Tests, während andere Aufgaben, wie das Durchsuchen einer Tabelle nach bestimmten Zeileneinträgen, schneller durch Menschen bewerkstelligt werden als automatisiert. Im Großen und Ganzen sind die mittels WDIO-WORKFLO automatisierten Tests in der Ausführung etwa gleich schnell wie manuell durchgeführte Tests. Allerdings ist in dieser Aussage der Ermüdungseffekt bei menschlichen Testern, welcher bei automatisierten Tests nicht auftritt, noch nicht berücksichtigt.

Allerdings bietet WDIO-WORKFLO im Vergleich zu „herkömmlich“ automatisierten Tests, welche ebenso auf Selenium beruhen, die in den Kapiteln 2.4.5.4 und 2.4.5.5 näher beschriebenen Funktionen „identifyBy()“, „waitXXX()“ und die Basisklasse PageElementMap, welche allesamt darauf abzielen, die Laufzeiten der automatisierten Tests zu senken.

2.5.12.5 Einsparungen in der Testauswertung

Auch in der letzten Ebene von Kostensenkungen durch Testautomatisierung aus Abbildung 73, den Einsparungen in der Auswertung, kann WDIO-WORKFLO überzeugen: So ermöglicht der in „AU4 Senkung des menschlichen Anteils an der Ergebnisanalyse“ geforderte Einsatz von Assertion Matchers aus der Jasmine Testing Library [21], dass die erwarteten und die tatsächlichen Ergebnisse eines Testfalls ohne menschliches Zutun verglichen werden können, wodurch der Aufwand für die Testauswertung beträchtlich sinkt. Zudem erstellt WDIO-WORKFLO im Fehlerfall automatisch Screenshots der getesteten Applikation und fügt diese, gemeinsam mit Stacktraces, im Report hinzu, was die Analyse von Fehlern beschleunigt.

Darüber hinaus nimmt WDIO-WORKFLO durch die Umsetzung der Anforderung „RE3 Möglichkeit zur Erstellung kompletter Testdokumentation“ den TesterInnen das manuelle Erstellen von Testspezifikation und Testabschlussbericht ab und auch Metriken für das Testmanagement werden vom Testframework automatisch erhoben, wodurch viel Zeit in der Testauswertung gespart werden kann.

2.5.12.6 Kostensenkungen durch frühzeitig erkannte Fehler

Im Allgemeinen gilt in der Softwareentwicklung: Je früher ein Fehler entdeckt und behoben wird, desto geringer sind die Kosten, die durch diesen verursacht werden. Dem zugrunde liegt die sogenannte „Rule of Ten“, welche Frank Witte [31, pp. 89-92] wie folgt beschreibt: Je nachdem, in welchem Stadium des Softwareentwicklungsprozesses Fehler entdeckt werden, sind die durch diese verursachten Kosten im Vergleich zur Anforderungsdefinition in der Entwicklung zehn Mal, im Test hundert Mal und im Live-Betrieb tausend Mal so hoch.

Auch wenn dieser Merksatz etwas übertrieben klingen mag, beweist Abbildung 74, welche auf von Barry Boehm erhobenen Daten beruht und von Roger Pressman [6, p. 423] veröffentlicht wurde, dass der Kern der Aussage auf jeden Fall zutrifft:

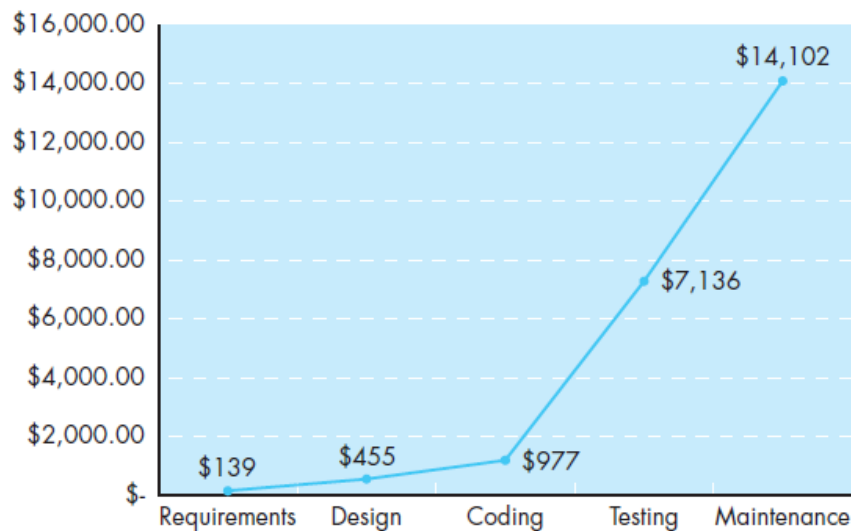


Abbildung 74: Relative Kosten, um Fehler zu korrigieren [6, p. 423]

Aus diesem Grund strebt WDIO-WORKFLO durch gezielte Maßnahmen danach, Fehler in der getesteten Software, sowie in den Testfällen selbst, so früh wie möglich aufzudecken:

Am besten ist, wenn Fehler erst gar nicht zustande kommen. Um dies zu erreichen, wird durch die Anforderung „TW7 Statische Überprüfung des Source-Codes“ mit Hilfe von „static checking“ eine Analyse des Test-Codes ausgeführt, noch bevor dieser ausgeführt wird. Zusätzlich stellt die Verwendung von TypeScript sicher, dass Code mit Syntax- und Typenfehlern erst gar nicht kompiliert.

Ein weiterer Fokus von WDIO-WORKFLO zur frühzeitigen Entdeckung von Fehlern liegt auf der Gestaltung und Verifizierung von Anforderungen. Gemäß „ST1 Trennung zwischen logischen und konkreten Testfällen“ können die logischen Testfälle in Form von Specs bereits sehr früh im Softwareentwicklungsprozess, nämlich vor der Entwicklung, geschrieben werden. Um nun Fehler in den Anforderungen anhand der Specs aufzudecken, werden diese im Rahmen der Anforderung „FS1 Verifizierung der Anforderungen im Rahmen der externen Spezifikation“ geprüft. So können fehlerhafte, inkonsistente oder unvollständige Anforderungen bereits frühzeitig erkannt und korrigiert werden, was, wie Abbildung 74 verdeutlicht, zu großen Einsparungen bei den Fehlerkosten führt.

Kapitel 2.2.6 erklärt, wie genau Anforderungen beschaffen sein sollten, um möglichst von Fehlern verschont zu bleiben, und definiert zudem eine Reihe von Forderungen an das Testframework, um eine möglichst hochqualitative Anforderungsbeschreibung sicherzustellen.

Wie wichtig eine hohe Qualität in der Anforderungsbeschreibung ist, belegt Capers Jones [16, pp. 259, 475], demzufolge die schwerwiegendsten Fehler vorrangig aus fehlerhaften Anforderungen stammen. Doch nicht nur die Schwere, sondern auch die Anzahl der Fehler sind im Web Bereich mit 40 Prozent aller Fehler in der Anforderungsdefinition außergewöhnlich stark vertreten. Über die gesamte Branche hinweg ergeben Durchschnittswerte laut Thaller [30, p. 93], dass von 50 Fehlern pro tausend Zeilen Code 10 Fehler ihren Ursprung in der Analyse der Anforderungen haben und 12,5 Fehler dem Design der Software entspringen.

Jene Fehler, die erst während der Implementierung in die Software eingebracht werden, können zwangsläufig jedoch erst durch Testen aufgedeckt werden. Um hierbei ein möglichst frühes Feedback an die Entwickler zu ermöglichen, erlaubt die Anforderung „FS7 Unterstützung von Smoke Testing“ die wichtigsten Bereiche der Anwendung rasch automatisiert zu testen. Eine gründliche, aber etwa im Rahmen von „Nightly Builds“ immer noch zeitnahe Rückmeldung über die Auswirkungen von Änderungen an der Software, gibt schließlich die Anforderung „AU7 Unterstützung von Regressionstests“. Sowohl Smoke Tests als auch Regressionstests zielen darauf ab, dass Fehler nicht erst in der Phase des Live-Betriebs, sondern bereits während der Testphase entdeckt und korrigiert werden, um die Fehlerkosten gering zu halten.

2.5.12.7 Automatisierung als langfristige Investition

Trotz aller Maßnahmen, mit denen WDIO-WROFKLO versucht, die Kosten automatisierter funktionaler Systemtests und des gesamten Softwareprodukts durch eine gesteigerte Qualität zu senken, soll an dieser Stelle darauf hingewiesen werden, dass Qualität auch seinen Preis hat. Die Automatisierung funktionaler Systemtests und deren Entwicklung mithilfe von WDIO-WORKFLO führt initial zu einem erheblichen Mehraufwand gegenüber manuellen Tests und weniger ausgereiften Testskripten. In sehr kleinen Projekten oder bei äußerst geringem Testumfang kann es daher aus wirtschaftlicher Hinsicht sinnvoller sein, die funktionalen Systemtests von Hand durchzuführen oder Testskripte in einem imperativen, auf hintereinander gereihten Selenium Commands basierten Stil, zu entwickeln.

Je öfter allerdings die automatisierten Tests ausgeführt werden, desto mehr rentiert sich das initiale Investment für die Automatisierung, wie Formel (1) veranschaulicht:

$$\text{Ersparnis} = n * (T_m - T_a) - A \quad (1) \quad (\text{Quelle [49, p. 61]})$$

Hierbei repräsentieren n die Anzahl der Testläufe, T_m den Aufwand für eine manuelle Durchführung, T_a den Aufwand für eine automatisierte Durchführung und A die Kosten für die Automatisierung der Testfälle.

Die Kosten für die automatisierte Durchführung von Testläufen sind im Vergleich zu den Kosten für die manuelle Durchführung äußerst gering, da für diese, mit Ausnahme einiger Analysetätigkeiten der Ergebnisse, keine menschlichen Ressourcen erforderlich sind. Somit kann anhand von Formel (1) festgestellt werden, dass sich die Automatisierung der funktionalen Systemtests wirtschaftlich auszahlt, sobald das Produkt aus Testläufen n mal der durch die Automatisierung reduzierten Aufwände für einen Testlauf $T_m - T_a$ die initialen Aufwände für die Automatisierung A übersteigt.

Allerdings ist die Anzahl an Testläufen, um ein positives Return of Investment zu erzielen, von Projekt zu Projekt unterschiedlich und kann nicht pauschal beziffert werden. Dustin et al. [15] weisen zudem darauf hin, dass für eine realitätsnähere Berechnung der Wirtschaftlichkeit ebenso die Kosten für die Wartung der automatisierten Testfälle berücksichtigt werden müssen, und nicht nur deren initiale Entwicklungskosten. Gleichzeitig räumen die Autoren jedoch ein, dass die Testabdeckung durch automatisierte Tests im Vergleich zu manuellen Tests meist beträchtlich erhöht werden kann und, dass die automatisierten Tests im Rahmen von Regressionstests und Smoke Tests viel häufiger ausgeführt werden können als manuell durchgeführte Tests und zudem schnelleres Feedback für die EntwicklerInnen der Softwareanwendung erlauben, wodurch Fehlerkosten aufgrund der frühzeitigeren Entdeckung von Fehlern gesenkt werden.

3 Beurteilung und Schlussbetrachtung

3.1 Diskussion der Resultate der Framework-Evaluierung

In Kapitel 2.5 wurden die Erfüllung der an das Framework gestellten Anforderungen, sowie die durch das Framework erzielten Qualitätssteigerungen und Kostensenkungen im Detail evaluiert und somit die Ergebnisse dieser Arbeit ausführlich präsentiert. Ziel dieses Kapitels ist es nun, die tatsächlichen Resultate den ursprünglichen Erwartungen an das entwickelte Testframework gegenüberzustellen, um die dadurch gewonnen Erfahrungen in künftige Arbeiten im Bereich der Automatisierung funktionaler Systemtests miteinfließen lassen zu können. Zugleich werden Fragestellungen aufgezeigt, die im Verlauf dieser Arbeit nicht ausreichend geklärt werden konnten und einer eingehenderen Untersuchung bedürfen.

Zunächst sei darauf hingewiesen, dass die anfänglichen Erwartungen stark von den Erfordernissen des beruflichen Alltags, in dem das Testframework später zum Einsatz kam, geprägt waren. Konkret wurde die Einhaltung der Anforderungen einer Webapplikation für die Erstellung und Verwaltung von Angeboten mittels funktionaler Systemtests deren graphischer Oberfläche überprüft. Vor dem Beginn der Arbeiten an WDIO-WORKFLO wurde bereits eine Automatisierung der funktionalen Systemtests durchgeführt, die jedoch nur schleppend vorankam und zudem keine integrierte Zuordnung zwischen Testfällen und zugehörigen Anforderungen erlaubte. So war die Analyse, welche Anforderungen durch fehlerhafte Testfälle beeinträchtigt waren, sehr aufwändig. Die zwei wichtigsten Gründe für die Entwicklung von WDIO-WORKFLO waren daher, die Implementierung der automatisierten Testfälle zu beschleunigen und zugleich eine Verknüpfung zwischen Testfällen und Anforderungen innerhalb des Frameworks umzusetzen.

Die Literaturrecherche, die im Zuge dieser Masterarbeit durchgeführt wurde, offenbarte darüber hinaus jedoch noch viel weitreichendere und vielseitigere Potenziale, die durch das entwickelte Framework realisiert werden könnten. Zahlreiche Aussagen der Fachliteratur ließen die Vermutung zu, dass durch den Einsatz von WDIO-WORKFLO signifikante Qualitätssteigerungen und Kostenreduktionen der funktionalen Systemtests und der gesamten Software möglich wären. Daher wurden auf Basis dieser Aussagen funktionale Anforderungen definiert und Maßnahmen zur Qualitätssteigerung und Kostenreduktion abgeleitet, welche im Design und der Entwicklung von WDIO-WORKFLO ebenso wie die ursprünglichen treibenden Faktoren Berücksichtigung fanden.

Um den Vergleich zwischen anfänglichen Erwartungen und erzielten Ergebnissen strukturierter zu gestalten, werden diese getrennt nach den vier Phasen Testfallentwurf, Testfallentwicklung, Testausführung und Testanalyse in eigenen Unterkapiteln betrachtet. Zudem werden in zwei weiteren Unterkapiteln die Einrichtung der Testumgebung diskutiert und nicht gänzlich geklärte, noch näher zu untersuchende Fragestellungen aufgezeigt.

3.1.1 Diskussion der Resultate im Testfallentwurf

Wie bereits angesprochen, war eine zentrale Motivation im Testfallentwurf, die Testfälle mit den durch ihnen überprüften Anforderungen zu verknüpfen. Dazu mussten die Anforderungen testbar gestaltet werden und somit logisch überprüfbar sein. Darüber hinaus gab es jedoch keine weiteren Erwartungen im Bereich des Testfallentwurfs.

Rückblickend betrachtet konnten diese beiden, in ihrem Umfang doch relativ bescheidenen Voraussetzungen, mehr als bloß erfüllt werden: Die `validate()` Funktion ermöglicht, beliebig viele Specs in einem Testcase zu referenzieren und mit der Einführung von Specs als Framework-eigene Formalisierungsschicht für Anforderungen wurde ein gänzlich neues Level an Qualität und Kostensenkungspotenzialen im Bereich des Testfallentwurfs geschaffen.

Da Specs in der Gherkin Language definiert sind, werden sie sowohl in natürlicher Sprache und damit für alle Stakeholder leicht verständlich ausgedrückt, als auch nach logischen Regeln überprüfbar und daher testbar formuliert. Die Beschreibung der Vorbedingungen mittels „Given“, der Zustandsänderungen mittels „When“ und der Zustandsüberprüfungen mittels „Then“ steigern die Qualität der Anforderungen, indem sie die Sicherstellung deren Konsistenz, Vollständigkeit und Eindeutigkeit unterstützen.

Die zeitliche und räumliche Trennung zwischen Specs als logischen und Testcases als konkreten Testfällen ermöglicht zudem, die ursprünglichen Anforderungen bereits vor der eigentlichen Implementierung der Software zu verifizieren und dadurch Fehler in diesen sehr früh im Entwicklungsprozess aufzudecken. Da fehlerhafte Anforderungen oftmals besonders schwerwiegende Folgen haben, können somit auch die Kosten, die durch diese im Projekt entstehen, bedeutend gesenkt werden.

Insgesamt kann im Bereich des Testfallentwurfs also zusammengefasst werden, dass die ursprünglichen Erwartungen nicht nur erfüllt, sondern in weitem Maße übertroffen wurden. Im gleichen Zuge bedeutet dies jedoch auch, dass der Aufwand, der im Design und der Implementierung des Frameworks in die Gestaltung der Anforderungen floss, deutlich unterschätzt wurde.

3.1.2 Diskussion der Resultate in der Testfallentwicklung

Eine Beschleunigung der Testfallentwicklung durch wiederverwendbare Komponenten wurde von Beginn an angestrebt. Ziel war es hierbei zunächst, den imperativen Charakter der Testskripte, die anfänglich nur aus einer Abfolge an Selenium Befehlen bestanden, mehr an den objektorientierten Programmierstil anzupassen, der auch in der Entwicklung der eigentlichen Webapplikation angewandt wurde.

Wie erwartet bestätigte die Evaluierung im Zuge der Untersuchung des Qualitätskriteriums „Wartbarkeit“ eine sehr hohe Wiederverwendbarkeit. Ebenso bestätigt wurde die bereits vor Beginn der Framework Implementierung getroffene Annahme, dass die Verwendung einer typischeren Sprache in Form von TypeScript zu einer Reduzierung der während der Testfallentwicklung eingebrachten Fehler und damit zu einer Verringerung der Fehlerkosten der Testfälle selbst führt. Überraschend waren jedoch die zahlreichen weiteren Vorteile, die aus der Verwendung der in hohem Grade wiederverwendbaren Komponenten „Steps“ und „Page Objects“ und dem Einsatz von TypeScript realisiert werden konnten:

So stellte sich heraus, dass TypeScript nicht nur vor fehlerhaftem Testcase-Code schützt, sondern auch die Entwicklung der Testfälle durch zahlreiche Komfortfunktionen erleichtert, welche von Visual Studio Code's IntelliSense Feature dank dem Einsatz von TypeScript zur Verfügung gestellt werden konnten (Viele dieser Features erfordern eine typischere Programmiersprache). Beispielsweise konnte durch die Auto-Vervollständigung von Step-Namen, die Vorschau von Parametern mitsamt deren Typen beim Aufruf von Funktionen, der Darstellung aller auf einem Objekt definierten Properties oder der Möglichkeit, sich alle Verweise auf eine Funktion oder Variable anzeigen zu lassen, viel Zeit und daher auch viele Kosten in der Testfallentwicklung und -wartung eingespart werden.

Die soeben aufgezählten, durch den Einsatz von TypeScript ermöglichten IntelliSense Features trugen zudem in hohem Maße zu einer deutlich verbesserten Wartbarkeit bei, da sie im Gegensatz zu herkömmlichem JavaScript viel mehr Wissen durch den Code selbst vermitteln können. Es macht etwa einen großen Unterschied, ob die Typen für einen Funktionsaufruf oder alle auf einem Objekt definierten Properties bereits während der Testfallimplementierung von IntelliSense eingeblendet werden, oder ob die EntwicklerInnen dafür mühsam den Ursprung der aufgerufenen Funktion oder des verwendeten Objekts zurückverfolgen und sich das Wissen über deren Beschaffenheit langwierig selbst aneignen müssen. Dieser Umstand gewinnt umso mehr an Bedeutung, als dass der Aufwand für die Wartung von Testfällen im Vorfeld kaum beachtet und komplett unterschätzt wurde, obwohl sich während der Literaturrecherche herausstellte, dass Wartungskosten im Durchschnitt knapp 77 Prozent der gesamten Projektaufwände in der Softwarebranche ausmachen.

Insgesamt trugen die in natürlicher Sprache definierten Steps, eine dank dem Einsatz des Page Object Patterns applikationsspezifische API sowie der Einsatz von TypeScript zu einer merklichen Erhöhung der Verständlichkeit und Erlernbarkeit des Codes von Testfällen bei, womit zu Beginn dieser Masterarbeit nicht wirklich gerechnet wurde.

Von einer Steigerung der Wartbarkeit und einer Senkung der Entwicklungskosten durch den Einsatz wiederverwendbarer Komponenten wurde zwar von vornherein ausgegangen, allerdings führten diese im Zusammenspiel mit den bereits erläuterten IntelliSense Features außerdem zu einer deutlich höheren Analysierbarkeit und Änderbarkeit der Testfälle, was nicht der anfänglichen Ziele war. So wurde erst nach einem längerfristigen Einsatz des

Testframeworks und einer damit einhergehenden Änderung der ursprünglichen Anforderungen an die entwickelte Webapplikation sichtbar, dass die in Page Objects gekapselte Struktur und Logik der Webseite nur an einer einzigen Stelle an die geänderten Anforderungen angepasst werden mussten. Außerdem erwies sich in dieser Hinsicht die Suche nach allen Verweisen auf eine Variable oder Funktion als drastischer Vorteil in der Analysierbarkeit der Adaptierungen.

Zusammengefasst wurden die erwartete Beschleunigung und Kostensenkung der Testfallentwicklung tatsächlich erreicht und zusätzlich ein hoher qualitativer Mehrwert in Sachen Usability und Wartbarkeit erzielt. Gerade in der Wartung der Testfälle können durch die erhöhte Verständlichkeit, Analysierbarkeit und Änderbarkeit zudem beträchtliche Kosten eingespart werden.

3.1.3 Diskussion der Resultate im Bereich Testausführung

Im Vergleich zu manuell ausgeführten, oder ohne den Einsatz von WDIO-WORKFLO (jedoch mit einem ähnlichen Technology-Stack) automatisierten funktionalen Systemtests wurde durch die Verwendung von WDIO-WORKFLO ursprünglich mit einer Beschleunigung der Testausführung gerechnet. Es bestand die Hoffnung, dadurch die Zeitspanne zwischen dem Start der Testausführung und der Verfügbarkeit der Resultate zu senken, um ein schnelleres Feedback an die EntwicklerInnen zu ermöglichen. Aufgrund mehrerer Faktoren, die im Folgenden näher erläutert werden, konnten diese Erwartungen allerdings nur bedingt erfüllt werden:

So zeigte sich beim Einsatz von WDIO-WORKFLO im Rahmen der konkreten Webanwendung zur Erstellung und Verwaltung von Angeboten, dass mithilfe des Testframeworks einzelne Schritte wie das Ausfüllen von Formularfeldern gegenüber manueller Ausführung zwar geringfügig beschleunigt werden konnten, andere Szenarien wie das Durchsuchen einer Liste oder Tabelle nach bestimmten Einträgen jedoch in der automatisierten Fassung mehr Zeit benötigten als bei händischer Durchführung.

Auch gegenüber funktionalen Systemtests, die ohne den Einsatz von WDIO-WORKFLO automatisiert wurden, konnten durch die Verwendung des Testframeworks keine wesentlichen Geschwindigkeitsvorteile in der Testausführung erzielt werden. Zwar gewährleistet WDIO-WORKFLO durch das Caching von Page Nodes einen schonenderen Umgang mit Ressourcen, jedoch fällt dieser in der Geschwindigkeit der Testausführung kaum ins Gewicht. Die Zeitersparnis aufgrund der „intelligenten“ Gruppen-Selektoren von WDIO-WORKFLO, wie etwa der Funktion `firstBy()` der Klasse `PageElementList`, kann auch außerhalb von WDIO-WORKFLO ohne bedeutenden Mehraufwand nachgestellt werden, indem stattdessen direkt Selektoren für einzelne Elemente verwendet werden. Ihre Rolle im Framework dient somit eher zu Abstraktionszwecken, denn sie vermitteln, dass auf diese Weise angesprochene Elemente Teil einer Sammlung gleichwertiger Elemente sind.

Anfänglich sollte die Testausführungsdauer zudem dadurch verkürzt werden, dass in einem Testcase mehrere unterschiedliche Specs validiert werden können und somit bei gesteigertem Umfang einzelner Testcases insgesamt weniger Testcases benötigt werden. Bis zu einem gewissen Grad traf diese Annahme zu. Allerdings stellte sich heraus, dass ein zu umfangreicher Testcase auch einen entscheidenden Nachteil hat: Wird dieser aufgrund eines Fehlers vorzeitig beendet, können nämlich alle Specs, die im Code erst nach der Abbruchstelle verknüpft sind, gar nicht validiert werden. Es gilt also, einen Kompromiss zwischen Testcase-Umfang und Testausführungsdauer zu finden, der je nach konkretem Fall unterschiedlich stark in eine der beiden Richtungen ausschlagen kann.

Ebenso von Anfang an geplant war, in der mit Hilfe von WDIO-WORKFLO betesteten Webanwendung für Angebotserstellung und -verwaltung nächtliche Regressionstests durchführen zu können, um regelmäßig Auskunft über den Zustand der Applikation zu erhalten. Diesem Anspruch wurde WDIO-WORKFLO gerecht und mit der Zeit stellte sich heraus, dass die Vorteile des zuvor geforderten, schnelleren Feedbacks an die EntwicklerInnen auch durch ein nächtliches Feedback ohne große Abstriche realisiert werden konnten. Zwar ist es im Allgemeinen kostengünstiger, Fehler im Software-Code so früh wie möglich aufzudecken und zu beheben, da den EntwicklerInnen die betroffenen Code-Passagen dann noch besser im Gedächtnis sind und sie sich in den Code nicht erst erneut „einarbeiten“ müssen, um Fehler in diesem zu korrigieren. Allerdings zeigte sich, dass den EntwicklerInnen auch am folgenden Morgen, an dem die Resultate der nächtlichen Testläufe analysiert wurden, noch der größte Teil der von Fehlern betroffenen Code-Passagen in Erinnerung war und der Gedächtnisverlust bis zum folgenden Morgen nicht bedeutend ins Gewicht fiel.

Abschließend soll in diesem Unterkapitel auf die Erwartungen an Selenium eingegangen werden. Da Selenium bereits seit vielen Jahren entwickelt wird, weitverbreitet im Einsatz ist und eine große Community hinter sich versammelt hat, wurde davon ausgegangen, dass Selenium stabil und zuverlässig arbeitet. Aus diesem Grund wurde Selenium auch als technologische Basis für die Testautomatisierung mittels WDIO-WORKFLO ausgewählt. Auch wenn sich Selenium im praktischen Einsatz größtenteils bewähren konnte, wurden die Erwartungen an das Testwerkzeug teilweise enttäuscht. Selbst nach mehrjährigem, weitverbreiteten Einsatz bestehen in vereinzelten Fällen Kompatibilitätsprobleme mit unterschiedlichen Browsern und andere Fehler im Testwerkzeug. So funktionierte etwa der Wechsel zwischen mehreren Browser-Fenstern nicht zuverlässig – manchmal wurden Klicks nach dem Wechsel immer noch an das zuvor aktive Fenster weitergeleitet. Das Problem konnte nur durch simulierten Druck der „Tabulator“ Taste nach dem Fensterwechsel umgangen werden.

Dies zeigt auch einen der größten Nachteile von Open Source Lösungen: Es gibt zumeist keinen garantierten Support und im Falle von Fehlern in der Open Source Lösung sollte man in der Lage sein, diesen selbstständig auf den Grund zu gehen und sie zu beheben.

3.1.4 Diskussion der Resultate im Bereich Testanalyse

Wie bereits angesprochen war das vorrangige Ziel in der Phase der Testanalyse, Traceability zwischen Testcases und Specs zu gewährleisten, um ohne große Anstrengungen feststellen zu können, ob und welche Anforderungen durch die in den Tests aufgedeckten Fehler von der Anwendung nicht mehr erfüllt wurden. Dadurch sollten die Kosten für die Fehleranalyse beträchtlich gesenkt werden.

Dieses Ziel konnte vollständig, den Erwartungen entsprechend, umgesetzt werden. Darüber hinaus konnte die Testanalyse auch in weiteren Bereichen verbessert werden, die anfänglich gar nicht vorgesehen waren.

Dazu zählen etwa die Screenshots und Stacktraces, die bei aufgetretenen Fehlern automatisch dem Allure Report hinzugefügt werden. In der Praxis genügte oftmals die bloße Betrachtung des Screenshots, um ein Problem in der graphischen Benutzeroberfläche zu identifizieren und zu beschreiben. Aufwände für zusätzliche Analysetätigkeiten konnten damit eingespart werden.

In Fällen, in denen die alleinige Betrachtung des Screenshots nicht genug Informationen zur Beschreibung eines Defekts bereitstellte, konnte außerdem durch die vollständige Ausgabe aller Testschritte mitsamt deren Input-Parametern und Resultaten im Allure Report die Nachvollziehbarkeit und Reproduzierbarkeit der Testfälle beträchtlich erhöht werden. Die dadurch ermöglichte, manuelle Nachstellbarkeit fehlerhafter Tests erlaubte in vielen Fällen eine gezieltere Untersuchung des Fehlverhaltens, ohne den automatisierten Testfall erneut ausführen zu müssen, wodurch wiederum Kosten in der Fehleranalyse reduziert werden konnten.

3.1.5 Diskussion des Einrichtens der Testumgebung

Ein Nebenaspekt in der Entwicklung von WDIO-WORKFLO war zudem, die Aufwände für das Einrichten der Testumgebung für zukünftige Projekte so gering wie möglich zu halten.

Dadurch, dass WDIO-WORKFLO all seine Abhängigkeiten bereits den eigenen Bedürfnissen entsprechend modifiziert sowie aufeinander abgestimmt hat, und diese im Rahmen eines einzelnen Pakets, welches über den NodeJS Package Manager npm bezogen werden kann, zur Installation zur Verfügung stellt, konnte der Aufwand für die Auswahl, Abstimmung und Installation der einzelnen Bestandteile der Testumgebung für künftige Projekte tatsächlich auf ein Mindestmaß reduziert werden. Zudem trugen neben der Installation mittels eines Konsolen-Einzeilers auch die einfache Konfigurierbarkeit anhand einer einzigen Konfigurationsdatei, sowie die Tatsache, dass WDIO-WORKFLO auch die gesamte Verzeichnisstruktur für die Tests von selbst anlegt, zu einer Verringerung der Kosten für das Einrichten der Testumgebung bei.

Auch ein Wechsel von Betriebssystem oder Browser stellt dank WDIO-WORKFLO keine zusätzlichen Aufwände dar, denn das Framework ist sowohl mit Windows, Linux als auch MacOS „out-of-the-box“ kompatibel und der für die Testausführung verwendete Browser kann über die simple Anpassung eines Zeileneintrags in der Konfigurationsdatei geändert werden.

3.1.6 Weiterführende Fragestellungen

In Kapitel 2.5.1 wurde darauf hingewiesen, dass die Evaluierung des entwickelten Testframeworks nicht auf gemessenen Werten, sondern logischen Rückschlüssen zu Qualität und Kosten der funktionalen Systemtests auf Basis von Aussagen in der Fachliteratur beruht. In Anbetracht des vorwiegend quantitativen Charakters dieser Arbeit mit knapp 70 unterschiedlichen funktionalen Framework-Anforderungen in 10 verschiedenen Themenbereichen, 8 betrachteten Qualitätskriterien und einer Vielzahl an kostensenkenden Faktoren war eine tatsächliche Messung all dieser Aspekte im zeitlichen Rahmen dieser Masterarbeit nicht möglich. Um dennoch Ergebnisse auf Basis wissenschaftlich fundierter Fakten zu erlangen, wurde daher die logische Rückführung auf Aussagen in der Fachliteratur als Evaluierungsstrategie ausgewählt.

Somit können im Rahmen dieser Arbeit zwar logisch begründbare Qualitätssteigerungen und Kostensenkungen nachgewiesen, jedoch keine exakte Quantifizierung dieser Vorteile vorgenommen werden. Gerade jedoch, um die Fragestellung zu klären, in welchem Ausmaß sich der Einsatz von WDIO-WORKFLO wirtschaftlich rentiert, und zu beziffern, nach welcher Zeitspanne/nach wie vielen Testläufen sich der initiale Aufwand für die Automatisierung der funktionalen Systemtests mithilfe von WDIO-WORKFLO amortisiert, ist eine Quantifizierung der bisher nur logisch begründeten Vorteile erforderlich.

Da eine solche im zeitlichen Rahmen dieser Arbeit nicht realisiert werden konnte, empfiehlt sich die Durchführung einer Studie, bei der die Auswirkungen eines Einsatzes von WDIO-WORKFLO anhand unterschiedlicher Projekte über einen längeren Zeitraum gemessen werden. Diese Projekte sollten zuvor bereits manuelle oder mittels anderweitigen Open Source Lösungen automatisierte funktionale Systemtests durchgeführt und relevante Metriken erhoben haben, um über Vergleichsdaten zu verfügen.

3.2 Empirische Empfehlungen

Im Rahmen der Verwendung von WDIO-WORKFLO zur Durchführung der automatisierten funktionalen Systemtests einer Webapplikation für Angebotserstellung und -verwaltung konnte über mehrere Monate hinweg eine Vielzahl an empirischen Erfahrungen gesammelt werden, wie das Testframework in der Praxis am sinnvollsten und nützlichsten einzusetzen ist. Diese Erfahrungen sollen den LeserInnen keinesfalls vorenthalten bleiben und werden daher durch die Formulierung von „Best-Practices“ in diesem Kapitel weitergegeben.

Die erste Empfehlung betrifft das Management von Testdaten: Generell stellt sich die Frage, ob diese über die Aktionsmöglichkeiten der graphischen Benutzeroberfläche angelegt werden sollen oder ob sie bereits vor Beginn der eigentlichen Testausführung in der Datenbank angelegt werden. Als guter Mittelweg hat sich erwiesen, für Überprüfungen, die den Zustand der Testdaten nicht verändern, Testdaten bereits vor der Testausführung in der Datenbank einzuspielen, um in der Testausführung Zeit zu sparen. Wichtig ist hierbei jedoch, dass diese Daten während den Tests ausschließlich für Abfragen, aber nicht für Schreibvorgänge verwendet werden, um zu vermeiden, dass Fehler bei der Manipulation der Daten sich auf spätere Testfälle, die auf dieselben Daten zugreifen, auswirken.

Testdaten, die während der Testausführung verändert werden müssen, sollten jedoch stets zu Beginn eines Testfalls über die graphische Benutzeroberfläche selbst angelegt und nicht in mehreren Testfällen wiederverwendet werden, um die Unabhängigkeit von Testfällen zu gewährleisten.

Bei der Formalisierung der ursprünglichen Anforderungen durch Specs gilt es, mehrere Dinge zu beachten:

Zuerst empfiehlt es sich, vor oder begleitend zur Beschreibung der Specs einen visuellen Prototyp der graphischen Benutzeroberfläche zu erstellen, und diesen als Grundlage für die Abfolge der einzelnen Schritte innerhalb einer Spec zu verwenden. Dadurch werden die einzelnen Zustände und Zustandsänderungen für alle Stakeholder wesentlich einfacher vorstellbar und Probleme im Design der graphischen Oberfläche können so eher frühzeitig aufgezeigt werden. Auch Rupp [18, p. 315] unterstützt eine derartige Vorgehensweise und streicht die Bedeutung visueller Prototypen heraus.

Wichtig ist zudem, die Specs vor der eigentlichen Implementierung der getesteten Software zu verfassen. Wie in Anforderung FS1 näher aufgeführt wird, sollen dadurch Fehler in den ursprünglichen Anforderungen so früh wie möglich aufgedeckt werden, denn in diesem Stadium sind die Kosten für die Behebung dieser Fehler noch vergleichsweise gering. Zudem sollten nicht bloß TestautomatisiererInnen, sondern auch fachliche Verantwortliche, ProjektleiterInnen und EntwicklerInnen in den Prozess der Spec-Formulierung miteingebunden werden, denn sie alle sind für bestimmte Bereiche einer hochqualitativen Anforderungsgestaltung von besonderer Bedeutung, wie Abbildung 75 vermittelt:

	rechtliche Fachliche Richtigkeit	Auftrag des Projektes / Notwendigkeit	Verständlichkeit, gute Struktur / Vollständigkeit	Eindeutigkeit, Widerspruchsfreiheit / Konsistenz	Traceability / Nachvollziehbarkeit	Realisierbarkeit	Testbarkeit	Kritikalität
Anwender	+	+	++	++	0	0	0	0
Fachlich Verantwortlicher	++	+	0	0	0	0	0	++
Analytiker	0	0	++	+	++	++	+	+
Projektleiter	+	++	0	+	0	0	0	+
Entwickler	0	0	0	0	0	0	++	0
Tester	0	0	0	+	+	0	0	++
Projektexterne Auditoren	0	0	++	+	+	+	0	+

Abbildung 75: Prüfende Personen bei Anforderungen [18, p. 309]

So können EntwicklerInnen die Realisierbarkeit bestimmter Anforderungen am besten einschätzen und ProjektleiterInnen die Notwendigkeit einer Anforderung am besten beurteilen. ProjektleiterInnen, die in der agilen Entwicklung oft in der Rolle des „Product Owners“ auftreten, sind zudem in ständigem Kontakt mit den fachlichen Verantwortlichen des Projekts und können in Zusammenarbeit mit diesen Aussagen zur fachlichen Richtigkeit und zur Kritikalität von Anforderungen treffen. TestautomatisiererInnen werden in diesem Beispiel in einer Doppelrolle bestehend aus AnalytikerIn und TesterIn betrachtet. Sie sind in besonderem Maße dazu befähigt, die Vollständigkeit, Verständlichkeit, Eindeutigkeit, Widerspruchsfreiheit sowie Konsistenz und vor allem die Testbarkeit von Anforderungen zu bewerten.

Zusätzlich sollten gezielt Reviews des Designs der getesteten Software durchgeführt werden, da diese gemäß Jones [16, p. 478] am besten zum Auffinden von Design-Fehlern geeignet sind, während sich WDIO-WORKFLO vor allem auf das Aufdecken von Programmier- und Anforderungsfehlern konzentriert.

Beim Testfallentwurf gilt es, einen Kompromiss zwischen dem Umfang der innerhalb eines Testcases validierten Specs und der Ausführungsdauer des Testcases zu finden. Zwar lassen sich die Anzahl der insgesamt benötigten Testcases und damit die Durchlaufzeit der gesamten Testsuite reduzieren und außerdem Zweideutigkeiten und Redundanzen in den Anforderungen leichter erkennen, wenn eine Vielzahl fachlich nah verwandter Anforderungen im selben Testcase validiert wird, allerdings steigt damit auch die Gefahr, dass bei einem vorzeitigen Fehlschlag dieses Testfalls ebendiese Anforderungen gar nicht erst überprüft werden können. Ein allgemein gültiger Richtwert, wie viele Specs durch einen Testcase validiert werden sollten, lässt sich nicht nennen. Stattdessen ist das Finden eines sinnvollen Kompromisses ein Prozess, der sich mit steigender Erfahrung im Umgang mit dem Testframework ständig verbessert.

Um zudem eine hohe Zuverlässigkeit und Robustheit der Testcases zu gewährleisten, sollte beim Einsatz von Wartebedingungen nicht sparsam vorgegangen und keine zu restriktiven Timeouts gesetzt werden. Etwas langsamere Ausführungszeiten sollten in jedem Fall knapp bemessenen Timeouts, die aufgrund eines vollständigen Verstreichens zum Fehlschlag eines Testcases führen können, vorgezogen werden.

Eine letzte Empfehlung hat sich bei der Durchführung automatisierter funktionaler Systemtests mithilfe von WDIO-WORKFLO zudem stets als hilfreich erwiesen: Steps sollten möglichst immer in feingranularen und grobgranularen Varianten verfasst werden. Ein grobgranularer Step könnte den kompletten Login-Vorgang als Admin-User kapseln und damit den Aufruf und das Warten auf das vollständige Laden der Login Seite, den Eintrag von Usernamen und Passwort, den Klick des Login-Buttons, sowie das Warten auf das Laden der Folgeseite nach dem Login umfassen. Dieser Step würde wohl in den meisten Testcases als initialer Schritt wiederverwendet werden. Ein feingranularer Step hingegen dient etwa der Überprüfung einzelner Schritte im Login-Vorgang selbst, und könnte sich somit nur auf die Eingabe von Username und Passwort sowie das Klicken des Login-Buttons beschränken. Im Anschluss an diesen Step könnte bei fehlerhaften Login-Daten festgestellt werden, ob eine Fehlermeldung auf den gescheiterten Login-Versuch hinweist und im Falle von gültigen Login-Daten geprüft werden, ob die auf die Login-Maske folgende Seite auch tatsächlich angezeigt wird.

Auch für die Rolle, welche die automatisierten funktionalen Systemtests im gesamten Qualitätssicherungsprozess ausüben, kann eine Empfehlung ausgesprochen werden: Da funktionale Systemtests im Vergleich zu niedrigeren Teststufen wie Komponententest und Integrationstest komplexer sind, sollten sie eher zu Validierungszwecken und nicht zum Finden neuer Fehler eingesetzt werden. Das bedeutet, dass sich die funktionalen Systemtests gut dazu eignen, die Einhaltung der Anforderungen an die getestete Anwendung zu überprüfen, aber nicht dazu, möglichst gezielt neue Fehler in der Anwendung aufzudecken. Letzteres kann nämlich durch exploratives Testen oder automatisierte Komponenten- und Integrationstests wesentlich kostengünstiger bewerkstelligt werden. Daher weisen Dustin et al. [7] darauf hin, dass vor den automatisierten funktionalen Systemtests zumindest auch Komponententests durchgeführt und die niedrigeren Teststufen auf keinen Fall vernachlässigt werden sollten.

Zusätzlich zu den mittels WDIO-WORKFLO automatisierten funktionalen Systemtests sollten außerdem auch die nicht-funktionalen Anforderungen der betesteten Webanwendung, welche in Kapitel 2.2.1.4 definiert sind, in der Teststufe der Systemtests überprüft werden. Da WDIO-WORKFLO jedoch nicht gezielt auf das Testen von Benutzbarkeit, Sicherheit und Lastverhalten ausgelegt ist, bietet sich hierbei die Benutzung alternativer, auf diese Zwecke maßgeschneiderter Testwerkzeuge an.

Im Allgemeinen sollte mit dem Testen der Software so früh wie möglich begonnen werden, um von Anfang an eine hohe Qualität der Anwendung sicherzustellen und eine Explosion der Fehlerkosten durch „Verschleppung“ der Behebung zu vermeiden. Diese Einschätzung wird auch in der Fachliteratur geteilt, etwa von Spillner et al. [1, p. 34].

Schließlich liegt es nahe, die Durchführung der automatisierten funktionalen Systemtests in den Continuous Development Prozess einzubinden, und etwa jede Nacht einen Build der Software mit anschließender Testausführung vorzunehmen. Dadurch ist stets ein Überblick über den Zustand der entwickelten Anwendung möglich und aufgrund des regelmäßigen Feedbacks kann zeitnah auf entdeckte Fehler in der Software reagiert werden.

Zuletzt erwies es sich als praktisch, die Umgebungen, in denen die automatisierten funktionalen Systemtests im Browser ausgeführt werden, sowie jene, in denen die getestete Anwendung und alle benötigten Services laufen, in einem „Virtual Machine Image“ oder einem „Docker Container“ zu „konservieren“, damit diese mit geringem Aufwand auf unterschiedliche Hardware übertragen werden können. Für die Installationsumgebung von WDIO-WORKFLO selbst ist ein solches Vorgehen jedoch nicht unbedingt erforderlich, da das Framework dank NodeJS' Package Manager „npm“ sehr schnell in unterschiedlichsten Betriebssystemen installiert werden kann.

3.3 Resümee und Erfüllung der Aufgabenstellung

In Kapitel 1.4 wurde die Entwicklung eines Automatisierungs-Frameworks, welches den mit funktionalen Systemtests verbundenen Aufwand senkt, jedoch gleichzeitig die Qualität dieser Tests und des gesamten Softwareprodukts erhöht, als Ziel dieser Arbeit definiert. Das vorliegende Kapitel legt nunmehr dar, ob und in welchem Ausmaß dieses Ziel erreicht werden konnte und fasst dazu die wichtigsten Ergebnisse, sowie Erkenntnisse über die gewählte Vorgehensweise, in aller Kürze zusammen.

In der vorliegenden Arbeit zeigte sich, dass durch eine Steigerung der Qualität von funktionalen Systemtests, Anforderungen und der getesteten Anwendung mithilfe von WDIO-WORFKLO initial zwar ein Mehraufwand anfällt, langfristig gesehen jedoch eine Reduktion der Kosten für die Durchführung funktionaler Systemtests sowie eine Senkung des gesamten Projektaufwands erreicht werden kann. Dieses Ergebnis lässt sich vor allem durch folgende Maßnahmen und Features von WDIO-WORKFLO begründen:

Sämtliche Komponenten und externen Libraries, die in WDIO-WORFKLO zum Einsatz kommen, sind gezielt aufeinander abgestimmt, den Bedürfnissen des Frameworks entsprechend angepasst und über ein einzelnes, einfach installierbares und schnell konfigurierbares npm-Paket zugänglich gemacht. Die Aufwände für das Aufsetzen und die Konfiguration der Testumgebung werden somit auf ein Minimum reduziert.

Ein großer Fokus von WDIO-WORKFLO liegt zudem auf der Verifizierung der Anforderungen mit Hilfe einer eigenen Formalisierungsschicht, welche die Anforderungen in Gherkin-Language (siehe Kapitel 2.4.1) wiedergibt und damit eine natürlichsprachliche Ausdrucksweise mit logisch überprüfbaren Kriterien kombiniert. Dieses Vorgehen fördert die Korrektheit, Vollständigkeit und Konsistenz der Anforderungen. Die oftmals besonders schwerwiegenden, in Anforderungen vorhandenen Fehler und Unklarheiten können somit bereits in der Phase der Anforderungsdefinition erkannt und die Kosten, die diese verursachen, so gering wie möglich gehalten werden. Eine Trennung logischer und konkreter Testfälle anhand von „Specs“ und „Testcases“ erlaubt es, mit dem Testfallentwurf bereits vor dem Start der Anwendungsimplementierung zu beginnen und den EntwicklerInnen damit eine Hilfestellung in Form fertig ausformulierter, logischer Testfallbeschreibungen zur Verfügung zu stellen.

Durch die Gewährleistung von Traceability zwischen formalisierten Anforderungen und Testfällen, aber auch zwischen Testfällen und den durch diesen aufgedeckten Fehlern, sowie zwischen formalisierten und ursprünglichen Anforderungen (etwa in Form von User Stories), können die Analysierbarkeit der funktionalen Systemtests gesteigert und damit die Kosten für Fehler- und Änderungsanalyse entscheidend verringert werden.

Die Aufwände für die Testfallentwicklung werden ebenso deutlich gesenkt. Wichtigster Faktor sind hierbei die in hohem Grade wiederverwendbaren Komponenten „Steps“ und „Page Objects“. „Steps“, welche Testbefehle zu in natürlicher Sprache formulierten Schritten zusammenfassen, sowie „Page Objects“, welche die Logik und Struktur einer Webseite hinter einer applikationsspezifischen API kapseln, erhöhen die Verständlichkeit und Erlernbarkeit der Testfälle. Zudem wirken sich beide Komponenten positiv auf die Änderbarkeit der Testfälle aus, da Anpassungen nur an einer einzigen Stelle vorgenommen werden müssen. Neben dem Einsatz von „Steps“ und „Page Objects“ trägt die Anwendung bewährter Architekturprinzipien wie hoher Kohäsion, geringer Kopplung, „Separation of Concerns“, Abstraktion sowie den „SOLID“ Prinzipien der Softwareentwicklung zu einer Erhöhung der Änderbarkeit und Analysierbarkeit der Testfälle bei, was deren Wartbarkeit signifikant verbessert. In Anbetracht der Tatsache, dass Wartungskosten den Großteil der Gesamtkosten eines Softwareprojektes ausmachen, können allein durch die dank WDIO-WORKFLO erhöhte Wartbarkeit äußerst umfangreiche Kostenreduktionen erzielt werden. Zudem trägt auch die Verwendung von TypeScript zu einer kostengünstigeren Testfallentwicklung bei, denn Fehler im Test-Code können dank „static checking“ bereits während der Testfallimplementierung in der Entwicklungsumgebung angezeigt werden und gelangen daher erst gar nicht in die Testsuite. Zusätzlich erlauben die „IntelliSense“ Features von Visual Studio Code („Auto Complete“, „Alle Verweise suchen“, „Parameter- und Definitions-Preview“), die dank TypeScript zur Gänze ausgeschöpft werden können, eine wesentlich komfortablere und effizientere Entwicklungserfahrung.

Dank der Automatisierung der funktionalen Systemtests ist gegenüber manueller Testausführung ein weitaus häufigeres Feedback an die EntwicklerInnen und ein ständiger Überblick über die Qualität der Software möglich. Dies resultiert in einer geringeren Fehleranzahl in der Anwendung und damit in einer höheren Softwarequalität, wodurch eine Minderung der Folgekosten qualitativ mangelhafter Software möglich wird.

Schließlich bleiben dank der automatischen Generierung von Testspezifikation und Testabschlussbericht zusätzliche Aufwände für das Reporting erspart. Die anhand von Diagrammen graphisch aufbereiteten Testresultate erlauben eine rasche Aussage über den Zustand der getesteten Anwendung. Durch die genaue Auflistung aller durchgeführten Schritte mitsamt deren Eingabeparametern und Ergebnissen, sowie Screenshots und Stacktraces im Fehlerfall, sind eine lückenlose Nachvollziehbarkeit und Reproduzierbarkeit der Testfälle möglich und die Kosten für die Analyse von Fehlern werden gesenkt.

Die geplante Vorgehensweise, die spezifischen Eigenschaften automatisierter funktionaler Systemtests zunächst durch eine ausführliche Literaturrecherche zu eruieren, daraus funktionale Anforderungen und Qualitätskriterien an das Testframework abzuleiten und deren Erfüllung durch konkrete Maßnahmen in dem Design und der Implementierung von WDIO-WORKFLO zu gewährleisten, konnte ohne größere Probleme eingehalten werden.

Einzig die schiere Menge an Anforderungen, welche durch die große Anzahl an Themenbereichen, die für WDIO-WORKFLO von Relevanz sind, erklärt werden kann, stellte bei der Evaluierung ein Hindernis dar, da der zeitlich begrenzte Rahmen dieser Masterarbeit keine Messung aller funktionalen Anforderungen, Qualitätskriterien und Kostenauswirkungen erlaubte.

Um in der Evaluierung dennoch zu wissenschaftlich begründeten Ergebnissen zu gelangen, wurden anhand von Aussagen und Forschungsergebnissen der Fachliteratur logische Rückschlüsse auf die Auswirkungen der umgesetzten Anforderungen auf die Qualität und die Kosten funktionaler Systemtests und der gesamten Software gezogen.

Insgesamt kann die Aufgabenstellung somit als zum Großteil erfüllt betrachtet werden. Allerdings sollten die tatsächlichen Auswirkungen des Einsatzes von WDIO-WORKFLO zur automatisierten Durchführung funktionaler Systemtests auf Qualität und Kosten anhand von längerfristig in mehreren Projekten erhobenen „Feldstudien“ gemessen und damit quantifiziert werden, um die logischen Folgerungen zu bestätigen oder zu widerlegen.

3.4 Ausblick und Schlussfolgerung

In diesem letzten Kapitel sollen anstelle der bereits ausführlich behandelten Qualitätssteigerungen und Kostensenkungen durch den Einsatz von WDIO-WORKFLO Auswirkungen des Frameworks betrachtet werden, welche nicht unmittelbar im Fokus dieser Arbeit stehen. Allen voran werden hierzu die sozialen Aspekte und die Folgen des Framework-Einsatzes für die betroffenen Stakeholder auf menschlicher Ebene hervorgehoben. Des Weiteren werden potenzielle Anknüpfungspunkte an diese Arbeit sowohl auf technischer als auch auf wissenschaftlicher Seite aufgezeigt und der Stellenwert von WDIO-WORFKLO für die Automatisierung funktionaler Systemtests herausgestrichen.

In dem Projekt, in dessen Rahmen die Webapplikation für Angebotserstellung und -verwaltung entwickelt wurde, und dessen funktionale Systemtests mithilfe von WDIO-WORKFLO automatisiert wurden, konnte etwa eine Veränderung der Arbeitsprozesse beobachtet werden: Anforderungen wurden nun bereits deutlich früher durch TestautomatisiererInnen kritisch hinterfragt und auch außerhalb der funktionalen Systemtests wurde ein verstärkter Fokus auf eine testbare Formulierung von Anforderungen mittels Akzeptanzkriterien festgestellt.

Insgesamt schien es, als würde das Vertrauen in die Webapplikation aufgrund der nächtlich ausgeführten, automatisierten funktionalen Systemtests bei allen Projektbeteiligten erhöht. So gingen Product Owner und leitende EntwicklerInnen mit ruhigerem Gewissen zu Abnahmetermeninen mit den AuftraggeberInnen, da sie stets über den aktuellen Zustand der Software Bescheid wussten. Und gerade die EntwicklerInnen der Software nahmen das regelmäßige Feedback als Mittel zur „Selbstreflektion“ ihrer Leistungen dankbar an, denn dort wo viel Herzblut in die Entwicklung eines Produktes fließt, wird eine reibungslose Funktionalität desselben besonders energisch verfolgt.

Beim Autor dieser Arbeit setzte darüber hinaus eine tiefgreifende innere Reflektion über die Art und Weise, wie Wissensvermittlung durch den Programm-Code gefördert werden kann, ein. Das Problem fehlender Wissensvermittlung tritt vor allem dann auf, wenn bestimmte Code-Stellen nach längerer Zeit angepasst werden oder wenn EntwicklerInnen, die nicht mit der Anwendung vertraut sind, mit dieser arbeiten müssen. In dieser Hinsicht stellte sich heraus, dass TypeScript viel mehr Vorteile bietet als bloße Typensicherheit und „static checking“. Typdefinitionen, Interfaces und IntelliSense Features wie das Anzeigen aller auf einem Objekt definierten Properties mitsamt deren Typen in einem Vorschaufenster entpuppten sich als Mittel, um das Wissen über den Aufbau und die Funktionsweise einer Anwendung aus den Köpfen der EntwicklerInnen hinaus in den Code selbst zu verlagern und den Prozess der (erneuten) Wissensaneignung reibungslos und kurzweilig zu gestalten. Nach Meinung des Autors zeichnen sich gute EntwicklerInnen nämlich gerade dadurch aus, dass sie ihr Wissen (über die Anwendung) nicht für sich behalten, sondern dieses so breit und verständlich wie möglich nach außen hin kommunizieren und zwar vor allem auch durch den geschriebenen Programm-Code.

Auch wenn im Rahmen dieser Arbeit bereits eine Version von WDIO-WORKFLO entwickelt werden konnte, die einen Großteil der an ein Framework zur Automatisierung funktionaler Systemtests gestellten Anforderungen überaus zufriedenstellend erfüllt, so gibt es ohne Zweifel noch Raum für Verbesserungen.

Besonders die fehlende Dokumentation des Frameworks beeinträchtigt derzeit die Erlernbarkeit und Verständlichkeit. Es ist daher vorgesehen, in naher Zukunft eine vollständige und ausführliche Dokumentation für WDIO-WORKFLO zu erstellen.

Zudem wird eine eindeutiger Trennung zwischen Specs und Testcases in den von Allure generierten Reports angestrebt. Der Umstand, dass Allure von Haus aus nicht zwischen diesen unterscheidet, führte dazu, dass zum jetzigen Zeitpunkt alle Diagramme in den Allure Reports sowohl Testcases als auch Specs als „Testfälle“ bezeichnen, was die Analysierbarkeit der Testresultate erschwert. Deswegen soll künftig nach einem Weg gesucht werden, die Erstellung der Allure Reports tiefergehender zu beeinflussen.

Schließlich wurden, wie bereits mehrfach erwähnt, die Auswirkungen eines Einsatzes von WDIO-WORKFLO auf die Qualität und die Kosten der funktionalen Systemtests, sowie des gesamten Softwareproduktes, durch logisch begründbare Rückschlüsse auf Basis von Aussagen der Fachliteratur hergeleitet. Dadurch lässt sich die Verwendung von WDIO-WORKFLO zwar bereits vernunftmäßig argumentieren, allerdings fehlen quantitative Anhaltspunkte, die den mit der Qualitätssicherung betrauten Entscheidungsträgern ermöglichen, die Größenordnungen, mit welchen hinsichtlich Qualitätssteigerungen und Kostensenkungen gerechnet werden kann, einzuschätzen. Um hierfür wissenschaftlich relevante Daten zu liefern, wird eine langwierig angelegte Feldstudie vorgeschlagen, in welcher WDIO-WORKFLO in mehreren unterschiedlichen Software-Projekten zur Automatisierung funktionaler Systemtests eingesetzt wird. Dabei sollen die Auswirkungen von WDIO-WORKFLO auf Qualität und Kosten der entwickelten Software und im speziellen der funktionalen Systemtests selbst anhand ausgewählter Metriken gemessen werden. Die betrachteten Projekte sollten bereits in der Vergangenheit Daten über Qualität und Kosten der produzierten Software und deren Qualitätssicherungsmaßnahmen erhoben haben, welche als Vergleichswerte für die Messungen der durch den Einsatz von WDIO-WORKFLO erzielten Veränderungen herangezogen werden können.

Alles in allem wurde durch die Entwicklung von WDIO-WORKFLO eine budgetschonende Alternative zu kostenpflichtigen Testwerkzeugen für die Automatisierung funktionaler Systemtests geschaffen. Zwar existieren gerade im NodeJS Ökosystem bereits einige Open Source Libraries, die für einzelne Aufgaben funktionaler Systemtests wie Testausführung, Assertions und Reporting genutzt werden können, doch in WDIO-WORKFLO wurden diese in einem einzigen „Plug & Play“ Paket kombiniert und gezielt aufeinander abgestimmt, sowie um einen typischeren Core erweitert und das Testframework dadurch auf die spezifischen Eigenschaften und Anforderungen automatisierter funktionaler Systemtests zurechtgeschnitten, was sich in einer Erhöhung der Qualität und einer Senkung der Kosten funktionaler Systemtests sowie der gesamten Software niederschlägt.

Gegenüber bestehenden Open Source Lösungen im NodeJS Ökosystem wurden zudem eine eigene Formalisierungsschicht zur Anforderungsgestaltung und -verifizierung, Traceability zwischen Anforderungen und Testfällen sowie eine Architektur mit großem Fokus auf Benutzbarkeit und Wartbarkeit als Neuerungen durch WDIO-WORKFLO eingeführt. Vorteile gegenüber kostenpflichtigen Testwerkzeugen umfassen die durch den frei zur Verfügung stehenden Quellcode gewonnene Unabhängigkeit vom Hersteller des Testwerkzeugs sowie die Möglichkeit, den Code des Frameworks an die speziellen Bedürfnisse eines Softwareprojektes anzupassen.

Insgesamt erlaubt es WDIO-WORKFLO vor allem kleineren Firmen und Softwareprojekten, ihre hohen Qualitätsansprüche nicht aufgrund eines beschränkten Budgets hintanstellen zu müssen und gleicht dadurch deren Wettbewerbsnachteil gegenüber finanziell potenteren Konkurrenzanbietern, welchen oftmals eigene Testabteilungen und teure Testwerkzeuge zur Verfügung stehen, zumindest teilweise wieder aus.

Literaturverzeichnis

- [1] A. Spillner, T. Linz und H. Schaefer, Software Testing Foundations, Santa Barbara, CA 93103: Rocky Nook Inc., 2014.
- [2] I. Sommerville, Software Engineering, 6. Hrsg., Boston, Massachusetts: Pearson Education Limited, 2011.
- [3] G. Meyers, C. Sandler und T. Badgett, The Art of Software Testing, Hoboken, New Jersey: John Wiley & Sons, Inc., 2012.
- [4] K. Franz, Handbuch zum Testen von Web-Applikationen, Berlin Heidelberg: Springer, 2007.
- [5] E. Dustin, T. Garret und B. Gauf, „Chapter 2: Why Automate?“, in *Implementing Automated Software Testing: How to Save Time and Lower Costs while Raising Quality*, Addison-Wesley Professional, 2009, pp. 23-50.
- [6] R. Pressmann, Software Engineering, New York: McGraw-Hill, 2001.
- [7] E. Dustin, T. Garret und B. Gauf, „Chapter 4: Why Automated Software Testing Fails and Pitfalls to Avoid“, in *Implementing Automated Software Testing: How to Save Time and Lower Costs while Raising Quality*, Addison-Wesley Professional, 2009, pp. 69-97.
- [8] M. Emrich, Behaviour Driven Development with JavaScript, Developer.Press.
- [9] Ranorex, „Ranorex“, [Online]. Available: <https://www.ranorex.com/>. [Zugriff am 13 November 2017].
- [10] SmartBear, „Testcomplete“, [Online]. Available: <https://smartbear.com/product/testcomplete/overview/>. [Zugriff am 13 November 2017].
- [11] Tricentis, „Tosca“, [Online]. Available: <https://www.tricentis.com/de/tricentis-tosca-testsuite/>. [Zugriff am 13 November 2017].
- [12] SeleniumHQ, „SeleniumHQ Browser Automation“, [Online]. Available: <http://www.seleniumhq.org/>. [Zugriff am 14 November 2017].

- [13] D. Burns, Selenium 2 Testing Tools Beginner's Guide, 2. Hrsg., Birmingham: Packt Publishing, 2012.
- [14] Innovative Defense Technologies, „Innovative Defense Technologies,“ [Online]. Available: <https://idtus.com/>. [Zugriff am 19 Januar 2018].
- [15] E. Dustin, T. Garret und B. Gauf, „Chapter 3: The Business Case,“ in *Implementing Automated Software Testing: How to Save Time and Lower Costs while Raising Quality*, Addison-Wesley Professional, 2009, pp. 51-68.
- [16] J. Capers, Applied Software Measurement: Global Analysis of Productivity and Quality, 3. Hrsg., New York City: McGraw-Hill, 2008.
- [17] Hochschule Bremerhaven; Hochschule Bremen; Technische Hochschule Köln; German Testing Board; Swiss Testing Board, Softwaretest in Praxis und Forschung, Heidelberg: dpunkt.verlag, 2016.
- [18] C. Rupp, Requirements-Engineering und Management, München Wien: Carl Hanser Verlag, 2007.
- [19] P. Loucopoulos und B. Karakostas, System Requirements Engineering, New York: McGraw-Hill, 1995.
- [20] „WebdriverIO,“ [Online]. Available: <http://webdriver.io/>. [Zugriff am 14 November 2017].
- [21] Pivotal Labs, „Jasmine,“ [Online]. Available: <https://jasmine.github.io/>. [Zugriff am 14 November 2017].
- [22] Allure Team, „Allure Test Report,“ [Online]. Available: <http://allure.qatools.ru/>. [Zugriff am 14 November 2017].
- [23] Facebook Inc., „React,“ [Online]. Available: <https://reactjs.org/>. [Zugriff am 22 November 2017].
- [24] J. Velasco und J. Jiménez, „React Toolbox,“ [Online]. Available: <http://react-toolbox.io/#/>. [Zugriff am 22 November 2017].
- [25] Mozilla, „Introduction,“ [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction#What_is_JavaScript. [Zugriff am 22 November 2017].

- [26] The Linux Foundation, „Node.js,“ [Online]. Available: <https://nodejs.org/en/>. [Zugriff am 27 November 2017].
- [27] The Linux Foundation, „The Node.js Event Loop, Timers, and process.nextTick(),“ [Online]. Available: <https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/>. [Zugriff am 23 November 2017].
- [28] S. Kamani, „How is javascript asynchronous AND single threaded?,“ [Online]. Available: <https://www.sohamkamani.com/blog/2016/03/14/wrapping-your-head-around-async-programming/>. [Zugriff am 23 November 2017].
- [29] International Software Testing Board, Certified Tester Foundation Level Syllabus, Austrian Testing Board, German Testing Board e.V., Swiss Testing Board, 2011.
- [30] G. E. Thaller, Software-Metriken, 2. Hrsg., Berlin: Verlag Technik, 2000.
- [31] F. Witte, Testmanagement und Softwaretest, Wiesbaden: Springer Vieweg, 2016.
- [32] Wikipedia, „Single responsibility principle,“ [Online]. Available: https://en.wikipedia.org/wiki/Single_responsibility_principle. [Zugriff am 07 Dezember 2017].
- [33] „Dependency Inversion Principle,“ [Online]. Available: <http://www.oodesign.com/dependency-inversion-principle.html>. [Zugriff am 24 November 2017].
- [34] ISO/IEC 25010:2011, Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models, 1 Hrsg., ISO/IEC JTC 1/SC 7 Software and systems engineering, 2011.
- [35] The Linux Foundation, „Node.js v8.9.1 Documentation,“ [Online]. Available: <https://nodejs.org/dist/latest-v8.x/docs/api/>. [Zugriff am 23 November 2017].
- [36] C. Bromann, „wdio-spec-reporter,“ [Online]. Available: <https://www.npmjs.com/package/wdio-spec-reporter>. [Zugriff am 8 Dezember 2017].
- [37] Microsoft, „TypeScript,“ [Online]. Available: <https://www.typescriptlang.org/>. [Zugriff am 14 November 2017].
- [38] Microsoft, „Visual Studio Code,“ [Online]. Available: <https://code.visualstudio.com/>. [Zugriff am 8 Dezember 2017].

- [39] M. Fowler, „PageObject,“ [Online]. Available: <https://martinfowler.com/bliki/PageObject.html>. [Zugriff am 10 Dezember 2017].
- [40] Atlassian, „Jira Software,“ [Online]. Available: <https://www.atlassian.com/software/jira>. [Zugriff am 10 Dezember 2017].
- [41] SourceMaking.com, „Facade Design Pattern,“ [Online]. Available: https://sourcemaking.com/design_patterns/facade. [Zugriff am 14 Dezember 2017].
- [42] Refsnes Data, „XPath Tutorial,“ [Online]. Available: https://www.w3schools.com/xml/xpath_intro.asp. [Zugriff am 16 Dezember 2017].
- [43] SourceMaking.com, „Composite Design Pattern,“ [Online]. Available: https://sourcemaking.com/design_patterns/composite. [Zugriff am 17 Dezember 2017].
- [44] SourceMaking.com, „Factory Method Design Pattern,“ [Online]. Available: https://sourcemaking.com/design_patterns/factory_method. [Zugriff am 17 Dezember 2017].
- [45] NetApplications.com, „Browser Market Share,“ [Online]. Available: <https://www.netmarketshare.com/browser-market-share.aspx>. [Zugriff am 27 Dezember 2017].
- [46] E. Dustin, T. Garret und B. Gauf, „Chapter 7: Key 3: Test the Automated Software Test Framework (ASTF),“ in *Implementing Automated Software Testing: How to Save Time and Lower Costs while Raising Quality*, Addison-Wesley Professional, 2009, pp. 167-184.
- [47] J. Capers, „The economics of software maintenance in the twenty first century,“ Capers Jones, 2006.
- [48] RedMonk, „The RedMonk Programming Language Rankings: June 2017,“ [Online]. Available: <http://redmonk.com/sograde/2017/06/08/language-rankings-6-17/>. [Zugriff am 28 Dezember 2017].
- [49] German Testing Board, Basiswissen SoftwareTest Certified Tester - Kapitel 7, 2.1. Hrsg., German Testing Board, 2017.

Abbildungsverzeichnis

Abbildung 1: Allgemeines V-Modell [1, p. 40]	9
Abbildung 2: Abgewandelte Form des V-Modells von Meyers et al. (Quelle: [3, p. 117]) ...	10
Abbildung 3: Automatisierungsraten dynamischer Testarten (Quelle: [17, p. 14])	22
Abbildung 4: Einschätzung der Testintensität (Quelle: [17, p. 17])	23
Abbildung 5: Typische Architektur einer E-Commerce Seite (Quelle: [3, p. 195])	31
Abbildung 6: "Autocomplete"-Komponente von React-Toolbox [24] in der GUI	32
Abbildung 7: Einbindung der "Autocomplete"-Komponente von React-Toolbox [24]	33
Abbildung 8: "Autocomplete"-Komponente von React-Toolbox [24] im DOM	33
Abbildung 9: Blockierende Queue-Abarbeitung (Quelle: [28])	35
Abbildung 10: Nicht-blockierende Abarbeitung am Beispiel Node.js (Quelle: [28])	35
Abbildung 11: ISTQB fundamentaler Testprozess (Quelle: [1, p. 19])	38
Abbildung 12: Fehlerkostenentwicklung fortlaufender Projektphasen (Quelle: [31, p. 88]) ..	58
Abbildung 13: Nichtbeachtung von Dependency Inversion Principle (Quelle: [33])	77
Abbildung 14: Einhaltung des Dependency Inversion Principles (Quelle: [33])	78
Abbildung 15: Skizze der Systemumgebung von "WDIO-WORKFLO"	89
Abbildung 16: WebdriverIO Codebeispiel mit synchronem Testrunner	91
Abbildung 17: WebdriverIO Codebeispiel mit asynchronem Testrunner	91
Abbildung 18: Klassendiagramm der internen Komponenten von WDIO-WORKFLO	93
Abbildung 19: Beispiel von Specs in WDIO-WORKFLO	97
Abbildung 20: Beispiel von Manual Results in WDIO-WORKFLO	99
Abbildung 21: Beispiel von Testcases in WDIO-WORKFLO	100
Abbildung 22: Beispiel von Steps in WDIO-WORKFLO	102
Abbildung 23: IntelliSense unterstützt bei der Auswahl von Steps	102
Abbildung 24: Schließen eines Dialogs ohne Einsatz des Page Object Patterns	104
Abbildung 25: Schließen eines Dialogs unter Einsatz des Page Object Patterns	104
Abbildung 26: Beispiel einer Page in WDIO-WORKFLO	105
Abbildung 27: Schnittstellenfunktionsaufrufe sowie direkter Zugriff auf PageNodes	107
Abbildung 28: Implementierung der Checkbox Komponente von React Toolbox [24]	107
Abbildung 29: Verknüpfung der Selektoren verschachtelter PageElements	110
Abbildung 30: Der XPath-Query-Builder von WDIO-WORKFLO	110
Abbildung 31: Iteration über alle von einer PageElementList verwalteten PageElements ..	111
Abbildung 32: Auswahl eines Listeneintrags über die „firstBy()“ Funktionskette	112
Abbildung 33: Anwendungsbeispiel von PageElementMap für Navigationsbereich	112
Abbildung 34: Zugriff auf mit PageElementMap gemappte PageElements via "\$"	113
Abbildung 35: Verwendung einer PageElementGroup für ein Login-Formular	114
Abbildung 36: "Solve()" Funktion der PageElementGroup Basisklasse	115
Abbildung 37: Die Interfaces "IProblem" und "ISolveResult"	116
Abbildung 38: Aufruf der "ValueGroup"-Funktionen	116

Abbildung 39: Überprüfung von Formularwerten mit und ohne Einsatz von "GetValue()"	117
Abbildung 40: Default-Typen und -Parameter von PageNodes dank PageElementStore	118
Abbildung 41: Vergleich manuell und durch PageElementStore erzeugter PageNodes ..	119
Abbildung 42: Caching von identen PageNodes in PageElementStore	120
Abbildung 43: Vorschläge für Instanziierungsmethoden durch IntelliSense	120
Abbildung 44: Innerhalb von List-Files verfügbare Ausführungsfilter	121
Abbildung 45: "mapToObject()" und "mapProperties()" als Beispiele für Utility Functions	122
Abbildung 46: Der UID Store generiert eindeutige Bezeichner für Testdaten	123
Abbildung 47: Ausgabe von Testcase und Spec Results im BDD-Format	124
Abbildung 48: Der Schlussbereich der Ausgabe des Spec Reporters	125
Abbildung 49: Die Startseite des Allure Reports	126
Abbildung 50: Die Suites-Seite des Allure Reports	127
Abbildung 51: Hierarchisch dargestellte Steps und farblich hervorgehobene Fehler	128
Abbildung 52: Stacktraces und Kopieren des Testcase-Namens in Allure Reports	128
Abbildung 53: An fehlerhafte Steps im Allure Reporter angehängte Screenshots	129
Abbildung 54: History der Ergebnisse einzelner Tests	129
Abbildung 55: Behaviors-Seite von Allure Reports	130
Abbildung 56: Categories Seite von Allure Reports	131
Abbildung 57: Zeitliche Abfolge und Dauer von Tests	131
Abbildung 58: Diagramme für überblicksmäßige Informationsvermittlung	132
Abbildung 59: Die "--help" Option des CLI	136
Abbildung 60: Filterung der Testausführung nach Testcase	136
Abbildung 61: Filterung der Testausführung nach Suiten	136
Abbildung 62: Filterung der Testausführung nach Features	137
Abbildung 63: Kombination mehrerer Filtereinträge mittels Wildcards und Negationen ...	137
Abbildung 64: Ausführungsfilter in WDIO-WORKFLO	137
Abbildung 65: Ausführung von Listen mittels CLI	138
Abbildung 66: Filterung der Testausführung nach Schweregrad	138
Abbildung 67: Anzeige aller länger nicht geprüften manuellen Specs in CLI	138
Abbildung 68: Anzeige des letztbekannten Status mittels "--printStatus"	139
Abbildung 69: Ausgabe von Traceability Informationen in der CLI	140
Abbildung 70: Die Konfigurationsdatei von WDIO-WORKFLO	141
Abbildung 71: Ordnerstruktur von WDIO-WORFKLO	142
Abbildung 72: Eigenschaften von Anforderungs-Prüftechniken [18, p. 319]	150
Abbildung 73: Berechnung der Ersparnisse durch Testautomatisierung (Quelle: [15])	185
Abbildung 74: Relative Kosten, um Fehler zu korrigieren [6, p. 423]	188
Abbildung 75: Prüfende Personen bei Anforderungen [18, p. 309]	199

Tabellenverzeichnis

Tabelle 1: Aufteilung von Fehlern im Entwurf (Quelle: [30, p. 100])	57
Tabelle 2: Im Testframework eingesetzte Werkzeuge	86
Tabelle 3: Beitrag funktionaler Anforderungen zu Qualitätskriterien	177
Tabelle 4: Bewertung der funktionalen Anforderungen	178