Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Einführung und Auswertung des Nutzen-Aufwand-Verhältnisses von automatisierten GUI-Tests

Marcel Graf

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	Dipl.-Ing. Jan-Peter Ostberg, Dr. Ivan Bogicevic, Robert Seidel (AEB), Dipl.-Ing. Uwe Hoell (AEB)
Beginn am:	19. Juli 2017
Beendet am:	19. Dezember 2017
CR-Nummer:	D.2.5

Kurzfassung

Um die Funktionalität von Software sicherzustellen, ist die Prüfung der grafischen Benutzeroberfläche (GUI) als wichtigste Benutzerschnittstelle unerlässlich. Dies wird oft manuell durchgeführt, was einen hohen personellen Aufwand bedeutet. Durch automatisierte GUI-Tests kann dieser Prozess reproduzierbar und günstiger durchgeführt werden. Diese Bachelorarbeit behandelt die Einführung von automatisierten GUI-Tests und wertet das Nutzen-Aufwand-Verhältnis aus. Es werden die Grundlagen für Qualitätssicherung und automatisierte GUI-Tests gelegt. Außerdem werden die technischen Details der Browserautomatisierung betrachtet. Anschließend werden erste Tests für ein gewähltes Produkt der AEB Gesellschaft zur Entwicklung von Branchensoftware mbH (AEB) automatisiert. Folgend wird das Nutzen-Aufwand-Verhältnis für die Testautomatisierung analysiert. Hierbei kommt eine quantitative sowie eine qualitative Analyse zum Einsatz. In der quantitativen Analyse wird der personelle Aufwand zwischen manueller Testdurchführung und Testautomatisierung sowie die zusätzlichen Anforderungen an das Testsystem bewertet. In der qualitativen Analyse wird der nicht direkt messbare Nutzen und Aufwand analysiert. Hierzu werden unter anderem die Auswirkungen auf die Wartbarkeit der Software und die Qualitätssicherung für Releases untersucht. Das Nutzen-Aufwand-Verhältnis zeigt, dass mit einer Amortisierung innerhalb weniger Jahren gerechnet werden kann. Um die Testautomatisierung weiterzuführen werden abschließend Handlungsempfehlungen für die AEB aufgezeigt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AC	Application Controller	31
ACV	Application Controller View	31
AEB	AEB Gesellschaft zur Entwicklung von Branchen-Software mbH	13
AOP	Aspektorientierte Programmierung	32
API	Programmierschnittstelle	22
BDD	Behavior Driven Development	42
BPMN	Geschäftsprozessmodell und -notation	27
CSS	Cascading Style Sheets	30
DOM	Document Object Model	23
GUI	Grafische Benutzeroberfläche	13
PT	Personentag	38
W3C	World Wide Web Consortium	22
XML	Extensible Markup Language	30

1 Einleitung

Dieses Kapitel stellt die Motivation und Ziele der Bachelorarbeit vor. Die Bachelorarbeit entstand im Rahmen einer Zusammenarbeit mit der AEB Gesellschaft zur Entwicklung von Branchen-Software mbH (AEB). Das Unternehmen aus Stuttgart ist seit über 35 Jahren am Markt und ein führender Anbieter globaler IT-Lösungen und Services für Supply-Chain-Management [AEB17a].

1.1 Motivation

Grafische Benutzeroberflächen (graphical user interfaces, GUIs) sind eine der wichtigsten und meistgenutzten Schnittstellen für die Interaktion zwischen Mensch und Computer [RSG17]. Die GUI sorgt für die Wahrnehmung und Nutzbarkeit der Software [RSG17]. Um die Nutzung des Softwaresystems zu ermöglichen, muss eine funktionierende GUI sichergestellt werden. Laut [Kle12] entfallen zwischen 30 und 70 Prozent der Entwicklungszeit bei Software auf die Qualitätssicherung [RSG17]. Aufgrund dieses großen zeitlichen Anteils am gesamten Entwicklungsprozesses, haben Optimierungen bei Zeit und Aufwand in diesem Bereich einen großen Einfluss auf die schnelle und korrekte Entwicklung von Software. Selbstverständlich dürfen Prozessoptimierungen und Senkung der Entwicklungszeit hierbei keine Einbußen in der Qualität verursachen.

Für GUI-Tests werden bei der AEB Testpläne erstellt und entsprechend vor den Releases manuell durchgeführt. Diese GUI-Tests sind ein zeitintensiver Prozess und benötigen entsprechend Ressourcen von Mitarbeitern und können daher nicht beliebig oft wiederholt werden. Um trotzdem regelmäßiges Feedback an die Entwickler zu geben, muss dieser Prozess automatisiert werden. Wenn Cross-Browser-Testing erforderlich ist, vervielfacht sich der manuelle Aufwand. Somit wird das Verhältnis zwischen Nutzen durch Zeitersparnis und Aufwand für die Automatisierung besonders interessant. Durch automatisierte Tests können Fehler schneller entdeckt und behoben werden, was häufigere Releases ermöglicht und Sicherheit bei den Entwicklern schaffen kann. Durch Vertrauen in eine umfassende Testsuite, kann die Bereitschaft für Refactorings erhöht werden und in einer verbesserten Wartbarkeit resultieren.

Meines Erachtens sind automatisierte Tests eines der spannenden Felder in der Softwaretechnik. Durch Continuous Delivery und eine immer kürzere Time-to-Market, werden automatisierte Tests und günstige Wiederholbarkeit immer wichtiger. Trotzdem ist es ein Thema, welches

nach meiner Ansicht viel zu oft vernachlässigt wird. Mit dem bereits angesprochenen Nutzen-Aufwand-Verhältnis, lassen sich durch automatisierte Tests Verbesserungen im Releaseprozess realisieren. Dies macht es für mich zu einem hoch interessanten Thema.

1.2 Ziele

Ziel dieser Bachelorarbeit ist eine Analyse des Nutzen-Aufwand-Verhältnisses für automatisierte GUI-Tests bei der AEB. Im ersten Schritt werden beispielhaft anhand eines ausgewählten Produkts GUI-Tests automatisiert. Für diesen Zweck kommt ein für die AEB geeignetes Werkzeug zum Einsatz. Anschließend wird das Nutzen-Aufwand-Verhältnis analysiert und die Integration der automatisierten GUI-Tests in die Build-Pipeline beurteilt. Hierzu wird ein Bewertungsschema festgelegt, um die Veränderung durch automatisierte GUI-Tests möglichst objektiv bewerten zu können. Die Auswertungsergebnisse werden so transparent wie möglich aufgelistet, um die Analyse nachvollziehbar zu gestalten. Damit kann diese Arbeit als Entscheidungsgrundlage für oder gegen automatisierter GUI-Tests dienen.

1.3 Zustand vor der Automatisierung

GUI-Tests werden bei der AEB in einem definierten Zeitraum vor jedem Produkt- oder Update-Release durchgeführt. Hierfür werden mögliche Anwendungsfälle systematisch anhand von Testbögen überprüft. Es fällt einmalig Aufwand für die Erstellung der Testbögen an. Unabhängig von Testläufen müssen diese Testbögen aktualisiert und gewartet werden. Für jeden Testlauf fällt zusätzlicher Aufwand für die Vorbereitung der Daten, sowie das Durchführen der Testbögen an. Pro Testlauf beträgt der manuelle Aufwand, für einen vollständigen Test des in dieser Arbeit gewählten Produkts, etwa sechs Stunden.

1.4 Gliederung

Diese Bachelorarbeit gliedert sich in die folgenden Kapitel:

Kapitel 2 – Qualitätssicherung und automatisierte Softwaretests stellt die theoretischen Grundlagen für die anschließenden Kapitel vor. GUI-Tests werden als Teil von Systemtests in ein Gesamtbild eingeordnet und es werden verschiedene Techniken für die Automatisierung und Testerstellung vorgestellt.

Kapitel 3 – Bewertungsgrundlagen beschreibt die Kriterien, nach denen das Nutzen-Aufwand-Verhältnis untersucht und beurteilt werden soll. Um eine möglichst objektive Bewertung zu ermöglichen, wird zwischen quantitativer und qualitativer Analyse unterschieden.

Kapitel 4 – Einführung automatisierter GUI-Tests zeigt die Schritte der Umsetzung und Integration der GUI-Tests. Hierzu gehören die Auswahl der Testfälle, genutzte Selektoren und Abstraktionslevel, Cross-Browser-Testing und die Veränderung der Build-Pipeline.

Kapitel 5 – Nutzen-Aufwand-Verhältnis präsentiert das Nutzen-Aufwand-Verhältnis. Hierbei kommen die in Kapitel 3 definierten Kriterien, für die Bewertung der automatisierten GUI-Tests, zum Einsatz. Es liefert einen umfassenden Überblick über die Vor- und Nachteile der Automatisierung.

Kapitel 6 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen. Es gibt Empfehlungen für das weitere Vorgehen bei der AEB und enthält Best-Practices. Abschließend werden weiterführende Themen vorgestellt.

2 Qualitätssicherung und automatisierte Softwaretests

Für die Einführung und die anschließende Analyse sind Grundlagen der Qualitätssicherung und der Automatisierung von Softwaretests, im Speziellen von GUI-Tests, notwendig. Hierzu werden zuerst die Grundlagen der Qualitätssicherung und die Messbarkeit in Bezug auf Softwaretests betrachtet. Anschließend werden die Softwaretests genauer beschrieben und Einblicke in die technischen Details der Umsetzung gegeben. Die Bachelorarbeit baut unter anderem auf die Fachstudie „Evaluierung von GUI-Testing Frameworks“ auf [RSG17]. Aufgrund der thematischen Überschneidungen der Grundlagen, wird diese Fachstudie an diversen Stellen als Quelle genutzt.

2.1 Qualitätssicherung

Qualitätssicherung ist ein Begriff für verschiedene systematische und geplante Maßnahmen, welche der Einhaltung von Qualitätsanforderungen dienen. Mit Hilfe einer Qualitätssicherung kann eine gleichbleibende Produktqualität erreicht werden. In DIN-ISO-Norm 9126 wird Software-Qualität definiert als „Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte Erfordernisse zu erfüllen“ [Hof12]. Zu den Qualitätskriterien zählen die Funktionalität, Laufzeit, Zuverlässigkeit, Wartbarkeit und viele weitere Eigenschaften [Hof12; LL13]. Welche Qualitätssicherungsmaßnahmen die geforderten Kriterien sicherstellen, muss je nach Produkt, Kunde und Unternehmen definiert werden. Die Eigenschaften Funktionalität und Wartbarkeit sind für die AEB besonders von Interesse. Die Qualitätssicherung sollte validiert werden, um sicherzugehen, dass die Maßnahmen die gewünschten Effekte erzielen. Im Idealfall kann dies anschließend gemessen und belegt werden.

2.1.1 Softwaretests

Zur Qualitätssicherung von Softwareprodukten gehören Softwaretests [Kle12; RSG17]. Um die Qualität der Software sicherzustellen, dienen in erster Linie eine geeignete Softwarearchitektur und entsprechende Prozesse [Dij70; RSG17]. Softwaretests bieten keine Garantie, dass eine Software fehlerfrei ist [Dij70; Kle12; RSG17]. Tests können lediglich vorhandene

Fehler aufzeigen [Dij70; RSG17]. Für durchgeführte Tests kann ein Fehlverhalten der Software ausgeschlossen und somit korrektes Verhalten angenommen werden [Dij70; Kle12; RSG17]. Ein erfolgreicher Testlauf bestätigt das getestete Verhalten, welches im Idealfall dem erwarteten Verhalten entspricht [RSG17]. Werden hier Abweichungen entdeckt, empfehlen sich Anpassungen oder Ergänzungen der Testfälle [RSG17; Wac15].

Automatisierte Softwaretests, die ohne manuellen Eingriff durchgeführt werden, sind günstig in der Durchführung und können reproduzierbar wiederholt werden [Fow12; Kle12; RSG17]. Durch die Reproduzierbarkeit können automatisierte Tests als Regressionstests ausgeführt und unerwartete Verhaltensänderungen detektiert werden [RSG17]. Wenn Entwickler durch diese Regressionstests Vertrauen und Sicherheit entwickeln, führt dies zu einer erhöhten Bereitschaft für Refactorings und besserer Softwarequalität in Form von Wartbarkeit [Hof12; Kle12; RSG17].

2.1.2 Messbarkeit

Um die Effekte und das Erreichen der Ziele der Qualitätssicherung zu überprüfen, können verschiedene Metriken eingesetzt werden. Messbar sind beispielsweise die Anzahl der Testfälle, die Laufzeit der Tests, die Code-Abdeckung, der Aufwand für die Erstellung und Wartung der Tests, die Anzahl an gefundenen Fehlern oder das Verhältnis zwischen der Fehlerzahl beim Testen und im Produktivbetrieb [FG99]. Dabei muss betrachtet werden, was für die entsprechende Situation relevante Aspekte und gute Metriken sind. Während die Code-Abdeckung eine sehr aussagekräftige Metrik für Unittests ist, ist diese für Systemtests nicht unbedingt relevant. Bei GUI-Tests ist der Nutzen durch entfallene manuelle Tests gut messbar. Dies ist einer von vielen Faktoren des Nutzens. Oft sind die andern Faktoren des Nutzens schwer zu messen und zu bewerten. Der Nutzen eines entdeckten Fehlers ist in Zeit oder Geld meist nicht direkt zu beziffern. Noch komplexer ist der Nutzen eines Tests, der keine Fehler gefunden hat. Eine umfassende Testsuite hat einen eindeutigen Nutzen in Form von besserer Wartbarkeit des Produkts, welcher jedoch schwer und nicht direkt messbar ist [Ham10]. Ebenso unklar ist die Bezifferung der Kosten von Fehlern, welche erst beim Kunden entdeckt werden und möglicherweise das Vertrauen in das Produkt oder den Hersteller senken.

Für die Umsetzung von Qualitätssicherungsmaßnahmen, lässt sich der personelle Aufwand gut messen. Aus der Arbeitszeit können die finanziellen Aufwendungen sehr genau bestimmt werden. Zusammenfassen lässt sich dies nach [Ham10], dass der Aufwand für Prüfungen früh sicht- und messbar wird. Dagegen ist der Nutzen eine schwer zu messende Qualitätsverbesserung, welche erst langfristig zu Einsparungen führt [Ham10]. Diese Arbeit untersucht das Nutzen-Aufwand-Verhältnis daher mit einer quantitativen und einer qualitativen Analyse.

2.2 Testpyramide

Softwaretests können anhand verschiedener Kriterien unterschieden werden [BF14; RSG17]. Gebräuchliche Kriterien sind der Zweck (*Objective*) und das Testobjekt (*Target*) [BF14; RSG17]. Performanz, Sicherheit oder Zuverlässigkeit sind Unterscheidungen anhand des Zwecks [BF14; RSG17]. Testobjekte sind dagegen testbare Einheiten, wie Module oder ein ganzes System [BF14; RSG17].

Die Test-Pyramide in Abbildung 2.1 zeigt diese Einteilung anhand des Testobjekts [BF14; Coh09; Fow12; RSG17; Wac15]. Mit Defintionen von [BF14] wurden die Begriffe der Pyramide verallgemeinert. [Fow12; RSG17; Wac15] ergänzen diese Pyramide um Eigenschaften der Ebenen. Zusätzlich wird das explorative Testen dargestellt [Sco12]. Es handelt sich dabei um eine intuitive, jedoch nicht geplante Ergänzung der Tests [BF14]. Dies dient dem Entdecken von Fehlern, ohne explizit als Produktfreigabe für Releases zu dienen [BF14].

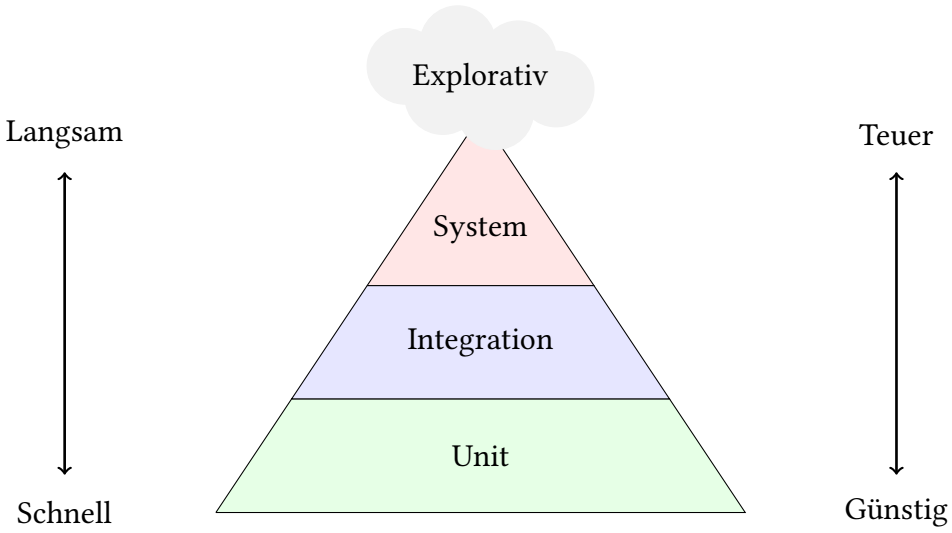

Abbildung 2.1: Die erweiterte Test-Pyramide mit ihren Ebenen und Eigenschaften [BF14; Coh09; Fow12; RSG17].

Die Pyramide besteht aus drei Ebenen, den Unittests, Integrationstests und Systemtests. Auf der untersten Ebene stehen die Unittests, diese sollten einen Großteil der Tests ausmachen [Fow12; RSG17; Sco12; Wac15]. Unittests prüfen einzelne Klassen, Methoden, Funktionen oder Unterprogramme und sind meist als White-Box-Tests implementiert [BF14; LL13; RSG17]. Durch die Überprüfung einzelner und in sich abgeschlossener Einheiten, sind diese Tests unabhängig von externen Schnittstellen und Fremdsystemen [Fow12; Wac15]. Dies sorgt für eine schnelle Ausführbarkeit und Lokalisierung von Fehlern. Im Vergleich mit den anderen Ebenen, sind sie günstiger in der Erstellung und Wartung [Fow12; RSG17]. Außerdem stellen Unittests meist keine speziellen Ansprüche an die Testumgebung.

Systemtests, auf der obersten Ebene, werden meist als Black-Box-Test implementiert [BF14; RSG17]. Sie prüfen ein ganzes System auf funktionale oder nicht-funktionale Anforderungen [BF14; RSG17]. Dabei werden zusätzlich Konfigurationen, externe Schnittstellen und Fremdsysteme mitgetestet. Systemtests stellen damit spezielle Anforderungen an die Testumgebung. Das Testsystem muss automatisiert bereitgestellt sowie die fehlerfreie Funktion von Anbindungen und Fremdsystemen sichergestellt werden. Deshalb sind Systemtests aufwändiger in der Implementierung, fragiler in der Ausführung und sollten daher mit Bedacht eingesetzt werden [Fow12; RSG17; Wac15]. Allgemein gilt, dass ein Test auf möglichst niedriger Ebene umgesetzt werden sollte [Coh09; Fow12; RSG17; Sco12; Wac15].

2.3 Nutzen von automatisierten Tests

Besonders drei Aspekte sind für den Nutzen automatisierter Tests hervorzuheben:

- Durch Automatisierung der Tests ist schnelles Feedback an den Entwickler möglich[Fow13a; Hof12].
- Eine gute Testsuite reduziert die Notwenigkeit für manuelle Tests [Fow13a].
- Je früher Fehler entdeckt werden, umso geringer ist der Aufwand für die Korrektur [Kle12; LL13].

Wenn der Fehler zeitnah entdeckt wird, muss sich der Entwickler nicht ein weiteres Mal in den betroffenen Code einarbeiten. Falls die Software bereits als Release an den Kunden ausgeliefert und installiert wurde, kann großer Schaden für ein Softwareunternehmen entstehen [Kle12; RSG17]. Die Software muss ein zweites Mal freigegeben und durch den Kunden installiert und in Betrieb genommen werden, was für diesen zusätzlichen Aufwand und Kosten bedeutet.

Für das Prüfen von funktionalen Anforderungen der Benutzeroberfläche kommen GUI-Tests zum Einsatz [Hof12; Kle12; RSG17]. Im Rahmen dieser Bachelorarbeit liegt der Schwerpunkt auf der Automatisierung von Benutzerverhalten. Es können Anwendungsfälle automatisiert verifiziert und die bestehenden Testbögen abgelöst werden [Kle12; RSG17]. Dies stellt sicher, dass die getesteten Anwendungsfälle der Software genauso durch den Kunden genutzt werden können [Kle12; RSG17].

Um die Wartbarkeit von GUI-Tests zu gewährleisten, müssen diese gut abstrahiert werden. So werden bei Veränderungen der Software nur an der jeweiligen Stelle Änderungen nötig und nicht in jedem Testfall [Fow12; Wac15]. Durch die Einhaltung des Prinzips der dargestellten Pyramide, wird der Aufwand für die Erstellung und Wartung der Testsuite minimiert [Coh09; Fow12; RSG17; Sco12; Wac15]. Tests sollten daher immer auf der jeweils niedrigst möglichen Ebene automatisiert werden [Coh09; Fow12; RSG17; Sco12; Wac15].

2.3.1 Vergleich zu manuellen Tests

Im Unterschied zu automatisierten Tests werden manuelle Tests durch Menschen durchgeführt. Für ein systematisches Vorgehen und ein effizientes Testen werden Testbögen erstellt [BF14; RSG17]. Während eines Tests werden die Testbögen abgearbeitet und dienen somit der Dokumentation und Nachvollziehbarkeit von Tests und Ergebnissen. Zusätzlich fällt parallel zur Softwareweiterentwicklung die Wartung dieser Testbögen an [RSG17]. Der größte Aufwand ist die manuelle und wiederholte Durchführung der Tests [Kle12; RSG17]. Deshalb sollten nach Möglichkeit die Softwaretests automatisiert und nur schwierig zu automatisierende Tests manuell durchgeführt werden [Kle12; RSG17; Sco12].

Wird nicht mit Testplänen gearbeitet, spricht man von explorativem Testen [BF14]. Explorative Tests stellen eine gute Ergänzung dar und ermöglichen es dem Entwickler, selbst Einblicke in die Software zu erlangen. Hierdurch können komplexere Zusammenhänge und Verbesserungen für die Software entdeckt und entwickelt werden. Exploratives Testen kann als Ergänzung dienen, jedoch nicht als effiziente Qualitätssicherung [BF14; RSG17].

Durch Testautomatisierung ist sichergestellt, dass die Tests immer exakt gleich ablaufen [FG99]. Die Eingaben und Reihenfolgen sind exakt wiederholbar [FG99]. Dies kann bei manueller Durchführung der Tests nicht garantiert werden [FG99]. Mögliche Fehler durch den Tester werden reduziert.

2.4 Techniken der Browser-Automatisierung

Automatisierte GUI-Tests für Webanwendungen setzen die Bedienung der Browser durch die Testsoftware voraus. Für die Automatisierung der Browser gibt es zwei verbreitete Technologien [RSG17]. Entweder wird der Netzwerkverkehr durch einen Proxy geleitet oder ein WebDriver greift direkt in Browser ein [17a; 17d; RSG17]. Diese beiden Technologien werden im Folgenden näher erläutert.

2.4.1 Proxy

Beim Proxyansatz wird beim Start des Browsers eine Proxyseite geladen [Mos17]. Wie in Abbildung 2.2 beschrieben, wird der Netzwerkverkehr des Browsers durch einen Proxy geleitet, welcher die Daten manipulieren und Code für Testzwecke injizieren kann [17d; Mos17; RSG17].

Bilder und ähnliche Elemente werden, je nach Implementierung, am Proxy vorbeigeleitet und direkt übertragen [Mos17]. Events, wie beispielsweise Mausklicks, werden durch den injizierten JavaScript Code ausgelöst [Dev17a]. Bei dieser Technologie wird keine zusätzliche Software für den jeweiligen Browser benötigt [Dev17a; Mos17; RSG17].

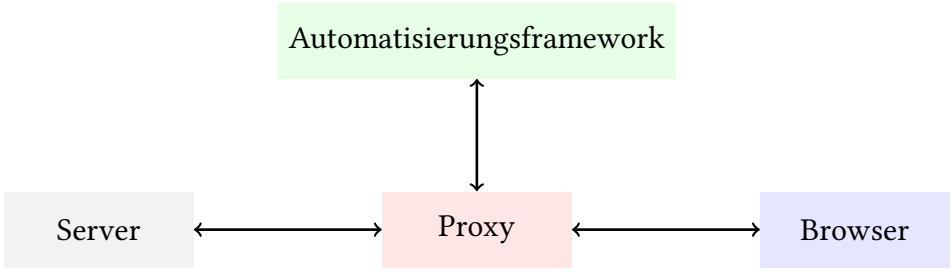

Abbildung 2.2: Ein Proxy für die Automatisierung eines Browsers.

2.4.2 WebDriver

Der Begriff WebDriver hat drei Bedeutungen:

- Das vom World Wide Web Consortium (W3C) standardisierte Protokoll [BS+17].
- Eine Sammlung an Programmierschnittstellen (application programming interfaces, APIs) für verschiedene Programmiersprachen [17a].
- Eine Implementierung zur Automatisierung eines Browsers [17b; RSG17]

Die Bezeichnung WebDriver umfasst den gesamten Bereich zwischen den APIs und der Automatisierung des Browsers, wie in Abbildung 2.3 dargestellt. Die Implementierungen des WebDrivers sind browserspezifisch und werden in vielen Fällen von den Browserherstellern entwickelt [17b; Goo17; Mic17b; Moz17; RSG17]. Durch die direkte Automatisierung des Browsers werden nach Möglichkeit ausschließlich native Events verwendet und auf die Ausführung von JavaScript-Code im Browser lediglich als Fallback-Methode zurückgegriffen [RSG17; Ste17]. Seiteneffekte werden hiermit vermieden und es können Sicherheitsbeschränkungen, die für JavaScript gelten, umgangen werden [RSG17; Ste17].

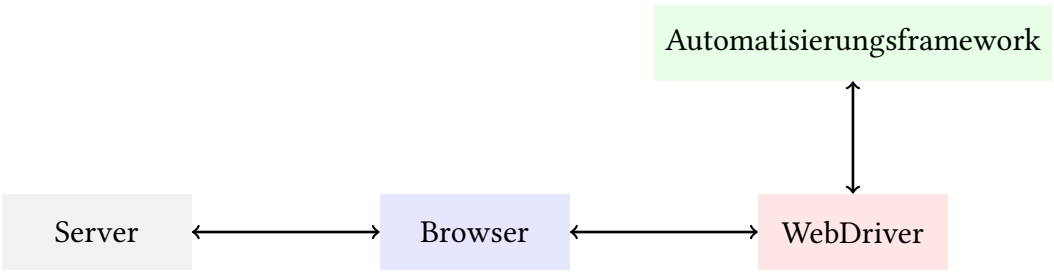

Abbildung 2.3: Ein WebDriver für die Automatisierung eines Browsers.

2.5 Erstellungsvarianten von automatisierten GUI-Tests

Für die Erstellung von automatisierten GUI-Tests kommen verschiedene Vorgehensweisen in Frage. Zu den wichtigsten Arten gehören „Record and Play“ und die Programmierung [RSG17]. Diese Erstellungsarten werden im Folgenden erläutert.

Beim „Record and Play“ wird ein Browser-Plugin oder ein Proxyserver verwendet, um die Interaktionen mit der Webseite aufzuzeichnen [17c; Dev17b; RSG17]. Die Selektoren werden für einzelne Elemente automatisch gewählt, was mit jedem Testfall wiederholt wird und somit zu einer Kopie führt [17c; RSG17]. Aufgrund fehlender Abstraktionsmöglichkeiten hat eine Veränderung der Software meist eine komplette Neuaufzeichnung des betroffenen Testfalls zur Folge [RSG17]. [Ham16] konnte etwa 74 Prozent der gebrochenen Testfälle auf veränderte Selektoren zurückführen.

Eine andere Art der Erstellung ist die Programmierung von Testfällen, ähnlich wie Unit- oder Integrationstests. Hierbei sind alle Möglichkeiten der gewählten Programmiersprache verfügbar [RSG17]. Es kann mit Datenbanken, Dateien oder anderen Systemen gearbeitet werden. Abstraktionsmöglichkeiten, wie das Page-Object-Pattern, sind möglich und sollten verwendet werden um den Wartungsaufwand zu reduzieren [Fow13b; RSG17]. Die Selektoren müssen durch den Programmierer manuell gewählt werden, dies ermöglicht den Selektor robuster festzulegen [RSG17]. Ändert sich ein Selektor im Rahmen der Softwarewartung und die Abstraktion wurde korrekt gewählt, genügt eine Anpassung für alle Testfälle.

2.6 Cross-Browser-Testing

Verschiedene Browser und Browserversionen stellen Webelemente unterschiedlich dar. Dies liegt zum einen an der ständigen Weiterentwicklung von Standardisierungen und Webtechnologien. Zum Anderen werden auch die Browser selbst kontinuierlich weiterentwickelt, unterstützen neue Technologien und entfernen Darstellungsfehler. Die Standardisierungen des W3C haben stark an Bedeutung gewonnen, wodurch die Inkompatibilitäten zwischen den aktuellen Browserversionen abgenommen haben. Dennoch werden nicht in allen Browsern die aktuellen W3C Spezifikationen umgesetzt. Je nach Browser fehlen Funktionen ganz oder werden nur teilweise unterstützt [Dev17c].

Zusätzlich gehen die Browser unterschiedlich tolerant mit Fehlern in HTML und CSS um. Dies kann dazu führen, dass in manchen Browsern ein Element aufgrund eines Fehlers im Document Object Model (DOM) nicht dargestellt wird, während andere Browser diesen Fehler ignorieren und das Element wie gewünscht darstellen.

Um diese Art von Problemen auszuschließen, sollte Software, auf allen Browsern für die sie freigegeben ist, getestet werden. Aufgrund der vielen möglichen Kombinationen aus Browsern und Betriebssystemen, respektive deren Versionen, ist dies kaum leistbar. In der Praxis schränkt man die Tests deshalb meist auf ein Betriebssystem und eine bis wenige Versionen pro Browser

ein. Für Businessanwendungen bedeutet dies dennoch oft mindestens die Browser Chrome, Firefox und Internet Explorer. Werden die Tests manuell durchgeführt, wird der Aufwand für die Tests damit mindestens verdreifacht.

3 Bewertungsgrundlagen

Dieses Kapitel legt die Grundlagen für die Bewertung des Zustands vor und nach der Automatisierung fest. Es zeigt, aufgrund welcher Kriterien diese Zustände verglichen und bewertet werden. Angesichts dieser Kriterien wird das Nutzen-Aufwand-Verhältnis für die AEB analysiert.

3.1 Bewertungsschema

Das Bewertungsschema ist zweigeteilt. In der quantitativen Analyse werden die messbaren Werte ausgewertet. In der qualitativen Analyse werden die Einschätzungen und Expertenmeinungen gesammelt und bewertet. Schwierigkeiten bei einer Analyse des Nutzen-Aufwand-Verhältnisses in der Qualitätssicherung macht der Unterschied in der Charakteristik von Nutzen und Aufwand. Nach [Ham10] ist der Aufwand für Tests sicht- und messbar, der Nutzen dagegen eine langfristige, nicht direkt messbare Qualitätsverbesserung. Die Nutzenanalyse für automatisierte GUI-Tests ist im Rahmen einer Bachelorarbeit beschränkt, da sich der Nutzen meist nicht genau beziffern lässt und die Auswirkungen erst über einen längeren Zeitraum zum Tragen kommen.

3.2 Quantitative Analyse

Ein Teil des Aufwands und Nutzens lässt sich genau bestimmen und messen. Um eine möglichst objektive Bewertung der Testautomatisierung zu erstellen, werden die messbaren Aufwendungen, wie Arbeitszeit oder Ressourcen, in der quantitativen Analyse betrachtet.

3.2.1 Aufwand der Build-Pipeline

Die Build-Pipelines mit GUI-Tests laufen während der Evaluierung parallel zu den bestehenden Build-Pipelines ohne automatisierte GUI-Tests. Die Laufzeit kann mit und ohne GUI-Tests in Jenkins ausgelesen und verglichen werden. Der Aufwand kann als Mittelwert über die Dauer der Testläufe bestimmt werden. Die sonstigen Veränderungen, im Produkt beziehungsweise in den Unit- oder Integrationstests, fließen somit nicht in die Bewertung ein.

3.2.2 Nutzen und Aufwand gegenüber manuellen Tests

Der größte Nutzen entsteht durch die automatische Ausführung der Tests. Die GUI-Tests müssen nicht manuell durchgeführt werden. Hier wird der Nutzen durch gesparte Arbeitszeit für die Erstellung und Wartung der Testpläne sowie manuelles Testen summiert. Dem gegenübergestellt wird der personelle Aufwand für die Automatisierung und Wartung von GUI-Tests. Damit lässt sich die Amortisationszeit prognostizieren.

3.3 Qualitative Analyse

Die qualitative Analyse stützt sich auf die Einschätzungen von Mitarbeitern der AEB. Im Vergleich zu der quantitativen Analyse fällt dieser Teil subjektiver aus. Deshalb wird in diesem Teil der Analyse nur der schwer und nicht direkt messbare Aufwand und Nutzen betrachtet.

3.3.1 Anforderungen der AEB

Anforderungen der AEB an das GUI-Testing Framework, wurden im Rahmen der Fachstudie „Evaluierung von GUI-Testing Frameworks“ erhoben [RSG17]. Mit der Entscheidung der AEB für Selenide sind bereits einige Anforderungen festgelegt. Beispielsweise die Technologien, wie Programmiersprache, Maven und JUnit-Integration oder testbare GUI-Technologien. Anforderungen, welche noch nicht festgelegt sind, beziehungsweise besonders relevant in Bezug auf die Umsetzung und das Nutzen-Aufwand-Verhältnis sind, werden hier betrachtet. Die Anforderungen dienen als Aspekte um den Nutzen und Aufwand zu identifizieren. Sind diese Anforderungen erfüllt, ergibt sich ein entsprechender Nutzen. Können sie nicht erfüllt werden, kann weiterer Aufwand entstehen.

In der folgenden Tabelle 3.1 werden die Anforderungen aufgelistet, die im Rahmen der Fachstudie erfasst wurden und im Rahmen dieser Arbeit untersucht werden [RSG17].

Übergreifende Ziele	
Essentiell	Kosten und Aufwand
	Wartungskosten
	Qualitätssicherung bei Release
	Testen von Abläufen

Wichtig	GUI-Testprozess standardisieren Testen von GUI Elementen Performance Test
Unwichtig	Testen von Darstellung
Erstellung von Tests	
Essentiell	Testbausteine Abstraktionsmöglichkeiten
Wichtig	Von Fachexperten Von Testexperten Recording von Tests “Schritt-für-Schritt“-Debugmodus
Ablauf der Tests	
Essentiell	Ausführen in Build-System Lokal beim Entwickler „Echter“ lokaler Browser
Wichtig	Headless Browser Testablauf sichtbar im Browser
Reporting	
Essentiell	In Jenkins Log-Dateien
Wichtig	Screenshots

Tabelle 3.1: Die Anforderungen der AEB an ein GUI-Testing-Framework.

3.3.2 Aufwand durch Softwarewartung

Um einzuschätzen, wie sich GUI-Tests auf die Wartung und Weiterentwicklung der Software auswirken, werden die typischen Veränderungen betrachtet. Änderungen, die Anpassungen der GUI-Tests erforderlich machen, führen zu Mehraufwand. Die betrachteten Änderungen umfassen die Internationalisierung, Ablaufänderungen via Code und Veränderungen am GUI-Framework. Außerdem Änderungen der Konfiguration von Produkt, Wizard oder Geschäftsprozessmodell und -notation (Business Process Model and Notation, BPMN). Anhand dieser Punkte wird betrachtet, welche Anpassungen an den GUI-Tests nötig sind. Unerwünscht ist beispielsweise, wenn Änderungen an den Übersetzungen der Software, Anpassungen an den Tests erfordern.

3.3.3 Nutzen bei Entwicklung und Releases

Weiteren Nutzen bei der Entwicklung und für Releases der Software kann durch verschiedene, teilweise langfristige Effekte erzielt werden. Diese werden anhand der folgenden Fragen untersucht:

- Werden Fehler entdeckt?
- Wird sichergestellt, dass funktionierende Software ausgeliefert wird?
- Entsteht ein Vorteil durch die erhöhte Feedbackfrequenz?
- Ist das Feedback hilfreich für die Entwickler?
- Steigt das Vertrauen und Sicherheitsgefühl der Entwickler in die Testsuite?

4 Einführung automatisierter GUI-Tests

Dieses Kapitel beschreibt die Auswahl und Priorisierung der Umsetzung der GUI-Tests sowie deren Einführung anhand des AEB Produkts BrokerIntegration. Die Umsetzung der Automatisierung wird aufgezeigt und die Browserkompatibilität für Cross-Browser-Testing untersucht. Abgeschlossen wird dieses Kapitel mit der Integration der Tests in die Build-Pipeline und Screenshotregressionstests.

4.1 BrokerIntegration

Im Rahmen dieser Arbeit werden automatisierte GUI-Tests exemplarisch anhand eines Produkts umgesetzt. Das zu testende Produkt BrokerIntegration dient der Zusammenarbeit mit Zollagenten [AEB17b]. Natürliche und juristische Personen können eine Vertretung (Zollagent) ernennen, welche Zollanmeldungen für diese abwickelt [Gen17]. Die Zusammenarbeit wird durch BrokerIntegration unterstützt, indem für die Zollanmeldung notwendige Daten mit anderen Systemen synchronisiert werden. Den Zollagenten werden diese Daten zur Verfügung gestellt und ermöglichen diesen die weitere Bearbeitung. Damit kann ein Zollagent die komplette Zollabwicklung für den Auftraggeber vornehmen. Als Auftraggeber ist eine übersichtliche Steuerung und Überwachung der Zollanmeldungen möglich. Zudem sind Benachrichtigung zu Statusänderungen für Zollagenten und Auftraggeber integriert. Die GUI des Produkts ist als eine Weboberfläche mit Vaadin realisiert.

4.2 Auswahl und Priorisierung der Tests

Begonnen wurde mit der Entscheidung welche Art von GUI-Tests umgesetzt werden sollen und der Priorisierung für die Automatisierung dieser Tests. In Gesprächen mit mehreren Mitarbeitern der AEB wurde die Priorität zum einen auf kritische, zum anderen auf die meistgenutzte Anwendungsfälle gelegt. Als eher unwichtig wurde von der AEB das Testen einzelner GUI-Elemente auf Funktion und Darstellung festgelegt.

Ein kritischer Anwendungsfall ist für die Nutzbarkeit der Software elementar. Die Nutzung ist bei einem kritischen Fehler nicht möglich, beispielsweise das Bezahlen in einem Onlineshop. Ein meistgenutzter Anwendungsfall wird überdurchschnittlich oft genutzt. Im Gegensatz zu

einem kritischen Anwendungsfall, kann die Software trotz eines Fehlers genutzt werden. Führt aber meist zu Komforteinbußen, beispielsweise die fehlende Darstellung eines Artikelbildes.

Die Testbögen für manuelle Tests umfassen derzeit 42 Seiten und die Durchführung benötigt etwa sechs Stunden. Die in den Testbögen beinhalteten Testfälle wurden anhand der festgelegten Priorisierung eingeteilt und in mehreren Iterationen automatisiert.

4.3 Automatisierung der Testfälle

Aufbauend auf das Ergebnis der vorangegangen Fachstudie zur Werkzeugauswahl und internen Evaluierungen der AEB, wurde Selenide als Framework festgelegt [RSG17]. Selenide baut auf Selenium auf und nutzt den WebDriver-Ansatz zur Automatisierung des Browsers. Prinzipiell kann jeder Browser, für welchen ein WebDriver vorhanden ist, für die Tests genutzt werden. Details hierzu befinden sich in Abschnitt 4.3.3.

Die Testfälle wurden in drei Iterationen ausgebaut. Begonnen wurde mit fünf Testfällen, die zunächst auf neun und schließlich auf 16 Testfälle erweitert wurden. In diesen drei Iterationen konnten etwa 70 Prozent der manuellen Testfälle automatisiert werden. Die anderen 30 Prozent der Testfälle beziehen sich auf die Interaktion mit E-Mails, weshalb diese in der ersten Phase zurückgestellt wurden. Prinzipiell könnte, mit Hilfe der JavaMail API, ein einfacher E-Mail-Client entwickelt werden, der den Inhalt für die GUI-Tests bereitstellt und die Automatisierung ermöglicht [Ora17]. Ob diese Erweiterung für die Praxis sinnvoll ist, wurde im Rahmen dieser Arbeit nicht untersucht.

Wichtige Bestandteile für die Einführung von automatisierten GUI-Tests werden in den folgenden Abschnitten behandelt. Dazu gehören die Selektoren für die Auswahl von Elementen und die Abstraktion der Tests für eine bessere Wartbarkeit und Wiederverwendbarkeit. Zusätzlich wird die Verwendung verschiedener Browser für die Tests umgesetzt.

4.3.1 Selektoren

Selektoren sind Definitionen anhand denen Elemente in einer HTML-Seite bestimmt werden können. Zur Definition eines Selektors kann unter anderem Cascading Style Sheets (CSS) und XPath genutzt werden [Cod17]. XPath bezeichnet eine Sprache um Knoten in einem Extensible Markup Language (XML) Dokument zu adressieren [RDS17]. Die Selektoren sollten möglichst so gewählt werden, dass die GUI-Tests robust gegen Änderungen sind.

Als Beispiel für robuste Selektoren dient eine Webanwendung mit drei Buttons. Angenommen in einem Test soll der dritte Button geklickt werden. Ein schlechter Selektor hierfür wäre beispielsweise die Position eines Buttons zu wählen. Besser wäre ein Selektor, der den Text des Buttons nutzt. Wird an die erste Position ein neuer Button hinzugefügt, wählt der schlechte Selektor einen falschen Button, der robustere Selektor funktioniert weiterhin.

Das eingesetzte Frontendframework Vaadin kann für die Elemente in der Anwendung das HTML-Attribut `id` setzen, sodass diese Elemente eindeutig angesprochen werden können. Die Produkte setzen nicht direkt Vaadin ein, sondern nutzen ein eigenes Framework, welches die Benutzeroberfläche abstrahiert. Im aktuellen Stand des GUI-Frameworks der AEB ist die mehrfache Darstellung eines Elements möglich und daher nicht vorgesehen einen eindeutigen Wert für die `id` zu setzen. Aufgrund dessen ist eine andere Methode für die Selektion der Elemente erforderlich.

Als Selektoren kommen vor allem CSS-Klassen oder CSS-Selektoren in Frage. Da dies nicht für alle Elemente zu einer eindeutigen Identifikation ausreicht, können zusätzlich Textelemente, wie Buttontexte oder Labels, in den Selektoren genutzt werden. Kommen keine lokalisierten Texte in den Selektoren vor, kann der Selektor je Page-Object als Konstante definiert werden. Die Software der AEB nutzt ein zentrales Modul für die Bereitstellung der Lokalisierung. Beinhaltet der Selektor lokalisierten Text, kann dieses Modul verwendet werden um die entsprechenden Texte der aktuellen Übersetzung zu nutzen. Dazu können die zugehörigen Lokalisierungskeys der Texte verwendet werden. So entsteht durch Änderungen an den Übersetzungen kein Wartungsaufwand an den GUI-Tests.

Die Software kann oft durch entsprechende Konfiguration an die speziellen Bedürfnisse der Kunden angepasst werden. Beispielsweise können Bezeichnungen in der Datenbank konfiguriert und damit verändert werden. Für die Einführung der Tests wurden diese Texte in einem Java Interface gebündelt. Alternativ könnten diese Texte entsprechend zur Laufzeit der Tests aus der Datenbank gelesen und für die Selektoren verwendet werden.

Die BrokerIntegration kann an die, je nach Unternehmen, unterschiedlichen Import- / Export-Verfahren angepasst werden. So können verschiedene Masken für die Verfahren genutzt und die Reihenfolgen und Bedingungen individuell festgelegt werden. In diesem Fall müssen die automatisierten Testfälle analog zu manuellen Testfällen angepasst werden.

4.3.2 Abstraktion

Wichtig für die Wartbarkeit und Wiederverwendbarkeit der GUI-Tests ist die Abstraktion. Diese wurde in drei Leveln umgesetzt, welche in Tabelle 4.1 dargestellt werden.

Das niedrigste Level besteht aus Page-Objects, welche APIs für die GUI darstellen. Bei der AEB umfasst ein Page-Object immer ein Application Controller (AC) und Application Controller View (ACV). Diese sollen in den Modulen bei dem entsprechenden AC und ACV gepflegt werden. Page-Objects, die beim GUI-Framework angesiedelt sind, können in verschiedenen Produkten wiederverwendet und der Wartungsaufwand reduziert werden. Beispielsweise kann so jedes Produktteam das Page-Object für den Login wiederverwenden. Wird nun der Login verändert, ist nur eine Anpassung eines Page-Objects erforderlich, ohne dass jedes Produktteam tätig werden muss.

Abstraktionslevel	Beschreibung	Beispiel
Page-Object	Kapselt eine Seite oder ein Element einer Seite und stellt eine API dafür her.	Loginformular wird ausgefüllt und abgeschickt durch Aufruf von `login("Mandant", "Benutzer", "Passwort")` oder Zugriff auf einzelne Felder des Logins, beispielsweise `setUser("Benutzer")`.
Aktion	Kapselt eine Aktion, die (potentiell) Teil mehrerer Testfälle ist und mehrere Page-Objects umfasst.	Ein neuer Exportauftrag wird angelegt.
Testfall	Eine Abfolge von Aktionen, die einen Anwendungsfall ergeben.	Einen neuen Exportauftrag anlegen und diesen anschließend bearbeiten und ausfüllen.

Tabelle 4.1: Eine Übersicht der genutzten Abstraktionslevel.

Im mittleren Level sind Aktionen gekapselt, die keinen eigenständigen Anwendungsfall darstellen und als Bausteine in mehreren Testfällen wiederverwendet werden können. Dies ermöglicht die einfachere Variation der Testfälle.

Auf höchstem Level werden die Testfälle definiert, welche die APIs der Aktionen und Page-Objects nutzen. Hierdurch werden verschiedene Anwendungsfälle der Software abgedeckt.

4.3.3 Cross-Browser-Testing

Theoretisch funktionieren die automatisierten GUI-Tests mit jedem beliebigen Browser, für welchen es einen WebDriver gibt. Dies deckt alle weit verbreiteten Browser, wie Chrome, Edge, Firefox, Internet Explorer, Safari sowie viele weitere Browser, ab. Die AEB nennt als Systemvoraussetzungen für die Software einen HTML5-fähigen Browser und definiert explizit aktuelle Versionen von Chrome, Firefox und Internet Explorer [AEB17c]. Der Edge Browser wird aktuell nicht offiziell unterstützt, allerdings wird er voraussichtlich im Unternehmensumfeld den Internet Explorer mittelfristig ablösen und dann offiziell unterstützt werden. Aus den zuvor genannten Gründen beschränkt sich diese Arbeit auf die vier Browser Chrome, Edge, Firefox und Internet Explorer. Zwischen den verschiedenen Browsern zeigten sich bei der Umsetzung, teilweise große Abweichungen in Geschwindigkeit und Stabilität der Testausführung. Dies resultiert aus der Unterstützung und Integration des WebDrivers sowie der Browser Engine, welche für das Rendering und Ausführen der Scripte zuständig ist.

Die Browserautomatisierung ist durch das WebDriver Protokoll des W3C spezifiziert. Dennoch treten in den WebDriver-Implementierungen Fehlfunktionen auf und teilweise fehlen auch Funktionen des Protokolls. Diese verursachen in der Praxis Probleme, beispielsweise beim Dateiupload im Edge oder beim Doppelklick im Firefox. Hier kann mit Unterscheidungen der Browser in den Page-Objects gearbeitet werden, was jedoch in den Page-Objects einen erhöhten Wartungsaufwand erzeugt. Alternativ wäre es möglich ein eigenes Testingframework zu entwickeln, welches Selenide abstrahiert, was jedoch mit viel Aufwand einhergeht. Eine weitere Möglichkeit stellt die Aspektorientierte Programmierung (AOP) dar, um an den entsprechenden Positionen im Programmcode diesen zusätzlichen Code zu injizieren. Eine Unterscheidung war lediglich an wenigen Stellen nötig, weshalb im Rahmen dieser Arbeit mit Browserunterscheidung gearbeitet wurde. Dies führte zu einem geringeren Aufwand für die Testautomatisierung.

Der WebDriver des Edge Browsers unterstützt keine Dateiuploads [16]. Hierfür existiert seit etwa zwei Jahren ein Issue, welcher vom Microsoft Edge Team bestätigt wurde [16]. Dies führt dazu, dass mehr als die Hälfte der Testfälle mit dem Edge-Browser nicht durchgeführt werden kann. Um diese Tests dennoch mit dem Edge-Browser auszuführen, wurde ein Workaround implementiert. Hierbei wird das Element für den Dateiupload angeklickt und, mit einem kleinen zeitlichen Offset, werden asynchron Tastatureingaben simuliert. In der Regel treffen die simulierten Tastatureingaben das Dialogfenster zum Öffnen einer Datei. Das Auswählen einer Datei durch einen Benutzer wird so simuliert. Wenn parallel keine Benutzerinteraktionen mit dem System stattfinden, funktioniert dieser Workaround sehr zuverlässig.

Im Firefox ist es aktuell nicht möglich den Doppelklick des WebDrivers zu nutzen. Als Workaround kommt hier JavaScript zum Einsatz, welches ein neues Doppelklick-Event erstellt und anschließend auf dem entsprechenden Element auslöst. Dieses Problem kann zuverlässig umgangen werden.

In der Praxis funktionierten die automatisierten GUI-Tests nicht direkt mit jedem Browser. Die browserbedingten Besonderheiten lassen sich umgehen, aber bedeuten grundsätzlich einen leicht erhöhten Aufwand für das Cross-Browser-Testing.

Laufzeiten verschiedener Browser

Sobald die Testfälle automatisiert sind, ist der nächste Schritt, die Tests mit verschiedenen Browsern auszuführen, um das Verhalten der getesteten Software in allen unterstützten Browsern zu testen. Bei manuellen Tests würde dies einen sehr großen Aufwand bedeuten, der in der Praxis viel Zeit benötigen und hohe Kosten verursachen würde.

Die Tabelle 4.2 zeigt Details zu den verwendeten und gemessenen Browser- und WebDriver-Versionen und die Möglichkeit die Tests im Hintergrund ohne GUI (headless) auszuführen. Die

[1]Der Mittelwert aus jeweils zehn Testläufen, gerundet auf Sekunden.

Browser	Version		Headless	Laufzeit in Sekunden[1]	
	Browser	WebDriver	möglich?	mit GUI	headless
Chrome	61	2.32	✓	215	194
Edge	38	3.14393	✗	398	-
Firefox	57	0.19	✓	237	228
Internet Explorer	11	3.6	✗	1026	-

Tabelle 4.2: Die Browser und WebDriver Versionen und Testlaufzeiten.

alternative Darstellung Abbildung 4.1 visualisiert diese Laufzeiten ersichtlicher und vergleichbarer. Für die Laufzeitermittlung wurde der Mittelwert aus jeweils zehn Testläufen auf einem Entwicklersystem berechnet. Bei anderen Systemen können diese Laufzeiten abweichen, die Größenordnungen und Verhältnisse zwischen den Browsern aus Abbildung 4.1 bleiben jedoch bestehen.

Besonders der Internet-Explorer hat hier eine verhältnismäßig lange Laufzeit. Zum Einen ist die Rendering Engine des Internet Explorer sehr langsam [Res17]. Zum Anderen existiert kein WebDriver von Microsoft, sondern eine Implementierung von Thoughtworks [17b]. Deshalb kann keine perfekte Abstimmung zwischen Browser und WebDriver erzielt werden. Im Gegensatz zu unterschiedlichen Browsern zeigt Abbildung 4.1 nur geringfügige Unterschiede zwischen Chrome und Firefox mit und ohne GUI. Mit dem Chrome laufen die Tests im Vergleich

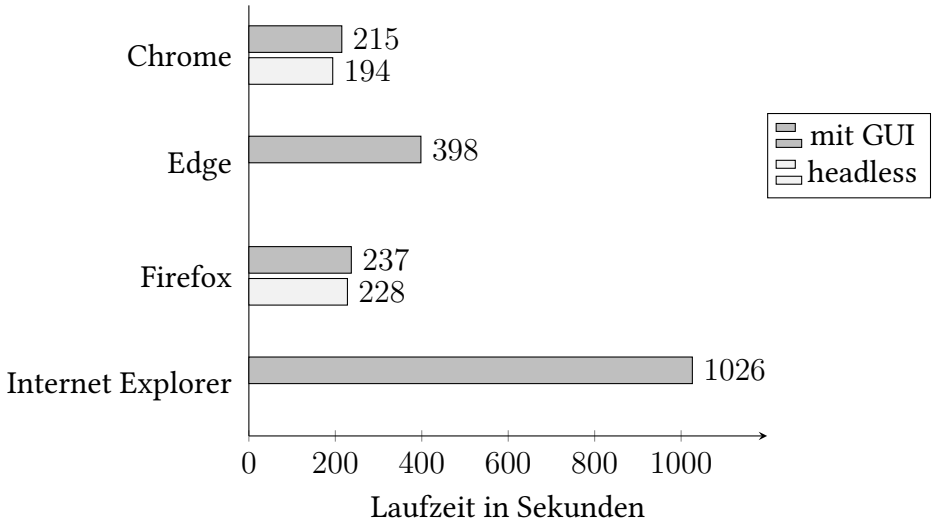

Abbildung 4.1: Die Testlaufzeiten in verschiedenen Browsern.

zum Internet Explorer um Faktor fünf schneller. Als primärer Testbrowser empfiehlt sich Chrome, da er am schnellsten ist und in der Praxis die stabilsten Testergebnisse lieferte.

4.3.4 Integration in die Build-Pipeline

Um die GUI-Tests automatisch und ohne manuellen Eingriff auszuführen, kommt eine Build-Pipeline in Jenkins zum Einsatz. Einmal täglich wird die aktuelle Version des Produktes erstellt. Diese Nightly Version des Produkts wird nach erfolgreichen Build, Unit- und Integrationstests durch Jenkins auf einem Testsystem installiert. Für automatisierte Systemtests ist diese vollautomatisierte Bereitstellung des Testsystems mit allen Anbindungen und Daten erforderlich.

Im Rahmen dieser Arbeit wurde die Ausführung der GUI-Tests nach dem Deployment ergänzt. In Abbildung 4.2 ist die Build-Pipeline dargestellt, welche durch die GUI-Tests erweitert wurde. Dabei werden die GUI-Tests mit dem Chrome, Firefox und Internet Explorer ausgeführt. Dies erfordert auf allen Jenkins-Knoten die Installation der Browser und WebDriver. Edge wird auf den Jenkins-Knoten nicht genutzt, da dieser nur mit Windows 10 verfügbar ist und als Betriebssystem Windows Server Versionen zum Einsatz kommen [Mic16; Mic17a].

Abbildung 4.2: Die aktuelle Build-Pipeline erweitert um die GUI-Tests.

Es fallen je Browser Reports in Form von Logfiles, HTML Dumps und Screenshots an. Bei Bedarf ist ein Zugriff über den Workspace von Jenkins möglich. Die Ergebnisse werden im Fehlerfall via E-Mail an die verantwortlichen Personen versandt.

4.3.5 Screenshotregressionstests

Bei der Softwarewartung und -weiterentwicklung kann es zu unvorhergesehenen Veränderungen der Software kommen. In der AEB wurden mit dem Upgrade von Vaadin 7 auf Vaadin 8 in allen Produkten Regressionstests fällig. Hierdurch hat sich die Einschätzung verändert und es hat sich der Wunsch entwickelt, auch Regressionstests für die Darstellung zu ermöglichen. Um in Zukunft diesen Aufwand zu minimieren, kann es hilfreich sein während eines Tests Screenshots zu erstellen. Fehlende oder verschobene Elemente können durch Veränderungen der Screenshots entdeckt werden.

In einer weiteren Iteration wurde die Möglichkeit implementiert, Screenshots mit Referenzbildern zu vergleichen. Hierfür werden die Farbwerte der einzelnen Pixel verglichen. Überschreitet die Differenz der Farbwerte eine bestimmte Schwelle, wird dieses Pixel als unterschiedlich markiert. Überschreitet die Summe der unterschiedlichen Pixel eine weitere Toleranzschwelle, bricht der Testfall ab und es wird ein Differenzbild erstellt. Abbildung 4.3 zeigt einen Vergleich,

bei dem die Toleranzen überschritten wurden. Die Darstellungen der Anwendung unterscheiden sich je nach Browser minimal. Meist verschiebt sich die Darstellung nur um einzelne Pixel. Das ist mit dem Auge nicht zu erkennen, erfordert jedoch für jeden Browser eigene Referenzbilder.

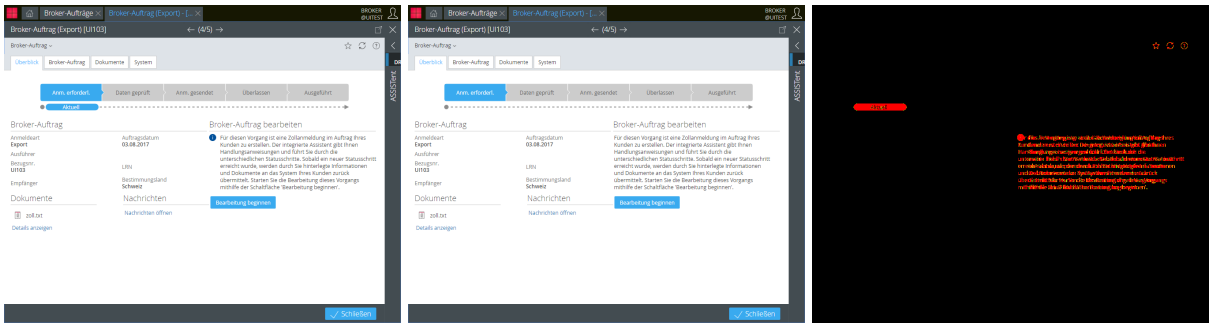

Abbildung 4.3: Der Vergleich von Referenz- und aktuellem Screenshot. Das aus dem Vergleich entstehende Differenzbild ist rechts dargestellt.

Die Referenzbilder werden mit dem Quelltext der Tests versioniert, somit sind die Tests nicht von andern Systemen abhängig. Eine Alternative hierzu ist, den Vergleich in einen Webservice wie Spectre auszulagern. Spectre verwaltet die Referenzbilder und vergleicht diese mit aktuellen Screenshots [WeA17]. Bei einer Differenz hat der Entwickler die Möglichkeit, den aktuellen Screenshot als neue Referenz für künftige Tests zu übernehmen [WeA17].

5 Nutzen-Aufwand-Verhältnis

Eine Analyse des Nutzen-Aufwand-Verhältnisses bildet die Grundlage für die Bestimmung der Wirtschaftlichkeit einer Maßnahme. Anhand dieser kann eine transparente Entscheidung für oder gegen die Automatisierung von GUI-Tests getroffen werden. Das folgende Kapitel betrachtet und bewertet den Aufwand und den gewonnenen Nutzen aus der Einführung der GUI-Tests bei der AEB. Die Analyse des Nutzen-Aufwand-Verhältnisses ist in zwei Teile gegliedert. Die quantitative Analyse betrachtet die messbaren Eigenschaften und lässt eine Ressourcen- und Personalaufwandsermittlung zu und ermöglicht einen Vergleich mit dem messbaren Nutzen. Darauf folgt eine qualitative Analyse, bei der schwierig zu messende Eigenschaften betrachtet werden. Dieses Kapitel dient der Grundlage für die abschließende Bewertung und Empfehlung.

5.1 Quantitative Analyse

Die quantitative Analyse unterteilt sich in den Ressourcenaufwand der Build-Pipeline und den Nutzen und Aufwand für die Automatisierung der GUI-Tests. Zuerst wird der zusätzliche messbare Ressourcenaufwand der Build-Pipeline betrachtet und für eine komplette Einführung bei der AEB prognostiziert. Anschließend wird der messbare personelle Nutzen und Aufwand in Bezug auf die Automatisierung der GUI-Tests betrachtet. Hierdurch kann eine Amortisationszeit prognostiziert werden.

5.1.1 Aufwand der Build-Pipeline

Dieser Abschnitt beschreibt den Ressourcenaufwand, welcher technisch für die GUI-Tests erforderlich ist. Dies stützt sich auf den gemessenen Aufwand der zur Evaluierung eingerichteten Build-Pipeline. Über die Jenkins API können die Laufzeiten ausgelesen und daraus kann ein Mittelwert gebildet werden. Das Initialisieren der Build-Pipeline dauert etwa zwei Minuten, was bei allen Produkten und Browsern ähnlich ist. Der reine Testlauf der GUI-Tests dauert mit dem Browser Chrome durchschnittlich vier Minuten und neun Sekunden. Daraus ergibt sich eine gesamte Verlängerung der Build-Pipeline um sechs Minuten und neun Sekunden. Dies führt zu der Prognose aus Tabelle 5.1. Die Hochrechnung auf alle Produkte der AEB basiert auf

dem Faktor 200. Hierbei handelt es sich um eine Einschätzung, die mit Unterstützung der verantwortlichen Mitarbeiter der AEB getroffen wurde. Sie resultiert aus der Anzahl der Produkte sowie dem voraussichtlichen Testumfang im Verhältnis zum Produkt BrokerIntegration.

Browser	Laufzeit BrokerIntegration[1]	Prognose für alle XNSG-Produkte[2]
Chrome	6:14 min	20:47 h
Edge	10:41 min	35:37 h
Firefox	6:59 min	23:15 h
Internet Explorer	24:23 min	81:18 h
Cross-Browser-Testing (Summe der Browser)	48:17 min	160:56 h

Tabelle 5.1: Die Testlaufzeiten der Build-Pipeline mit der Prognose für alle Produkte.

Es errechnen sich etwa 21 Stunden für einen Testlauf aller Produkte mit dem Chrome Browser. Sollen Cross-Browser-Tests ausgeführt werden, ergibt sich eine Summe von 161 Stunden. Wenn die Tests mit jedem Nightly Build laufen sollen, würde dies etwa sieben Jenkins-Knoten dauerhaft auslasten. Um den Aufwand gering zu halten und trotzdem Cross-Browser-Testing zu nutzen, könnte ein zweistufiges Modell zum Einsatz kommen. Beispielsweise kann täglich mit dem Chrome Browser und wöchentlich mit allen Browsern getestet werden. Dies spart Ressourcen und würde lediglich zwei Jenkins-Knoten auslasten.

5.1.2 Nutzen und Aufwand der Automatisierung

Um die Amortisationszeit zu bestimmen, wird der Zeitbedarf für die Erstellung und Wartung der Testpläne und das manuelle Testen sowie die Erstellung und Wartung der GUI-Tests verglichen. Für die Software BrokerIntegration wurden 70 Prozent der GUI-Tests automatisiert. Im Folgenden wird deshalb der manuelle Aufwand für diese 70 Prozent der Tests betrachtet. Hierbei werden nur die Zeiten für die Erstellung, Wartung und Durchführung der Tests bestimmt und hierfür die Amortisationszeit berechnet. Weiterer Nutzen und Aufwand, welcher in Abschnitt 5.2 Qualitative Analyse betrachtet wird, fließt hier nicht ein.

Für das Erstellen der Testbögen werden hier initial drei Personentage (PTs) und für die Wartung jährlich ein halber PT gerechnet. Das Durchführen der Tests wird jährlich mit zweieinhalb

[1] Die Zeiten für Edge, Firefox und Internet Explorer sind Prognosen, welche aus dem Verhältnis der Laufzeiten auf einem Entwicklersystem berechnet sind.

[2] Der Faktor basiert auf der Anzahl weiterer Produkte und dem Vergleich des Funktionsumfangs.

PTs veranschlagt. Für das Cross-Browser-Testing wird der Aufwand für die Durchführung pro Browser auf zwei PTs reduziert. Dem gegenüber stehen initial 20 PTs für die Umsetzung der Tests für das Produkt. Der generelle Aufwand für das Erstellen der zentralen Page-Objects wurde abgezogen, diese fallen einmalig an und können in den Produkttests wieder genutzt werden. Pro Jahr ist eineinhalb PTs für die Wartung veranschlagt. Die automatisierten GUI-Tests müssen nicht manuell durchgeführt werden, so fallen keine PTs für die Durchführung an. Es ergibt sich bei einem Browser eine in Abbildung 5.1 prognostizierte Amortisationszeit von etwa elf Jahren. Diese verringert sich, wenn drei beziehungsweise zukünftig vier Browser getestet werden. Ein Aufwand den man manuell kaum leisten kann. Die Amortisationszeit verschiebt sich auf zweieinhalb bis dreieinhalb Jahre, wie in Abbildung 5.1 dargestellt.

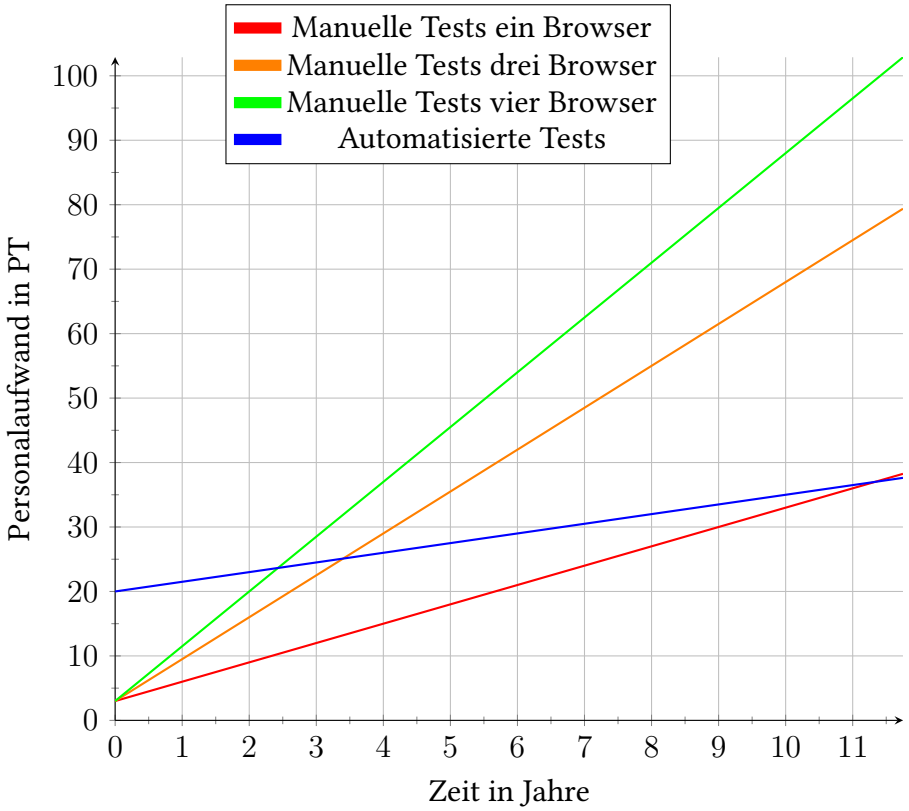

Abbildung 5.1: Die Amortisationszeit für die Testautomatisierung im Vergleich zum manuellen Testen mit einem, drei und vier Browsern.

5.2 Qualitative Analyse

Die qualitative Analyse untersucht verschiedene Punkte anhand der Einschätzungen von den zuständigen Mitarbeitern der AEB. Im ersten Teil werden die Anforderungen der AEB, welche

in einer vorangegangen Fachstudie erhoben wurden, untersucht. Gefolgt vom Aufwand durch Softwarewartung und dem Nutzen bei der Entwicklung und bei Releases.

Die qualitative Analyse des Nutzen-Aufwand-Verhältnisses wurde anhand der folgenden Leitfragen, die in diversen Besprechung diskutiert wurden, durchgeführt.

- Werden die von der AEB gestellten Anforderungen an automatisierte GUI-Tests erfüllt?
- Welcher Aufwand ergibt sich durch Wartung der Software?
- Welcher Nutzen wird bei Entwicklung und Releases erreicht?

Diese Fragestellungen wurden in den Besprechungen erörtert und prozesshaft erarbeitet. An den Besprechungen haben insgesamt zehn Mitarbeiter aus den Fachbereichen Frameworkentwicklung, Produktentwicklung und Qualitätssicherung in unterschiedlicher Intensität mitgewirkt. Diese Mitarbeiter sind als Softwarearchitekt, Produktmanager, Softwareentwickler beziehungsweise Tester bei der AEB tätig. Durch diese multidisziplinäre Gruppenkonstellationen, in verschiedenen Teams, konnten gemeinsame Analysen erarbeitet werden. Die verschiedenen Zuständigkeiten und Sichtweisen der Mitarbeiter wurden hierdurch in der Betrachtung abgedeckt. Die Ergebnisse wurden im Folgenden zusammengetragen.

5.2.1 Erfüllung der Anforderungen der AEB

Die Anforderungen der AEB an das GUI-Testing Framework, wurden im Rahmen der Fachstudie „Evaluierung von GUI-Testing Frameworks“ erhoben [RSG17]. Einige Eigenschaften der GUI-Tests stehen durch die Entscheidung der AEB für Selenide bereits fest. Noch offene Eigenschaften beziehungsweise besonders relevante Anforderungen werden im Folgenden bewertet.

Bewertung der übergreifenden Ziele

Zu den essentiellen Anforderungen gehört es, die *Kosten und Aufwand zu reduzieren*. Wie in Abbildung 5.1 zu sehen und in Abschnitt 5.1.2 quantitativ untersucht wurde, ist entscheidend, welcher Zeitraum und welche Anforderungen im Detail gestellt werden. Sofern hier die drei bis vier Browserplattformen betrachtet werden, findet diese Amortisation in etwa drei Jahren statt.

Die *Wartungskosten* können durch die Abstraktion mittels Page-Objects gesenkt werden. Diese sollten mit den ACs und ACVs zusammen gepflegt werden, dadurch fallen Änderungen nur an einer Stelle an. Viele Page-Objects können entsprechend von den Produktteams genutzt werden und die Automatisierung kann schneller erfolgen. Anpassungen an der Abstraktion müssen gegebenenfalls an mehreren Stellen nachgezogen werden. Im Rahmen dieser Bachelorarbeit war der Wartungsaufwand sehr gering und ließ sich auf wenige Minuten pro Woche beziffern.

Die *Qualitätssicherung bei Release* sicherzustellen, ist ein weiteres essentielles Ziel. Automatisierte GUI-Tests bieten hier den Vorteil des täglichen Testlaufs und unterstützen somit eine Korrektur noch vor der Testphase des Releases. Im Rahmen dieser Arbeit wurden durch Cross-Browser-Testing kritische Fehler bei der Nutzung des Internet Explorer entdeckt, die in Chrome, Firefox und Edge nicht aufgetaucht sind. Da dieser Browser offiziell von der Software unterstützt wird, konnte hierdurch erheblich zur Qualitätssicherung beigetragen werden. Ausführlicher wird dieser Aspekt in Abschnitt 5.2.3 behandelt.

Das *Testen von Abläufen* ist möglich und lässt sich durch Abstraktion gut implementieren. Sind die Page-Objects bereits vorhanden, lassen sich leichter Varianten der Abläufe bilden. Hierdurch konnten für die BrokerIntegration mehrere Exportvorgänge mit verschiedenen Abbrüchen und Korrekturen umgesetzt werden.

Als wichtiges Ziel wurde *GUI-Testprozesse zu standardisieren* ermittelt. Dies kann erst dann evaluiert werden, wenn die Page-Objects mit den ACs und ACVs gepflegt werden. Durch die zentral entwickelten Page-Objects wird eine Richtung vorgegeben. Die zentralen Page-Objects werden in den Produkttests genutzt und vereinheitlichen den Aufbau der GUI-Tests.

Testen von einzelnen *GUI-Elementen* ist durch Selenide bereits gegeben. Klicks oder Tastatureingaben können ausgelöst und Attribute sowie Properties ausgelesen werden. Zusätzlich können Screenshots angefertigt und mit Referenzen abgeglichen werden. Allerdings liegt der Schwerpunkt in dieser Evaluierung nicht auf einzelnen Elementen.

Performance Test wurden in Absprache ausgeklammert, hierfür sind GUI-Testing-Frameworks nicht ausgelegt. Es können mehrere Browser parallel gestartet und die Tests ausgeführt werden um damit mehrere simultane Benutzer zu simulieren. Um auf den Server eine wesentlich höhere Last zu erzeugen, wird ein entsprechend großes Selenium Grid benötigt. Besser ist es hierfür auf eine spezialisierte Lösung, wie JMeter, zu setzen, welche bereits in der AEB im Einsatz ist.

Während der Fachstudie wurde das *Testen von Darstellung* als eher unwichtig befunden. Im Laufe dieser Arbeit hat dieses Thema jedoch an Bedeutung gewonnen. In einer letzten Iteration ist die Möglichkeit zum Vergleich von Screenshots entstanden. Dieser Vergleich arbeitet auf Pixelebene. Durch einen Bildvergleich auf Musterebene könnte dies verbessert werden. Hier kann eine Folgearbeit zum Einsatz von OpenCV für Screenshotvergleiche erarbeitet werden. Möglicherweise einen speziellen Service, welcher die Referenzbilder verwaltet und mit welchem aktuelle Screenshots als Referenz übernommen werden könnten.

Erstellung von Tests

Eine essentielle Anforderung stellt die *Erstellung neuer Tests durch Entwickler* der AEB dar. Mit Hilfe der Programmiersprache Java kann dies gewährleistet werden, da Java ein gewohntes Werkzeug mit gewohnten Debuggingmöglichkeiten für die Entwickler darstellt.

Die *Abstraktionsmöglichkeiten* aus Tabelle 4.1 ergeben automatisch *Testbausteine*, für die Wiederverwendbarkeit in den Testfällen. Aufgrund der Programmierung in Java können die Methoden beliebig parametrisiert werden, um diese als Testbausteine flexibel einsetzen zu können. Zusätzlich wird durch die Abstraktion auch die Wartbarkeit der Testfälle verbessert, da Änderungen an der Anwendung nur wenige Änderungen an den GUI-Tests erfordern.

Als wichtige Anforderung wurden *Fachexperten* und *Testexperten* genannt. Diese Anforderung kann nicht direkt erfüllt werden. Um die automatisierten GUI-Tests zu erstellen, sind Programmierkenntnisse notwendig. Möglicherweise könnte diese Anforderung durch den Einsatz des Frameworks JBehave erfüllt werden. JBehave ist ein Testframework für Behavior Driven Development (BDD) und bei der AEB bereits produktiv im Einsatz. Es muss getrennt evaluiert werden, ob sich diese zusätzliche Abstraktionsebene an dieser Stelle lohnt.

Das *Recording von Tests* sollte nicht genutzt werden, da keine Abstraktionsmöglichkeiten vorhanden sind und somit Tests nach entsprechenden Änderungen neu aufgezeichnet werden müssten. Dies führt zu einem enormen Wartungsaufwand und wurde daher verworfen.

Ein *"Schritt-für-Schritt"-Debugmodus* ist durch die Debuggingfunktionalitäten von Eclipse gegeben. Es kann an einer beliebigen Stelle ein Breakpoint gesetzt werden und Variablenwerte inspiziert oder der Ablauf im Browser zeilenweise verfolgt werden.

Ablauf der Tests

Das *Ausführen in einem Build-System* ist eine essentielle Anforderung. Erste automatisierte GUI-Tests laufen bereits seit Anfang August in der Build-Pipeline mit Jenkins. Zu Beginn gab es Startschwierigkeiten aufgrund spezieller Anforderungen des neuen Headless-Modus im Chrome. Mit einem Update von Selenide und den eigenen GUI-Tests konnten diese erfolgreich überwunden werden. Das Cross-Browser-Testing konnte im Rahmen der Arbeit nicht auf dem Build-Server, sondern nur lokal, getestet werden. Das Reporting der Tests erfolgt durch Logs,

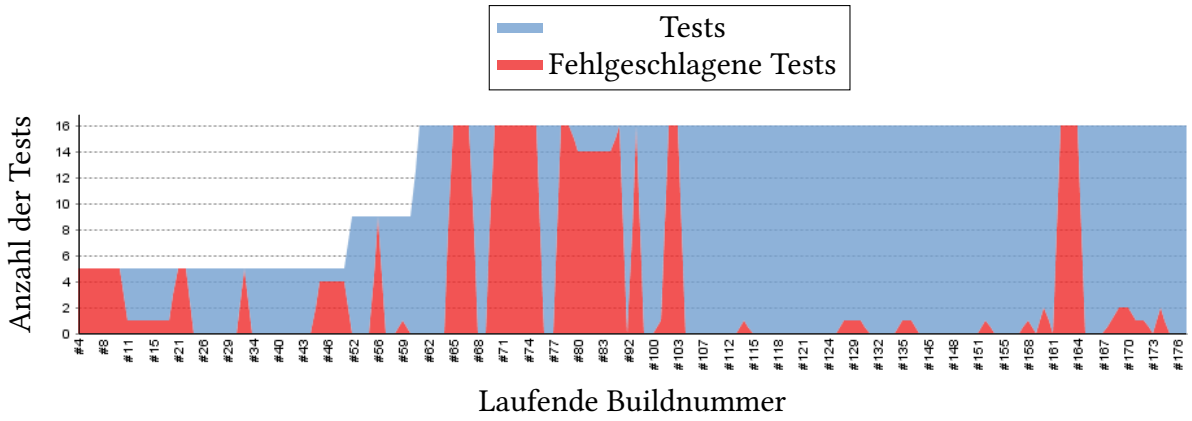

Abbildung 5.2: Der Verlauf von Testergebnissen der automatisierten GUI-Tests in Jenkins.

HTML-Dumps und Screenshots. Dieses Reporting kann bei Bedarf verbessert und übersichtlicher visualisiert werden. Dazu kann Allure oder ein HTML-Report zum Einsatz kommen [Sol16; Yan14]. Abbildung 5.2 zeigt den Testverlauf mit dem Chrome Browser.

Die Tests müssen *lokal beim Entwickler* ausführbar sein. Zum Einen erleichtert dies die Erstellung und zum Anderen wird das Debugging einfacher. Die umgesetzten GUI-Tests laufen in *„echten" lokalen Browsern*, mindestens auf den vier Browsern Chrome, Edge, Firefox und Internet Explorer. Als zusätzliche Softwarekomponente ist hier lediglich eine Installation des dafür passenden WebDrivers erforderlich. Es kann der *Testablauf sichtbar im Browser* verfolgt und nachvollzogen werden.

Tests mit Chrome und Firefox funktionieren im Headless-Modus und ermöglichen so das Weiterarbeiten, während die Tests lokal ausgeführt werden, vergleichbar mit klassischen Unittests. Edge und Internet Explorer ermöglichen keinen Betrieb im Headless-Modus, diese öffnen sich im Vordergrund. Die wichtige Anforderung *headless Browser* ist nur für die Browser Chrome und Firefox erfüllt. In der praktischen Anwendung ist dies ausreichend.

5.2.2 Aufwand durch Softwarewartung

In diesem Teil wird untersucht, welche Änderungen entsprechende Anpassungen der GUI-Tests erforderlich machen und somit zu einem Mehraufwand führen. Neue Texte oder Anpassungen der bisherigen Übersetzung, haben keine Auswirkungen auf die GUI-Tests. Wird auf angezeigte Texte zurückgegriffen, wird ein zentrales Modul genutzt um die Texte in der richtigen Sprache zu lesen. Sollen landesspezifische Eingaben genutzt werden, beispielsweise für ein Datum, müssen diese entsprechend der eingestellten Lokalisierung formatiert werden.

Bei Änderungen der Produkt-, Wizard- oder BPMN-Konfiguration oder Ablaufänderungen durch veränderten Code, müssen die GUI-Tests angepasst werden. In solchen Fällen sollen die bereits implementierten Tests Veränderungen detektieren und anzeigen.

Gibt es Veränderungen am GUI-Framework, müssen die betroffenen Page-Objects angepasst werden. Sind es keine Änderung der Schnittstelle, sondern nur interne Veränderungen, beispielsweise von Selektoren, sind keine Änderungen in den Testsuites der Produkte notwendig.

Gewisse Änderungen des Programmcodes bringen erwarteterweise die Notwendigkeit einer Anpassung der Testsuite mit sich. Eine Anpassung sollte nur an der Stelle nötig werden, an der das Framework oder das Produkt verändert wurde. Mit Hilfe der Page-Objects und der Abstrahierung kann dies auf wenige Änderungen eingegrenzt werden.

5.2.3 Nutzen bei Entwicklung und Releases

Weiterer Nutzen bei der Entwicklung und für Releases der Software kann durch verschiedene, teilweise langfristige, Effekte erzielt werden. Im Folgenden werden diese anhand verschiedener Fragestellungen betrachtet.

Werden Fehler entdeckt?

Mit Cross-Browser-Testing können Fehler direkt bei der Erstellung der Tests und Ausführung mit verschiedenen Browsern entdeckt werden. Durch die Integration der GUI-Tests in die Build-Pipeline werden Regressionstests durchgeführt, welche ungewollte Veränderungen aufdecken können. Im Rahmen der Testautomatisierung wurden zwei kritische Fehler entdeckt, die das Produkt in einem Browser unbedienbar machten. Des weiteren wurden wenige unkritische Fehler entdeckt, durch die geringe Auswirkungen auf die Nutzbarkeit des Produktes zu erwarten waren. Hierdurch entstand ein eindeutiger Nutzen für das Produkt.

Wird sichergestellt, dass funktionierende Software ausgeliefert wird?

Weder durch manuelle noch durch automatisierte Tests kann sichergestellt werden, dass die Software durchweg funktioniert. Für die getesteten Funktionen kann dennoch gesagt werden, dass die Software wie getestet funktioniert und ausgeliefert werden kann. Somit ist sichergestellt, dass die getesteten Anwendungsfälle auch beim Kunden funktionieren. Durch Cross-Browser-Testing gilt dies für alle unterstützten Browser.

Entsteht ein Vorteil durch die erhöhte Feedbackfrequenz?

Es konnte bereits ein Vorteil durch die höhere Frequenz der Tests festgestellt werden. Noch vor der manuellen Testphase wurden Fehler entdeckt. Mögliche Wechselwirkungen bei der Weiterentwicklung können durch die automatisierten Tests früher erkannt werden. Somit muss nicht während der Testphase vor einem Release ein Bugfix erstellt werden. Aus Entwicklersicht vereinfacht dies die Fehlerbehebung, da man erst vor kurzem am entsprechenden Code gearbeitet hat.

Ist das Feedback hilfreich für die Entwickler?

Wie hilfreich das Feedback durch automatisierte GUI-Tests für Entwickler ist, kann noch nicht abschließend beurteilt werden. Teilweise kommt es vor, dass ein Test nicht durchläuft, obwohl der Prüfling in Ordnung ist. Während den Tests werden Logfiles und Screenshots erstellt, anhand diesen kann anschließend der Fehler ermittelt werden. Für eine schnellere

Erfassung, warum ein Test fehlgeschlagen ist, kann die Aufbereitung der Testergebnisse verbessert werden.

Steigt das Vertrauen und Sicherheitsgefühl der Entwickler in die Testsuite?

Ob das Vertrauen und Sicherheitsgefühl der Entwickler in die Testsuite steigt, konnte in diesem zeitlich begrenzten Rahmen nicht untersucht werden. Eine zuverlässige Testsuite, bestehend aus Unit-, Integrations- und Systemtests, erleichtert das Refactoring der Software [LL13]. Durch Refactoring wird langfristig die Softwareerosion verringert und die Wartbarkeit erhöht [Kle12]. Eine Testsuite unterstützt außerdem bei der Implementierung neuer Features, da Seiteneffekte erkannt werden können.

Welche weiteren Effekte sind möglich?

Als Nebeneffekt dokumentieren die GUI-Tests stets den aktuellen Softwarestand und das erwartete Verhalten. Sollte die Technologie der Benutzeroberfläche geändert werden, können möglicherweise die Prozesse erhalten bleiben und die Page-Objects ersetzt werden.

Durch eine gute Testabdeckung und automatisierte Ausführung kann bei Bedarf die Frequenz für Releases erhöht werden. Der manuelle Testaufwand steigt nicht linear mit der Anzahl der Releases an. Dies ist ein weiterer Schritt Richtung Continuous Delivery.

5.3 Zusammenfassung

In der quantitativen Analyse konnte eine Amortisationszeit berechnet werden. Bis die Strukturen für automatisierte GUI-Tests geschaffen und die Prozesse etabliert sind, lohnt sich die Automatisierung aus zeitlicher Perspektive, im Vergleich zum manuellen Testen, nicht. Die Prognose der Amortisationszeit für Tests mit einem Browser beläuft sich auf elf Jahre. Werden hingegen drei bis vier Browser für die GUI-Tests genutzt, reduziert sich diese Zeit auf etwa drei Jahre.

Automatisierte GUI-Tests sind jedoch als langfristige Investition zu sehen, die nicht ausschließlich auf eine Zeitersparnis bei der Durchführung der Tests abzielt. Dieser Effekt ist wünschenswert, allerdings nur ein Teil des möglichen Nutzens. Deshalb erfolgte ergänzend eine qualitative Analyse um den nicht direkt messbaren Nutzen und Aufwand zu bewerten. Oft sind weitere Faktoren der eigentliche Grund für diese Investition [FG99]. Alle essentiellen Anforderungen der AEB konnten erfüllt werden, wodurch bereits ein Nutzen entstanden ist. Durch die automatisierten GUI-Tests wurden Fehler entdeckt, welche beim manuellen Testen nicht festgestellt wurden. Da nach jedem Testlauf ersichtlich ist, ob die Software funktioniert, wird die Qualitätssicherung für ein Release unterstützt. Ergänzt durch Cross-Browser-Testing, kann dies für alle unterstützten Browser automatisiert erfolgen. Die Abstraktionsschichten der

automatisierten GUI-Tests ermöglichen die Wiederverwendbarkeit einzelner Aktionen und können als Bausteine in weiteren Testfällen genutzt werden. Durch diese Abstraktionsmöglichkeit entsteht beispielsweise bei Änderungen von Übersetzungen kein Wartungsaufwand. In Besprechungen hat sich eine Veränderung des Bedarfs für das Testen der Darstellung herausgestellt. Daher wurde ein Vergleich von Referenz- und aktuellem Screenshot entwickelt, der das automatisierte Testen der Darstellung ermöglicht. Eine umfassende Testsuite, welche die drei Ebenen der Testpyramide abdeckt, kann die Bereitschaft zu Refactorings steigern und die Wartbarkeit der Software verbessern. Hilfreiches und schnelles Feedback für die Entwickler vereinfacht und beschleunigt die Weiterentwicklung der Software und reduziert den manuellen Testaufwand während der Entwicklung.

Die quantiative und qualitative Analyse zeigen, dass die Testautomatisierung für die GUI initial zu einem hohen Aufwand führt. Ist die Implementierung der automatisierten GUI-Tests gut abstrahiert, lässt sich der dauerhafte Aufwand für die Erstellung und Wartung der Tests reduzieren. Die Testautomatisierung erfordert eine hohe initiale Investition, welche einen langfristigen Nutzen verspricht.

6 Zusammenfassung und Ausblick

Dieses Kapitel umfasst eine Zusammenfassung der Arbeit, Handlungsempfehlungen für die AEB und anschließend weiterführende Themen für folgende Arbeiten.

6.1 Zusammenfassung

Zu Beginn dieser Arbeit wurden die Grundlagen für Qualitätssicherung und Automatisierung von Software- und speziell GUI-Tests betrachtet. Anschließend wurden die Bewertungsgrundlagen für die Automatisierung der GUI-Tests entwickelt und definiert. Diese wurden in eine quantitative und eine qualitative Analyse aufgeteilt. Folgend wurde die Automatisierung der GUI-Tests für das gewählte Produkt BrokerIntegration beschrieben. Die Testautomatisierung begann mit der Auswahl und Priorisierung der Tests für die Umsetzung. Während der Implementierung konnten etwa 70 Prozent der manuellen Testbögen automatisiert werden. Hierzu wurde die Build-Pipeline um die automatisierten GUI-Tests mit dem Chrome Browser ergänzt. Im weiteren Verlauf der Bachelorarbeit erfolgte eine Optimierung dieser automatisierten GUI-Tests für das Cross-Browser-Testing. Hierdurch wurden die automatisierten Tests auch für die Browser Edge, Firefox und Internet Explorer ermöglicht.

Die anschließende Analyse des Nutzen-Aufwand-Verhältnisses vergleicht Nutzen und Aufwand für eine Automatisierung der GUI-Tests. Aufgeteilt ist die Analyse in einen quantitativen und eine qualitativen Teil. Quantitativ wurde der Bereich des Nutzens durch das Reduzieren des manuellen Testaufwands mit dem Aufwand für die Testautomatisierung verglichen. Diese Prognose der Amortisationszeit wurde in Abbildung 5.1 berechnet. Für die Datenbasis ergeben sich bei manuellen Tests mit einem Browser elf Jahre. Wird mit drei oder vier Browsern getestet, verkürzt sich die Amortisierung auf etwa drei Jahre.

Eine rein quantitative Analyse spiegelt jedoch das Nutzen-Aufwand-Verhältnis nur begrenzt wider. Das Nutzen-Aufwand-Verhältnis sollte nicht auf die direkt messbaren Aspekte begrenzt bleiben und wird deshalb durch eine qualitative Analyse ergänzt. Automatisierte GUI-Tests sind eher als langfristige Investition zu sehen, die nicht ausschließlich auf eine Zeitersparnis bei der Durchführung der Tests zielt. Oft sind weitere Faktoren der eigentliche Grund für diese Investition [FG99]. Es konnten alle essentiellen Anforderungen der AEB erfüllt werden, wodurch bereits ein Nutzen entstand. Fehler, welche beim manuellen Testen nicht entdeckt wurden, konnten mit Hilfe der automatisierten GUI-Tests gefunden werden. Durch

Cross-Browser-Testing kann, für alle unterstützten Browser, eine automatisierte Unterstützung der Qualitätssicherung bei Releases erfolgen. Die Wiederverwendbarkeit kann durch Abstraktionsmöglichkeiten verbessert werden. Ergänzend führen die Abstraktionsschichten zu verringertem Wartungsaufwand bei Veränderung der Software. Eine umfassende Testsuite kann beispielsweise die Wartbarkeit der Software verbessern. Zudem kann durch das automatische Feedback die Weiterentwicklung der Software beschleunigt und vereinfacht werden. Auch der manuelle Testaufwand während der Entwicklung kann aufgrund der Automatisierung reduziert werden.

Durch die quantiative und qualitative Analyse zeigt sich, dass eine Testautomatisierung einen hohen Aufwand bedeuten. Der dauerhafte Aufwand für die Erstellung und Wartung der Tests kann durch gute Abstraktion und robuste Selektoren reduziert werden. Verallgemeinernd kann gesagt werden, dass Testautomatisierung eine hohe initiale Investition bedeutet und einen langfristigen Nutzen verspricht.

6.2 Handlungsempfehlungen

Wichtig für die Wartbarkeit der GUI-Tests ist die Abstraktion. Mit dem Page-Object-Pattern können Selektoren gekapselt und wiederverwendet werden. Dies ermöglicht eine gute Wartbarkeit der Tests und vermindert den Aufwand für die Pflege. Werden außerdem Aktionen gekapselt, können diese als Bausteine in verschiedenen Testfällen genutzt und leichter Variationen der Testfälle erstellt werden. Die Pflege der Page-Objects sollte daher mit den zugehörigen ACs und ACVs erfolgen. So wird die Grundlage für GUI-Tests mit dem GUI-Framework bereitgestellt. Die Produktteams können durch diese Strukturierung der Tests die Page-Objects direkt nutzen, wodurch eine leichtere Einführung der GUI-Tests in den Produkten ermöglicht wird. Durch die zentrale Pflege dieser Page-Objects, wird der Wartungsaufwand für die Codebasis reduziert.

Es wird empfohlen, ein dediziertes Testsystem für die Testautomatisierung zu nutzen. Bei Bedarf kann dies zu Beginn der Tests gestartet und nach den Tests beendet werden. Die Datenbank kann zu Beginn der Tests zurückgesetzt werden und es stehen direkt entsprechende Testdaten zur Verfügung. Bei der Browserautomatisierung können Timingprobleme auftreten. Beispielsweise wenn auf ein GUI-Element geklickt wird, welches noch nicht vollständig geladen wurde und deshalb nicht wie gewünscht auf den Klick reagiert. Sind Tests unabhängig und redundante Testdaten vorhanden, können diese Tests bei einem Fehlschlag wiederholt werden. Ein erneuter Versuch einen Test auszuführen reduziert Fehlschläge durch diese Timingprobleme. Dazu kann das Maven Surfire Plugin mit der Einstellung `surefire.rerunFailingTestsCount` auf die gewünschte Anzahl gesetzt werden. Dies verringert die Zahl der falsch positiven Tests und vermeidet damit direkt eine unnötige Fehlersuche.

Eine weitere Empfehlung ist, die Durchführung des Cross-Browser-Testing in regelmäßigen Abständen zu automatisieren. Anstatt Cross-Browser-Tests täglich auszuführen, kann die

Durchführung bei Bedarf auf einen wöchentlichen Rhythmus reduziert werden. Dagegen könnten die Testintervalle verkürzt werden. Vom geplanten Durchführen der Tests kann auf ein Auslösen durch bestimmte Aktionen umgestellt werden und beispielsweise nach jedem Commit, die Testsuite inklusive GUI-Tests mit einem Browser, ausführen. Hierfür sollte der Chrome Browser zum Einsatz kommen, welcher aktuell die beste Performance und Stabilität aufweist. Durch diese Anpassungen wird die Build-Pipeline weiter in Richtung Continuous Delivery ausgebaut.

Als nächster Schritt wird die Verbesserung des Reportings empfohlen. Durch ein verbessertes Reporting, beispielsweise mit Allure, kann der Zugriff auf die Logfiles und Screenshots vereinfacht werden [Sol16; Yan14]. So kann schneller erfasst werden, weshalb ein Test fehlgeschlagen ist.

Im Laufe der Bachelorarbeit wurde Vaadin 7 auf Vaadin 8 aktualisiert. Hierdurch entstand der Wunsch, auch Screenshotregressionstests durchführen zu können. Dies wurde in einer ersten Version in die Tests implementiert. Hierfür könnte auch ein selbstgehosteter Webservice für den Vergleich der Screenshots genutzt werden. Diese Funktionen werden beispielsweise von der Open-Source-Software Spectre implementiert. Eine weitere Idee hierzu ist der in Abschnitt 6.3 dargestellte Vergleich der Screenshots auf Musterebene. Hierbei werden nicht für jeden Browser unterschiedliche Referenzbilder benötigt und somit die Wartung der Referenzen deutlich vereinfacht.

Alternativ zum eigenen Bereitstellen der Testumgebung mit Browser und WebDriver, könnte auch eine Cloudlösung wie CrossBrowserTesting oder Souce Labs zum Einsatz kommen. Dies würde den Pflegeaufwand der Testumgebung und regelmäßige Updates reduzieren.

Abschließend empfehle ich den Einsatz von automatisierten GUI-Tests in weiteren Produkten. Die quantitative und qualitative Analyse zeigen, dass der erwartete Nutzen den Aufwand übersteigt. Es ist verallgemeinernd anzunehmen, dass sich dies auch bei anderen Softwareprodukten in ähnlicher Weise auswirkt.

6.3 Ausblick und zukünftige Arbeiten

In dieser Bachelorarbeit wurden erste GUI-Tests bei der AEB automatisiert und das Nutzen-Aufwand-Verhältnis analysiert. Dabei wurden weitere Themen angeschnitten, aus denen zukünftige Arbeiten hervorgehen können. Diese werden im Folgenden vorgestellt.

Direkt aufbauend auf diese Bachelorarbeit könnte in einiger Zeit die unternehmensweite Einführung der GUI-Tests bewertet werden. Hierbei können weitere Aspekte, welche bisher qualitativ untersucht wurden, quantitativ ausgewertet und analysiert werden. Außerdem kann überprüft werden, ob die quantitative Bewertung auch für die unternehmensweite Einführung skaliert.

Ein nächstes Thema könnte die Evaluierung unterschiedlicher Vergleichsmöglichkeiten für Screenshots sein. Dies können Bibliotheken sein, welche direkt in den Testfällen genutzt werden können, oder dedizierte Services mit eigenen Benutzeroberflächen, wie beispielsweise Spectre. Hierbei können die folgenden Faktoren interessant sein: der Funktionsumfang, Eignung für verschiedene Browser, die Bedienbarkeit durch Tester und die Versionierung der Referenzscreenshots.

Eine weitere Arbeit könnte sich mit Screenshotvergleichen auf Musterbasis beschäftigen. Die Darstellungen von Webanwendungen weichen in verschiedenen Browsern minimal ab, meist nur wenige Pixel, die für Menschen kaum wahrnehmbar sind. Um diese Toleranzen in Screenshotvergleichen und somit in einer Testsuite zu kompensieren, könnten ähnlich eines Fingerabdrucks die Muster einer Webseite betrachtet werden. Hierbei könnten Elemente lokalisiert und anschließend deren Positionen verglichen werden. Dieser Vergleich auf Musterbasis wäre ideal um einen Referenzscreenshot für alle verwendeten Browser zu nutzen.

Literaturverzeichnis

[16] *Edge webdriver bug: <input type=file> not working*. 2016. URL: https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/6052385/ (zitiert auf S. 33).

[17a] *Selenium - Web Browser Automation*. 29. Nov. 2017. URL: http://www.seleniumhq.org/ (zitiert auf S. 21, 22).

[17b] *Selenium Downloads*. 29. Nov. 2017. URL: http://www.seleniumhq.org/download/ (zitiert auf S. 22, 34).

[17c] *Selenium IDE*. 20. Nov. 2017. URL: http://docs.seleniumhq.org/projects/ide/ (zitiert auf S. 23).

[17d] *Selenium Remote Control*. 20. Nov. 2017. URL: http://www.seleniumhq.org/projects/remote-control/ (zitiert auf S. 21).

[AEB17a] AEB. *AEB - Software, Beratung und Services für Außenwirtschaft und Logistik*. 20. Aug. 2017. URL: https://www.aeb.com/de/ueber-aeb/index.php (zitiert auf S. 13).

[AEB17b] AEB. *Benutzerleitfaden ASSIST4 Broker Integration*. 2017 (zitiert auf S. 29).

[AEB17c] AEB. *Systemvoraussetzungen ASSIST4 AEB Engines*. 2017 (zitiert auf S. 32).

[BF14] P. Bourque, R. E. Fairley. *SWEBOK v3.0 - Guide to the Software Engineering Body of Knowledge*. 2014 (zitiert auf S. 19–21).

[BS+17] D. Burns, S. Stewart et al. *WebDriver*. 30. März 2017. URL: https://www.w3.org/TR/webdriver/ (zitiert auf S. 22).

[Cod17] Codeborne. *Documentation*. 22. Nov. 2017. URL: http://selenide.org/documentation.html (zitiert auf S. 30).

[Coh09] M. Cohn. *The Forgotten Layer of the Test Automation Pyramid*. 17. Dez. 2009. URL: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid (zitiert auf S. 19, 20).

[Dev17a] DeveloperExpress. *Frequently Asked Questions - TestCafe*. 2017. URL: https://devexpress.github.io/testcafe/faq/ (zitiert auf S. 21).

[Dev17b] DeveloperExpress. *Getting Started Guide*. 2017. URL: https://testcafe.devexpress.com/Documentation/Getting_Started/Getting_Started_Guide/ (zitiert auf S. 23).

[Dev17c] A. Deveria. *Can I use ... Support tables for HTML5, CSS3, etc.* 29. Okt. 2017. URL: https://caniuse.com/#compare=ie+11,edge+16,firefox+57,chrome+62 (zitiert auf S. 23).

[Dij70] E. W. Dijkstra. *EwD 249 Notes on Structured Programming.* 1970 (zitiert auf S. 17, 18).

[FG99] M. Fewster, D. Graham. *Software Test Automation.* 1999 (zitiert auf S. 18, 21, 45, 47).

[Fow12] M. Fowler. *TestPyramid.* 1. Mai 2012. URL: https://martinfowler.com/bliki/TestPyramid.html (zitiert auf S. 18–20).

[Fow13a] M. Fowler. *ContinuousDelivery.* 30. Mai 2013. URL: https://martinfowler.com/bliki/ContinuousDelivery.html (zitiert auf S. 20).

[Fow13b] M. Fowler. *PageObject.* 10. Sep. 2013. URL: https://martinfowler.com/bliki/PageObject.html (zitiert auf S. 23).

[Gen17] Generalzolldirektion. *Vertretung.* 28. Aug. 2017. URL: http://www.zoll.de/DE/Fachthemen/Zoelle/Zollanmeldung/Vertretung/vertretung_node.html (zitiert auf S. 29).

[Goo17] Google. *ChromeDriver - WebDriver for Chrome.* 3. Okt. 2017. URL: https://sites.google.com/a/chromium.org/chromedriver/ (zitiert auf S. 22).

[Ham10] T. Hampp. „Ein Kosten-Nutzen-Modell für die Softwareprüfung“. Diss. Institut für Softwaretechnologie der Universität Stuttgart, 2010 (zitiert auf S. 18, 25).

[Ham16] M. Hammoudi. „Why Do Record / Replay Tests of Web Applications Break?“ Magisterarb. 3. Mai 2016 (zitiert auf S. 23).

[Hof12] D. W. Hoffmann. *Software-Qualität.* 2012 (zitiert auf S. 17, 18, 20).

[Kle12] S. Kleuker. *Qualitätssicherung durch Softwaretests. Vorgehensweisen und Werkzeuge zum Test von Java-Programmen.* 2012 (zitiert auf S. 13, 17, 18, 20, 21, 45).

[LL13] J. Ludewig, H. Lichter. *Software Engineering. Grundlagen, Menschen, Prozesse, Techniken.* 2013 (zitiert auf S. 17, 19, 20, 45).

[Mic16] Microsoft. *Edge in Windows Server 2016.* 27. Okt. 2016. URL: https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/9556604/ (zitiert auf S. 35).

[Mic17a] Microsoft. 2017. URL: https://www.microsoft.com/de-de/windows/microsoft-edge (zitiert auf S. 35).

[Mic17b] Microsoft. *WebDriver.* 27. Aug. 2017. URL: https://docs.microsoft.com/en-us/microsoft-edge/dev-guide/tools/webdriver (zitiert auf S. 22).

[Mos17] A. Moskovkin. *TestCafe: An e2e Testing Tool That Doesn't Use Selenium.* 29. Sep. 2017. URL: https://dzone.com/articles/testcafe-e2e-testing-tool (zitiert auf S. 21).

[Moz17] Mozilla. *Geckodriver.* 13. Okt. 2017. URL: https://github.com/mozilla/geckodriver/ (zitiert auf S. 22).

[Ora17] Oracle. *JavaMail Reference Implementation*. 2017. URL: https://javaee.github.io/javamail/ (zitiert auf S. 30).

[RDS17] J. Robie, M. Dyck, J. Spiegel. *XML Path Language (XPath) 3.1*. 21. März 2017. URL: https://www.w3.org/TR/xpath-3/ (zitiert auf S. 30).

[Res17] R. Resch. *Browser-Benchmark: Mit diesen Browsern surfen Sie am schnellsten*. 14. Nov. 2017. URL: https://www.pcwelt.de/produkte/Mit_diesem_Browser_surfen_Sie_am_schnellsten-Benchmark-8760952.html (zitiert auf S. 34).

[RSG17] N. Rusam, A. Stifel, M. Graf. „Evaluierung von GUI Testing Frameworks“. Forschungsber. Institut für Softwaretechnologie, Universität Stuttgart, 3. Mai 2017 (zitiert auf S. 13, 17–23, 26, 30).

[Sco12] A. Scott. *Introducing the software testing ice-cream cone (anti-pattern)*. 31. Jan. 2012. URL: https://watirmelon.blog/2012/01/31/introducing-the-software-testing-ice-cream-cone/ (zitiert auf S. 19–21).

[Sol16] A. Solntsev. *Selenide + Allure + JUnit example*. 6. Apr. 2016. URL: https://github.com/selenide-examples/selenide-allure-junit (zitiert auf S. 43, 49).

[Ste17] S. Stewart. *The Architecture of Open Source Applications: Selenium WebDriver*. 10. Sep. 2017. URL: http://www.aosabook.org/en/selenium.html (zitiert auf S. 22).

[Wac15] M. Wacker. *Just Say No to More End-to-End Tests*. 22. Apr. 2015. URL: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html (zitiert auf S. 18–20).

[WeA17] We-Are-Friday. *Spectre*. 14. Juni 2017. URL: https://github.com/wearefriday/spectre (zitiert auf S. 36).

[Yan14] Yandex. *Allure Test Report*. 10. Okt. 2014. URL: https://ci.qameta.io/job/allure2/job/master/Demo_Report/index.html (zitiert auf S. 43, 49).

Alle URLs wurden zuletzt am 01. 12. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift