Integration eines neuen Tools zur Automatisierung von Storefront Tests für ein Projekt der dotSource GmbH

	I	II	III	IV
Projektarbeit Nr.	☒	☐	☐	☐

vorgelegt am: 13.05.2020

von: ■■■■■■■■■■

Matrikelnr.: ■■■■■■■■■■

Campus: ■■■■■■■■■■

Studienbereich: ■■■■■■■■■■

Studiengang: ■■■■■■■■■■

Kurs: ■■■■■■■■■■

Ausbildungsstätte: dotSource GmbH

Betreuer: ■■■■■■■■■■

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AST	Automatisierte Storefront Tests
CI	Continuous Integration
CJS	CodeceptJS
DS	dotSource GmbH
■■■	■■■■■■■■■■
■■■	■■■■■■■■■■
OS	Online Shop
QC	Quellcode
SF	Storefront
SFT	Storefront Tests
TF	Testfall
TR	TestRail
XLT	XLT Script Developer

1 Einleitung

1.1 Ausgangspunkt

Die dotSource GmbH (DS) möchte für ihr Projekt ■■■■■■■■■■ (■■■) ein neues Tool zur Automatisierung von Storefront Tests (SFT) einführen. Dabei soll ermittelt werden, ob der Umstieg mit wenig Aufwand umzusetzen ist und ob es zu Verbesserungen hinsichtlich Entwicklungszeit, Wart- und Lesbarkeit des Quellcodes (QC) führen wird.

1.2 Inhalte dieser Arbeit

In dieser Arbeit wird die Bedeutung der Tests sowie der Grund für den Toolwechsel erklärt. Dabei werden die benötigten Arbeitsschritte zur Integration des neuen Tools geschildert. Dazu wird die Struktur des Tools und aller verwendeten Systeme erläutert. Die Arbeit befasst sich zuletzt mit einem Vergleich beider verwendeten Tools anhand eines Experiments. Der genaue Lösungsansatz eines ähnlichen Problems für ein anderes Projekt der DS im Detail ist nicht Bestandteil dieser Arbeit. Da Konfigurationsdateien von diesem anderen Projekt übernommen werden können, wird in dieser Arbeit nicht weitgehend auf die Konfiguration des neuen Tools eingegangen.

1.3 Wissenschaftliche Relevanz der Arbeit

Die Integration des neuen Tools wurde bereits in einem anderen Projekt der DS durchgeführt. Es wurde jedoch nicht analysiert, ob und inwiefern sich das Ablösen des alten Tools für die Entwickler lohnt. Des Weiteren ist unbekannt, ob sich Konfigurationsdateien vollständig oder teilweise übernehmen lassen. Ein solcher Prozess in der DS ist bislang undokumentiert.

1.4 Praktische Relevanz der Arbeit

Das ■■■ verwendet Continuous Integration zur Verwaltung des QC. Dies ist eine Methode zur Softwareentwicklung, bei der gewöhnlicherweise jeder Entwickler täglich neue

Funktionen integriert.[1] Um die Funktionsweise der implementierten Funktionen zu gewährleisten, müssen sie regelmäßig getestet werden.
Der Zeitliche Aufwand alle möglichen Testfälle manuell durchzuprüfen wäre nach jeder Änderung des QC eines Projektes für eine Person viel zu hoch. Daher müssen sie automatisch von einer Maschine durchgeführt werden. Bei der DS sollen diese Tests täglich ausgeführt werden. Der Release-Verantwortliche eines Projekts überprüft mit einem Reporting-Tool alle Testergebnisse und achtet insbesondere auf negative Resultate.

Die SFTs helfen, den ordnungsgemäßen Ablauf von Online Shop (OS) Prozessen zu gewährleisten. Zu solchen Prozessen zählen beispielsweise Anmeldungen, Artikel in Warenkörbe legen, Bestellung von Artikeln usw. Falls hierbei Fehler vorhanden sind, können Kunden eventuell keine Käufe mehr betätigen.

Dies kann den Verlust von mehreren tausenden Euro Umsatz täglich für den Betreiber des OS bedeuten, bis der Fehler behoben wird. Das Schreiben und regelmäßige Ausführen von Tests verringert die Wahrscheinlichkeit, dass so ein Fall auftreten wird.

1.5 Grundlage der Arbeit

Die Grundlage der Arbeit bildet die bereits erfolgreiche Integration des neuen Tools für das Projekt ■■■■■■ der DS. (■■■) Die Vorgehensweise zur Integration des neuen Tools für ■■■ soll ähnlich verlaufen und eventuelle Probleme dadurch im Voraus reduzieren.

1.6 Arbeitsziel

Bisher wird ein Tool zur Erstellung von Automatisierten Storefront Tests (AST) namens XLT Script Developer (XLT) verwendet. Jedoch nutzt DS nicht den Standard Script Developer von XLT, sondern einen angepassten. Daher müssen auch die Updates angepasst werden.[2] XLT wurde von den Entwicklern der DS als veraltet und die Verwendung als umständlich angesehen.

[1] [Fow00] S. 1
[2] [Dom19b] S. 1

Es wurde daher nach einer besseren Lösung gesucht und es wurde beschlossen, ein neues Tool namens CodeceptJS (CSJ) in Zukunft zu verwenden, um das Schreiben von AST zu vereinfachen. Jedoch möchte man, dass die alten mit XLT geschriebenen Tests und das bisherige Tool zur Überwachung aller Reports erhalten bleiben. Daher müssen die Ergebnisse von CSJ Tests auf das bereits bestehende Reporting Tool übertragen werden. Das Ziel soll dabei sein, herauszufinden ob und inwiefern das Schreiben von zukünftigen Automatisierten SFT für die Entwickler im ■■■ mit dem neuen Tool erleichtert wird. Dies soll Anhand von Vergleichen von Kenngrößen bei der Verwendung beider Tools erfolgen. Auf folgende Kenngrößen wird sich hierbei bezogen:

- Entwicklungszeit zur Erstellung neuer Tests
- Installationszeiten der Tools
- Les- und Wartbarkeit des Geschriebenen Quellcodes

1.7 Forschungsfragen

Diese Forschungsfragen helfen bei der Ermittlung des Nutzens von CJS für die DS:

- Wie schnell kann CJS für ein neues Projekt aufgesetzt werden?
- Wie einfach ist die Einarbeit in das neue Tool für einen Entwickler?
- Ist die Wartung des QC für den Entwickler mit CJS einfacher?
- Liefert CJS hilfreichere Testergebnisse als die vorhandenen XLT Tests?

1.8 Annahmen über die Ergebnisse der Integration des neuen Tools

- Die Verwendung von CJS für TA spart mehr Entwicklungszeit ein als XLT.
- Die Verwendung von CJS erzeugt einfacher lesbaren Code als XLT.
- Die Verwendung von CJS erzeugt detailliertere Fehlerausgaben beim Fehlschlagen eines Tests als XLT.

1.9 Methoden zur Beantwortung der Forschungsfragen

Zur Ermittlung des Zeitaufwandes beim Schreiben neuer Tests wird ein Experiment durchgeführt, bei dem ein Testfall (TF) sowohl mit XLT als auch mit CJS geschrieben wird. Dabei wird die Zeit bis zum Abschließen dieser Aufgabe gemessen und verglichen. Die Lesbarkeit von Testergebnissen und des QC wird durch Befragungen der Entwickler ermittelt.

Für das Verständnis der Sachverhalte werden im nächsten Kapitel alle verwendeten Systeme und ihre Verhältnisse zueinander aufgezeigt.

2 Theoretischer Teil

2.1 Verwendete Systeme und Kernbegriffe

Für das Verständnis dieser Arbeit sind folgende Kernbegriffe hilfreich:

Begriff	Erklärung
Storefront	Eine Storefront (SF) beschreibt die Ansicht, die ein Nutzer im Browser erhält, wenn er einen OS aufruft. Sämtliche Nutzung eines OS über einen Web Browser (z.B. Sucheingaben, Mausklicks, Tastendrücke) laufen über die SF.
Report	Als Report werden zusammengefasste Ausgabedaten von Testergebnissen bezeichnet.
Git	Git ist ein Programm zur Dateiversionsverwaltung.[3]
CodeceptJS	CodeceptJS ist ein modernes Framework zur Ausführung von Tests mit einer besonderen verhaltensgetriebenen Syntax.[4]
Jenkins	Jenkins ist ein in sich geschlossener Server für die Automatisierung von Test-, Lieferungs-, Bereitstellungs- und Build-Prozessen.[5]
TestRail	TestRail (TR) ist eine Anwendung für die Verwaltung von Tests.[6]
XLT	XLT Script Developer ist eine Firefox Erweiterung zur Erstellung von TF Skripten. [7]
Node.js	Node.js ist eine Server Umgebung, welche das Ausführen von JavaScript erlaubt.[8]

Tab. 1: Kernbegriffe dieser Arbeit

[3] [Atl20] S. 1
[4] [Cod20] S. 1
[5] [Jen20] S. 1
[6] [Gur20] S. 1
[7] [Xce20] S. 1
[8] [W3S20] S. 1

2.2 Zusammenwirken der verwendeten Systeme

Der Entwickler schreibt den QC der Projektdateien. Des Weiteren ist der Nutzer für die Konfiguration von CJS und TR verantwortlich. Die im Projektordner angelegten Dateien werden in das Git übertragen. Die Jenkins Maschine wird durch Auslösen eines Triggers die Dateien aus dem Git ziehen und mit diesen dann CJS ausführen. CJS führt laut Konfiguration alle gewünschten Tests aus und sendet die Testergebnisse an TR weiter. Der Nutzer kann nun diese Ergebnisse und Reports über TR einsehen. Die folgende Grafik zeigt die Interaktionen zwischen den verwendeten Systemen:

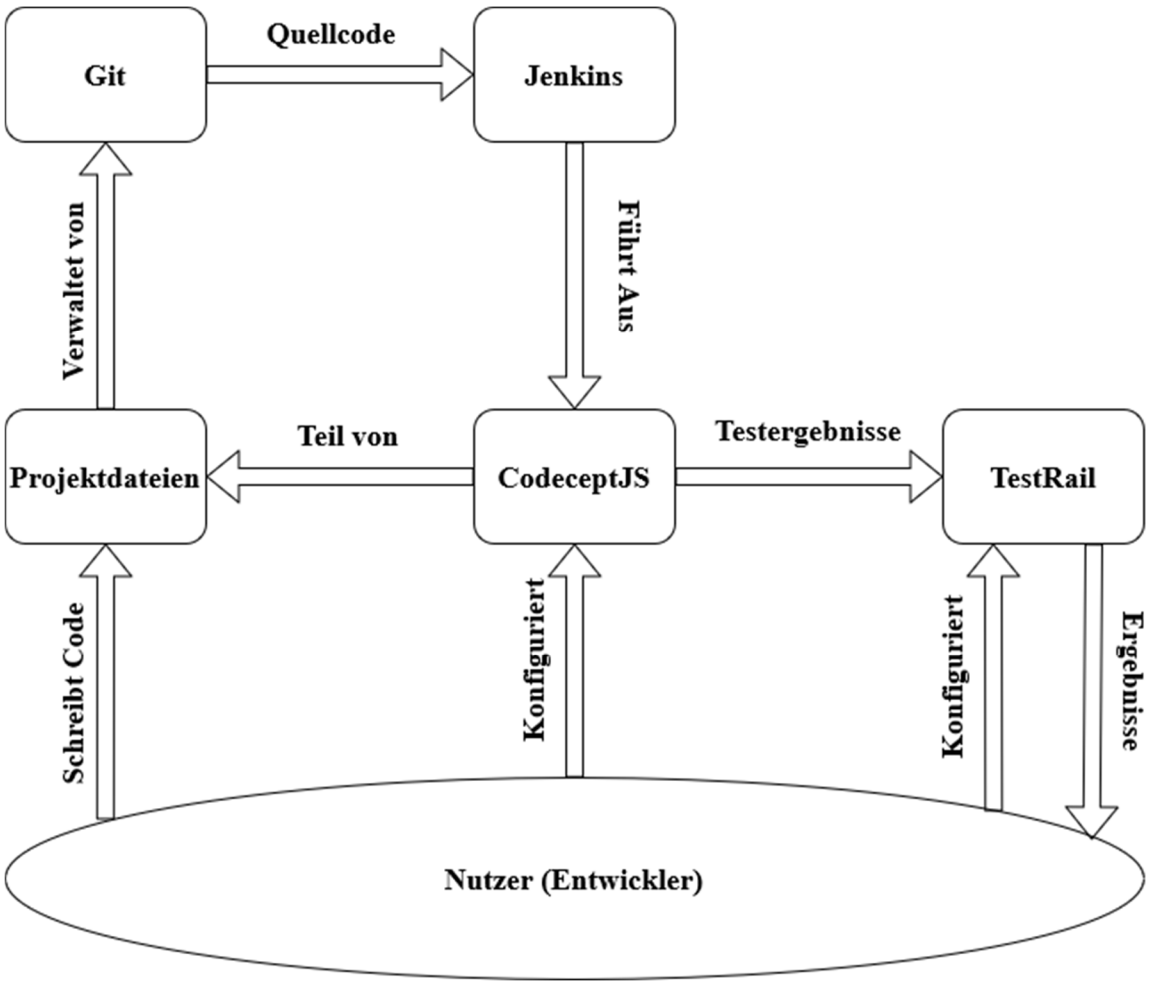

Abb. 1: Struktur der verwendeten Systeme

2.3 Bisheriger Wissensstand zur Problemstellung

Die Integration von CJS in TR wurde bereits erfolgreich für das Projekt ■■■ der DS eingeführt. Im Rahmen dieses Projektes wurden Pakete und Konfigurationsdateien erstellt, welche nun bei der Umsetzung in ■■■ teilweise wiederverwertet werden. Diese ermöglichen ein schnelles Aufsetzen des Systems und dient damit als Referenz für das Schreiben zukünftiger Tests.

2.3.1 Dateistruktur des ■■■■■CodeceptJS Projekts

Für die Übersichtlichkeit der Programmcode Dateien wurde folgende Ordnerstruktur angelegt:

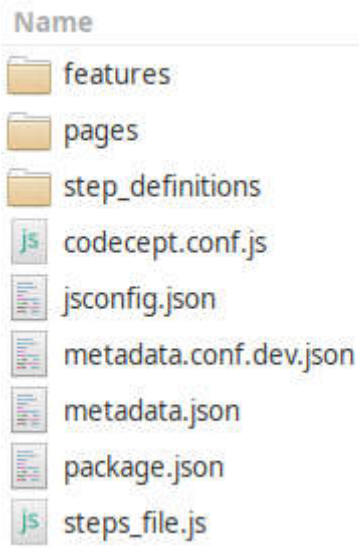

Abb. 2: Ordnerstruktur CodeceptJS von Projekt ■■■■■

Diese Ordnerstruktur unterteilt den Programmcode in Konfigurationen (root), Beschreibung der Testfälle (features), Definition der Testfälle (step_definitions) und durchzuführende Arbeitsschritte je Seite (pages).

2.3.2 Funktion der ■■■■■■ CodeceptJS Dateien

Datei / Ordner	Inhalt / Funktion
features	Enthält .feature Dateien, welche die Testfälle in Satzform beschreiben.
pages	Enthält page.js Dateien, welche die Arbeitsschritte der Tests beschreiben.
step_definitions	Enthält JavaScript Dateien, welche den Sätzen der .feature Dateien eine Funktion aus den .page Dateien zuordnen.
codecept.conf.js	Allgemeine Konfigurationsdatei von CJS
jsconfig.json	Von CJS benötigte Datei, um das Ausführen von JavaScript zu ermöglichen
metadata.conf.dev.json	Enthält Konfigurationsdaten über die verwendeten Module
metadata.json	Enthält Daten, die für die .page Dateien verwendet werden können (zum Beispiel Name und Passwort eines Testkunden)
package.json	Enthält Informationen zur Installation wichtiger Module
steps_file.js	Von CJS benötigte Datei, um das Schreiben eigener Arbeitsschritte zu ermöglichen

Tab. 2: Beschreibung der Dateistruktur von CJS

2.4 Voraussetzungen für die Integration von CodeceptJS

Für die Integration von CJS muss ein Server zur Ausführung von Tests zur Verfügung stehen. Ein- und Ausgabegeräte wie Bildschirme und Tastaturen werden benötigt, um mit dem System zu arbeiten. Eine Jenkins Maschine wird dafür angelegt, auf die der Nutzer über eine Weboberfläche zugreifen kann. Auf diesem Server muss ein Betriebssystem mit Node.js und Git installiert sein. Im Projektordner müssen sich alle auszuführenden Tests sowie die Konfigurationsdateien für CJS befinden.

2.5 Verwendung von Cucumber

Falls Kunden Testergebnisse von CJS einsehen möchten, ist es ratsam, die Arbeitsschritte eines Tests in einfachen Worten anstatt eines QC zu definieren. Damit einfache Formulierungen Programmcodegerecht geschrieben werden können, wird zusätzlich ein Tool namens Cucumber verwendet. Cucumber liest ausführbare Spezifikationen in Textform ein und überprüft, dass die Software das tut, was die Spezifikationen angeben.[9] Cucumber verwendet dabei einen Gherkin Syntax. Gherkin ist eine Reihe von Grammatikregeln, mit denen ein Testfall verständlich für Cucumber strukturiert werden kann.[10] In Anlage A1.1 wird beispielsweise Gherkin zur Definition eines Features verwendet.

[9] [Cuc20] S. 1
[10] ebenda S. 1

3 Methodik

3.1 Vorbereitung

Der Projektleiter wird nach der genauen Problem- und Zielstellung gefragt. Wenn diese festgelegt sind, wird die Verwendung des neuen Tools studiert und lokale Tests geschrieben. Wenn der Aufbau und die Funktionsweise von CJS verstanden wurden beginnt die Umsetzung für das Projekt ■■■. Dabei werden Mitarbeiter aus ■■■ zuerst nach Empfehlungen zum Vorgehen befragt sowie Ratschläge über die Dateistruktur und eventuelle Problemvorkommen eingeholt. Die Integration wird im Zeitraum vom 02.03.2020 bis zum 26.04.2020 durchgeführt.

3.2 Arbeitsschritte zur Integration von CodeceptJS

Zuerst wird ein neues Git-Repository angelegt, worin neue Konfigurations- und Testdateien von CJS abgespeichert werden können. Danach wird die Dateistruktur aus ■■■ übernommen und an das ■■■ angepasst. Um die Automatisierung der Tests zu ermöglichen, wurde eine Jenkins Maschine von der internen IT-Abteilung der DS bereitgestellt. Diese Maschine wird so konfiguriert, dass sie die Projektdaten aus dem Git übernimmt und über Konsolenbefehle alle benötigten Programme, Pakete und Module installiert. Im Anschluss daran werden in TR Testfälle definiert und die dazu passenden Testdateien im Projektordner erstellt. Zuletzt werden diese Arbeitsschritte im Detail dokumentiert, falls andere Projekte der DS CJS einführen wollen.

3.3 Arbeitsschritte zur Ermittlung des Praktischen Nutzens der Integration

Anfangs wird ein bestimmter Testfall ausgesucht. Anschließend wird der Test mit CodeceptJS geschrieben und ausgeführt. Dabei wird die Entwicklungszeit mit Hilfe einer Stoppuhr gemessen. Danach wird das Schreiben des gleichen TF wiederholt, jedoch nun unter der Verwendung des bisher genutzten XLT Tools. Zuletzt werden die Ergebnisse beider Tests verglichen und ausgewertet. Damit die Lesbarkeit verglichen werden kann, wird die Cucumber Integration betrachtet.

3.4 Dokumentation für zukünftige Projekte

Damit die oben genannten Arbeitsschritte nicht für jedes zukünftige Projekt der DS wiederholt werden müssen, wird eine Vorlage erstellt, welche die in Abb. 2 gezeigte Dateistruktur von CSJ enthält. Dabei werden Konfigurationsdateien auf die wichtigsten Einträge reduziert und benötigte Anmeldedaten (z.B. Name und Passwort für TR) entfernt. Auf einer Confluence Seite, die für alle Mitarbeiter der DS zugänglich sein wird, wird eine Installationsanleitung bereitgestellt. Diese enthält die benötigten Arbeitsschritte für die Integration von CJS in ein neues Projekt. Die Vorlage wird angehängt und es wird beschrieben von woher man benötigte fehlende Einträge beziehen kann.

4 Ergebnisse

Die Resultate der Untersuchung werden in der Reihenfolge bezüglich Zeitersparnisse und Lesbarkeit präsentiert. Alle Versuche und Messungen wurden vom Autor durchgeführt.

4.1 Zeitmessung zur Verwendung beider Tools

Aufgaben mit XLT Script Developer:

Aufgabe	Zeitaufwand	Bemerkung
Installation aller benötigten Programme	60 Minuten	Installation vieler einzelner Tools
Neuschreiben eines Tests: „Öffnen der ■■■■■■ Homepage und Cookie Einstellungen vornehmen“	5 Minuten	Erfolgt im gleichen Browserfenster
Übernehmen und Anpassen eines bisherigen Tests: „Einloggen auf der Startseite“	10 Minuten	Jede Anweisung muss einzeln überarbeitet werden.

Tab. 3: Zeitaufwand einiger Standartaufgaben mit XLT

Aufgaben mit CodeceptJS:

Aufgabe	Zeitaufwand	Bemerkung
Installation aller benötigten Programme	5 Minuten	Ausführen eines vorgefertigten Installationsskripts
Neuschreiben eines Tests: „Öffnen der ■■■■■■ Homepage und Cookie Einstellungen vornehmen“	12 Minuten	Fünf unterschiedliche Dateien müssen bearbeitet werden
Übernehmen und Anpassen eines bisherigen Tests: „Einloggen auf der Startseite“	3 Minuten	Anweisungen können kopiert werden.

Tab. 4: Zeitaufwand einiger Standartaufgaben mit CJS

Aus diesen Messungen lassen sich folgende Ergebnisse ablesen:

- Das Schreiben neuer Test mit CJS benötigt mehr Entwicklungszeit als mit XLT.
- Die Installationszeit von CJS mit allen Abhängigkeiten ist geringer als bei XLT.
- Das Übernehmen und Anpassen von TF anderer Projekte benötigt weniger Zeit als mit XLT.

4.2 Les- und Wartbarkeit des Quellcodes

CJS erlaubt durch Cucumber eigene wörtliche Formulierungen von Testschritten im Gegensatz zu XLT. Mit CJS können Elemente von zu testenden Seiten definiert werden, welche an mehreren Stellen des Programmcodes verwendet werden können (siehe Anlage A1.1). Falls sich dabei ein Element ändert, muss nur eine Programmzeile angepasst werden, um alle geschriebenen Tests der Datei funktionsfähig zu halten. XLT generiert automatisch einen Programmcode aus den befehlen der Browsereingaben. Wenn ein Element sich ändern sollte, muss für jeden Testfall einzeln eine Anpassung der Eingaben erfolgen.

4.3 Lesbarkeit der Fehlerausgabe bei Fehlschlag eines Tests

Unter der Verwendung beider Tools sind aus einem Testverlauf folgende Fehlermeldungen entstanden:

Abb. 3: Fehlerausgabe eines Tests mit CodeceptJS (links) und XLT (Ausschnitt, rechts)

CJS Fehlermeldungen beschreiben mit den gegebenen Termen aus der .feature Datei bei welchem Arbeitsschritt ein Fehler auftritt. Darauffolgend wird die exakte Fehlermeldung ausgegeben und ein Screenshot angehängt.

XLT gibt nur die exakte Fehlermeldung aus, der momentane Arbeitsschritt wird hierbei nicht erwähnt. Die Möglichkeit Screenshots bei Fehlermeldungen zu erstellen, wird zwar von XLT unterstützt, wurde jedoch nicht im ■■■ implementiert.

5 Diskussion

5.1 Validierung der Untersuchung

Durch das Messen der Zeit bis beide Tools vollständig installiert wurden sind, kann eindeutig bewiesen werden, dass das Aufsetzen eines XLT Projekts einen höheren Zeitaufwand gegenüber CJS benötigt. Die Lesbarkeit eines Programmcodes kann eindeutig durch die Verwendung von Cucumber beeinflusst werden. Der Nutzen der Testergebnisse kann von Person zu Person unterschiedlich eingeschätzt werden.

5.2 Mögliche Erklärungen für die Ergebnisse

5.2.1 Installationszeit

Das Installieren von CJS für ein neues Projekt benötigt nur das Kopieren einer Vorlage und das Ausführen weniger Konsoleneingaben. Solche Vorgänge benötigen in der Regel wenige Minuten. XLT hingegen benötigt die Installation weiterer Tools, wie zum Beispiel einen bestimmten Web Browser und eine Java Umgebung.[11] Daher ergibt sich eine Differenz bei der Installationszeit beider Tools.

5.2.2 Entwicklungszeit für neue Tests

Neue Tests mit CJS zu schreiben benötigt das Anpassen des QC mehrerer Dateien. Neue Befehle müssen über ein Textbearbeitungsprogramm eingetragen werden. XLT Tests können durch die Browsererweiterung per Mausklick schnell erzeugt werden. Dabei ist die Bearbeitung weiterer Dateien nicht erforderlich. Darüber hinaus ermöglicht XLT das Testen einzelner Befehle (z.B. das Suchen eines Textes auf einer Webseite) während der Eintragung. Daher ist die Entwicklungszeit eines neuen Tests mit CJS nicht kürzer als mit XLT.

[11] [Dom19a] S. 1

5.2.3 Übernehmen oder Anpassung eines Tests

Das Übernehmen eines CJS Tests von einem anderen Projekt benötigt nur das Kopieren der jeweiligen Testdatei. Formulierungen der Features können allgemein gehalten werden. Beispielsweise kann ein Test sich mit der Sucheingabe eines Produktes befassen. Wenn das HTML Tag der Webseiten aller Projekte den gleichen Namen besitzen, müssen im QC keine weiteren Anpassungen erfolgen. Durch die automatische Codegeneration von XLT können gesuchte Elemente im QC nicht zentral angepasst werden. Daher ergibt sich bei Änderungen ein höherer zeitlicher Aufwand bei Übernahme und Anpassung von Testdateien.

5.2.4 Aussagekraft der Testergebnisse

XLT Fehlermeldungen beschreiben welche Art Fehler aufgetreten ist, aber nicht bei welchem Arbeitsschritt. Für den Entwickler ist es leichter Fehler zu beheben, wenn verstanden wird bei welchem Arbeitsschritt diese auftreten. Screenshots helfen bei der Überprüfung von Sichtbarkeit der Elemente einer Webseite. Die Erzeugung von Screenshots ist bei XLT Tests in ■■■ nicht implementiert. Daher sind CJS Fehlermeldungen für das ■■■ aussagekräftiger als XLT Fehlermeldungen.

5.3 Neue Erkenntnisse aus dieser Arbeit

Obwohl das Schreiben neuer Tests mit CJS mehr Zeit in Anspruch nimmt als mit XLT, lassen diese sich schnell von einem Projekt auf ein anderes übertragen und anpassen.

5.4 Grenzen der Untersuchung

Die gemessenen Zeitwerte beziehen sich nur auf die Versuchsdurchführung des Autors. Die Arbeitsgeschwindigkeit kann je nach Entwickler variieren. Des Weiteren können Einarbeitungszeiten für jeden Entwickler unterschiedlich lang ausfallen. Während des Untersuchungszeitraums besteht keine Möglichkeit dies genauer zu messen, da die betroffenen Entwickler derzeitig aktiv an Projekten arbeiten und daher nicht zur Verfügung stehen.

5.5 Empfehlungen für zukünftige Untersuchungen

Da die benötigten Feldeingaben beim Anlegen der Testfälle im TestRail deckungsgleich der Anweisungen im Feature sind, besteht die Möglichkeit das Übertragen der Daten zu automatisieren. (siehe Abb. 4)

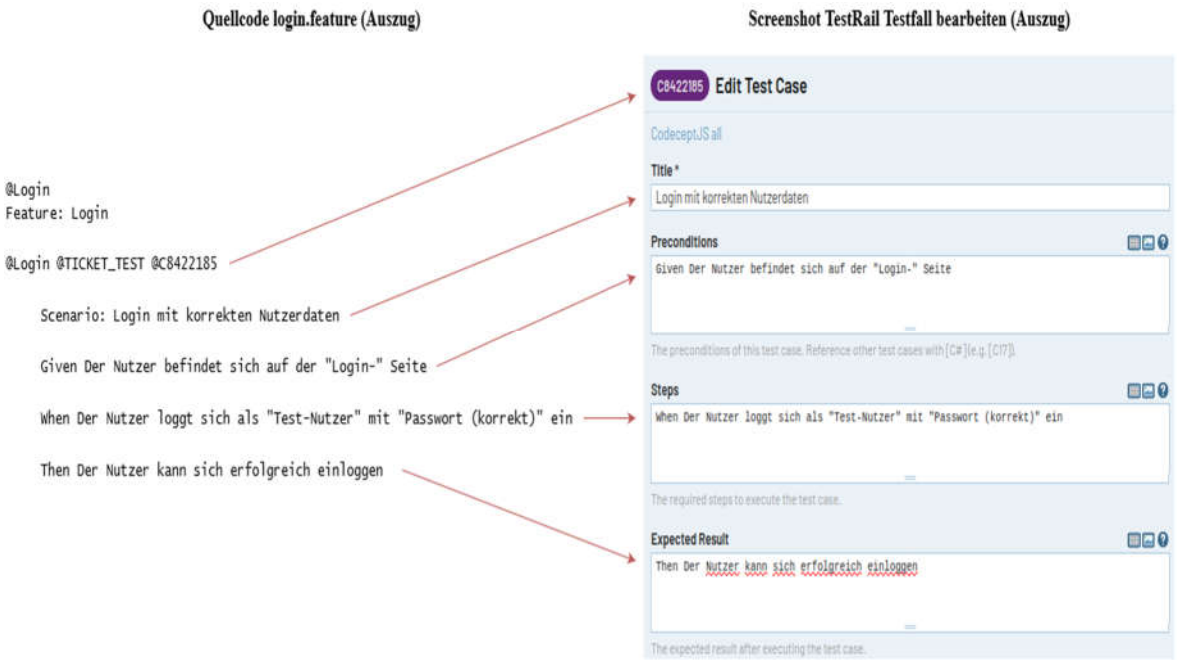

Abb. 4: Inhalt einer .feature Datei und Bearbeitung eines Testfalls in TestRail

Damit ließe sich die Entwicklungszeit für neue Tests sowie Fehlerrate bei der manuellen Eingabe noch weiter reduzieren.

6 Fazit

6.1 Zusammenfassung der Zielstellung

Das Ziel dieser Projektarbeit war es, das Tool CodeceptJS zur Automatisierung zukünftiger SFT für das Projekt ■■■ der DS in das bestehende TR zu Integrieren. Daran anknüpfend sollte ermittelt werden, ob der Arbeits- und Zeitaufwand für das Schreiben neuer SFT sich für die Entwickler positiv verändert.

6.2 Auswertung der Annahmen

Die Ergebnisse haben gezeigt, dass das Schreiben neuer Tests mit CJS keine Zeitersparnisse gegenüber XLT Script Developer mit sich bringt. Dabei hat sich jedoch herausgestellt, dass das Aufsetzen von CJS weitaus weniger Zeit in Anspruch nimmt als bei XLT. Daraus hat sich ergeben, dass CJS sich sehr schnell in andere Projekte der DS integrieren lässt. Darüber hinaus haben sich die Annahmen bestätigt, dass die Wartung und Anpassung der Testdateien durch eigenhändig geschriebenen Quellcode und leichter zu verstehende Fehlerausgaben für die Entwickler vereinfacht wird.

6.3 Abschluss

Die Integration von CJS in das ■■■ verlief erfolgreich. Das Schreiben sämtlicher gewünschten Tests sowie deren automatischer Ausführung für das ■■■ wurde ermöglicht.

Diese Arbeit hat dadurch neue Erkenntnisse über den Arbeitsaufwand bei der Übertragung und Anpassung des Quellcodes zur Integration des neuen Tools von einem Projekt (■■■) auf das andere (■■■) gezeigt. Dies ermöglicht das gleiche Verfahren auf andere Projekte der DS anzuwenden. Aus diesem Resultat könnte eine Studie anknüpfen, die sich mit dem Erstellen eines allgemeinen vollständigen Installationspakets mit fertiggeschrieben Testfällen für andere oder neue Projekte beschäftigt. Darüber hinaus sollte untersucht werden, welche finanziellen Folgen die Integration des neuen Tools mit sich bringen wird.

Literaturverzeichnis

[Atl20] Atlassian.: „What is Git: become a pro at Git with this guide | Atlassian Git Tutorial", o. O., 2020.
https://www.atlassian.com/git/tutorials/what-is-git
Abruf: 09.03.2020

[Cod20] Codecept.io.: „Getting Started | CodeceptJS", o. O., 2020.
https://codecept.io/basics/
Abruf: 09.03.2020

[Cuc20] Cucumber.io.: „Introduction - Cucumber Documentation", o. O., 2020.
https://cucumber.io/docs/guides/overview/
Abruf: 19.03.2020

[Dom19a] ■■■.: „Quick Guide - Installation benötigter Tools - Team Quality Assurance - Confluence dotSource GmbH", Jena, 2019.
https:// ■■■
Abruf: 03.03.2020

[Dom19b] ■■■.: „TQA-XLT-dotSouceScriptDeveloperUpdate-100320-1451-140", Jena, 2019.
https:// ■■■
Abruf: 13.03.2020

[Fow00] Fowler, Martin.: „Continuous Integration", o. O., 2000.
https://martinfowler.com/articles/continuousIntegration.html
Abruf: 19.03.2020

[Gur20] Gurock.: „Test Case Management & Test Management Software Tool - TestRail", o. O., 2020.
https://www.gurock.com/testrail
Abruf: 09.03.2020

[Jen20] Jenkins.: „Jenkins User Documentation", o. O., 2020.
https://jenkins.io/doc/
Abruf: 09.03.2020

[W3S20] W3Schools.: „Node.js Tutorial", o. O., 2020.
https://www.w3schools.com/nodejs/default.asp
Abruf: 12.03.2020

[Xce20] Xceptance Software Technologies GmbH.: „Script Developer", Cambridge, 2020.
https://lab.xceptance.de/releases/xlt/latest/user-manual/03-scriptdeveloper.html
Abruf: 09.03.2020

Anlagenverzeichnis

A1 Programmlisting

A1.1 Login.feature

```
@Login
Feature: Login

       @Login @TICKET_TEST @C8422185
       Scenario: Login mit korrekten Nutzerdaten
              Given Der Nutzer befindet sich auf der "Login-" Seite
              When Der Nutzer loggt sich als "Test-Nutzer" mit "Passwort (korrekt)" ein
              Then Der Nutzer kann sich erfolgreich einloggen
```

A1.2 login_steps.js

```
const { I, inbucket, loginPage, generalPage, data, data_conf } = inject();

Given('Der Nutzer befindet sich auf der {string} Seite', (param1) => {
  generalPage.callPage(param1);
});

When('Der Nutzer loggt sich als {string} mit {string} ein', (param1, param2) => {
  loginPage.login(param1, param2);
});

Then('Der Nutzer kann sich erfolgreich einloggen', () => {
  loginPage.loginSuccess();
});
```

A1.3 login.page.js

```
const I = actor();
const { data, data_conf } = inject();

module.exports = {
    locators: {
        emailLogin: '//*[@id="login_authenticate_username"]',
        passwordLogin: '//*[@id="login_authenticate_password"]',
        primaryButton: '//*[@data-id="btn-login"]',
        myAccountLinkTextLogin: locate('.account-info-box')
    },
    login(email, password) {
        // fill login form
        I.seeElement(this.locators.emailLogin);
        I.seeElement(this.locators.passwordLogin);
        I.fillField(this.locators.emailLogin, data[email].email);
        I.fillField(this.locators.passwordLogin, data[email][password]);
        // click login
        I.seeElement(this.locators.primaryButton);
        I.click(this.locators.primaryButton);
    },
    loginSuccess() {
        // check login state
        I.seeElement(this.locators.myAccountLinkTextLogin);
    }
};
```

A1.4 Login_Test.java

```
package gen;
import org.junit.Test;
import java.io.File;
import org.openqa.selenium.WebDriver;
import java.net.URL;
import org.junit.After;
import
com.dotsource.xlt.engine.scripting.AbstractWebDriverScriptTestCaseWithScreenOnFail;
import com.dotsource.xlt.chromedriver.AdjustableChromeDriver;

/**
 * TODO: Add class description
 */
public class Login_Test extends AbstractWebDriverScriptTestCaseWithScreenOnFail
{

    /**
     * Constructor.
     */
    public Login_Test()
    {
        super(new AdjustableChromeDriver(), "");
    }

    /**
     * Executes the test.
     *
     * @throws Throwable if anything went wrong
     */
    public void test() throws Throwable
    {
        deleteAllVisibleCookies();
        open("[redacted]");
        clickAndWait("//*[@id='submitAll']");
        clickAndWait("//*[@data-id='RoleB2BBanner']//*[@class='btn btn-primary']");
        click("//*[@id='link-login']");
        waitForElementPresent("//*[@id='account-login-popup']");
        type("//*[@id='j_usernamePopup']", "[redacted]");
        type("//*[@id='j_passwordPopup']", "[redacted]");
        clickAndWait("//*[@id='account-login-popup']//*[@class = 'btn btn-primary btn-
block']");
        assertVisible("//*[@class='alert positive']");
    }


    /**
     * Clean up.
     */
    @After
    public void quitDriver()
    {
        // Shutdown WebDriver.
        getWebDriver().quit();
    }
}
```

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Projektarbeit/Studienarbeit/Bachelorarbeit mit dem Thema:

Integration eines neuen Tools zur Automatisierung von Storefront Tests für ein Projekt der dotSource GmbH

ohne fremde Hilfe angefertigt habe,

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe und

3. dass ich meine Projektarbeit/Studienarbeit/Bachelorarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird

_____ _____

Ort, Datum Unterschrift