MASTERARBEIT

TESTKONZEPT ZUR EINFÜHRUNG VON END-TO-END TESTS FÜR SAP COMMERCE IM KONTEXT VON TEST-DRIVEN DEVELOPMENT

ausgeführt am

Studiengang
Informationstechnologien und Wirtschaftsinformatik

Von: Alexander Hödl
Personenkennzeichen: 1910320005

Graz, am 20. November 2020 .....

Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Alexander Hödl

DANKSAGUNG

Hiermit möchte ich mich bei all denjenigen bedanken, die mich während der Erstellung dieser Masterarbeit unterstützt und begleitet haben.

Besonderer Dank gilt Herrn DI Dr. Thomas Puchleitner für die Betreuung und Beurteilung der Arbeit sowie der konstruktiven Kritik und des Feedbacks im Zuge der Erstellung. Weiters bedanke ich mich bei Herrn Stephan Zacharias, MSc für die Unterstützung und Betreuung während der Umsetzung des praktischen Teils dieser Arbeit bei der B4B Solutions GmbH.

Abschließend möchte ich mich bei meinen Eltern und meiner Familie für die Unterstützung während des Studiums bedanken, die mir durch den starken Rückhalt mein Studium ermöglicht haben. Besonders möchte ich mich bei meiner Frau Marijana für das Verständnis und die Unterstützung in dieser intensiven Studienzeit bedanken.

Graz, am 4. Dezember 2020 Alexander Hödl

KURZFASSUNG

Testen von Softwareapplikationen gehört zu einem wesentlichen Bestandteil des Qualitätssicherungsprozesses und muss im agilen Ansatz der Softwareentwicklung laufend durchgeführt werden. Mithilfe des Konzepts von Test Driven Development kann die Testfall- und Anforderungsspezifikation verschmolzen werden, da der Testfall vor der Entwicklungsphase definiert wird und die Entwicklung erst mit positiver Absolvierung des Testfalls abgeschlossen ist. Dadurch können Zeit und Kosten minimiert werden.

Ziel dieser Arbeit ist die Entwicklung einer Anwendung, mit deren Hilfe End-to-End Tests für Webanwendungen vor der Implementierung erstellt und automatisiert ausgeführt werden können. Das entwickelte Softwarekonzept soll den Ansatz von *Test Driven Development* ermöglichen und die Testfallerstellung in textueller Form ohne technischen Hintergrundwissen ermöglichen. Weiters soll das Testkonzept prototypisch realisiert und die erzielten Erfahrungen aus einem Referenzprojekt dokumentiert werden.

Im ersten Schritt erhält der Leser Einblick über die Konzepte von *Test Driven Development* und *Behavior Test Driven*. Weiters werden die Möglichkeiten der Testautomatisierung für Webanwendungen erläutert und die gängigen verfügbaren Frameworks analysiert und bewertet. Anschließend wird ein Konzept für die Testfallerstellung und Testausführung entworfen, welches in einem Prototypen realisiert wird. Die resultierenden Erfahrungen aus dem Einsatz des Prototypen in einem Referenzprojekt sollen dazu dienen, um auf den Nutzen des Testkonzepts zu schließen.

Das Ergebnis der Arbeit zeigt, dass das entwickelte Konzept eine Testfallerstellung auch ohne technischen Hintergrundwissen ermöglicht und so von Personen mit Prozesswissen erfolgen kann. Der Ansatz von *Test Driven Development* kann mit dem Prototypen gezeigt werden. Im eingesetzten Projekt kann anhand der positiv ausgeführten Testfälle mit fortlaufender Zeit ein positiver Trend und Beitrag zur Qualitätssteigerung identifiziert werden.

Die vorliegende Arbeit lässt schlussfolgern, dass der Prototyp für die Erstellung von automatisierten Testfällen nach dem Ansatz *Test Driven Development* eingesetzt werden kann. Wenngleich im eingesetzten Referenzprojekt nur wenige Testfälle erstellt wurden, kann von einem positiven Trend im Zuge des Einsatz gesprochen werden.

ABSTRACT

Software testing is essential to the quality assurance process and the agile approach demands it be performed continuously. Test-Driven Development helps merge the test case and the requirements specification, since the test case is defined and completed successfully before the development phase. This minimises the time and cost burden.

This thesis develops an application to create and automate execution of end-to-end tests for web applications. The software enables a Test-Driven Development approach and allows test cases to be created using natural language without technical background knowledge. A prototype test concept is implemented as a reference project.

The thesis gives an insight into Test-Driven Development and Behavior-Test Driven. Furthermore, test automation for web applications are explained and standard frameworks analyzed and evaluated. Additionally, a prototype application was developed to show the test case creation for automated tests in textual form without technical background knowledge.

Results show a positive trend and contribution to quality improvement based on the positively executed test cases. The prototype can create automated test cases according to the Test-Driven Development approach. The developed web application can be used to generate test cases based on a company's process flows during a sprint planning phase.

INHALTSVERZEICHNIS

1 Einleitung und Problemstellung

In heutigen Softwareentwicklungsprozessen gehört die Tätigkeit des Prüfens und Testens von Softwareapplikationen zu einem wesentlichen Bestandteil des Qualitätssicherungsprozesses. Durch den agilen Softwareansatz der kontinuierlichen Integration (Continious Integration) muss sichergestellt werden, dass bereits vorhandene Funktionalität durch neue Erweiterungen oder Anpassungen nicht ungewollt zerstört oder verloren geht. Dieser Teilbereich der Softwareentwicklung bedarf nach Spillner (Spillner & Linz, 2012) einer professionellen Ausbildung und zählt zu einer immer wichtigeren Aufgabe in der Erstellung von Applikationen.

Dieser Aspekt wird in vielen Softwareprojekten jedoch aufgrund von Kosten- oder Zeitfaktoren meist vernachlässigt, wodurch es zu Qualitätsverlusten und damit verbunden zu Vertrauensverlusten des Kunden und Anwenders in die Applikation kommen kann. Aufgrund der Zeit- und Kostenintensität werden Softwaretests, speziell in kleineren Unternehmen (KMUs oder Start-Ups) oder bei Projekten mit engem Budget, vernachlässigt. Während bei größeren Unternehmen oder langfristigen Projekten meist eigene Testressourcen (eigene Bereiche oder Abteilungen) zur Verfügung stehen, werden Softwaretests in kleineren Projekten meist lediglich vom Entwicklungsteam selbst durchgeführt. Dadurch werden Funktionalitäten vom Hersteller (Entwickler) selbst geprüft, wodurch diese meist nur im Kontext der eigenen Lösungsidee geprüft werden. Dadurch können Fehler in den eigenen Funktionalitäten gefunden, aber im Zusammenspiel mit anderen Komponenten unentdeckt bleiben. Weiters kann durch dieses Vorgehen das V-Modell nach Böhm ((B. W. Boehm, 1984)) nicht umgesetzt werden, da Test- und Entwicklungstätigkeit keine korrespondierenden Aktivitäten einnehmen können (Spillner & Linz, 2012).

Speziell der Aspekt der fehlenden Integrations- und Systemtests verursacht Unsicherheit in Bezug auf Interaktion der einzelnen Softwarekomponenten miteinander. Heutige Softwaresysteme agieren meist nicht als eigenständige Monolithen, sondern kommunizieren mit anderen Softwarekomponenten, um ein gewünschtes Gesamtsystem abbilden zu können. Beispielsweise interagiert ein Webshop mit einem dahinterliegenden ERP System, um eine Preisfindung durchzuführen oder die Bestellung weiterführend zu verarbeiten. Während die Funktionalität einzelner Komponenten innerhalb eines einzelnen Systems in sich selbst mithilfe von Unit-Tests und Integrationstests durch den

Entwickler noch qualitätssichernd geprüft werden können, ist für die Überprüfung von Prozessabläufen und Schnittstellen ein ganzheitliches Testkonzept (End-to-End Tests) erforderlich. Dieses Testkonzept muss entsprechende Prozessabläufe widerspiegeln und sämtliche systemübergreifende Interaktionen - von der Oberflächeneingabe bis zur Verarbeitung im Backend - abbilden. Daher bedarf es für die Konzeption solcher Tests ein genaues System- und Prozessverständnis sowie Wissen über die Schnittstellenkommunikation zwischen den Systemen. Aufgrund dieses fehlenden System- und Prozessverständnisses und dem dazu benötigten Aufwand, erfolgt diese Prüfung meist durch den Endanwender oder Kunden, wodurch Fehler in den Schnittstellen oder Prozessabläufen erst bei Kundenkontakt identifiziert werden können.

Eine weitere Herausforderung liegt in der Sicherstellung der Testüberdeckung einzelner Funktionalitäten mithilfe von Unit-Tests durch die jeweiligen Softwareentwickler. Werden Tests zur Verifikation einer entwickelnden Funktionalität erst nach Fertigstellung dieser geplant, werden diese meist aufgrund von Zeit- und Kostenfaktoren nicht umgesetzt. Dadurch kann bei ständiger Weiterentwicklung eines Systems eine Testüberdeckung sämtlicher Funktionalitäten nur mehr mit erheblichem Aufwand sichergestellt werden. Dadurch kann es in weitere Folge zu Wartungsproblematiken und unerwarteten Fehlverhalten des Systems kommen. Mithilfe des Ansatzes Test-Driven Development (TDD) kann diese Problematik eingedämmt werden, da Testfälle vor der Realisierung der Funktionalität erstellt werden müssen (George & Williams, 2004).

1.1 Zielsetzung und Forschungsfragen

Das Ziel dieser Masterarbeit ist die Entwicklung eines Testkonzepts für die Erstellung von automatisierten Systemtests (End-to-End) auf Basis des Test-Driven Developmentansatzes für eine SAP E-Commerce Lösung (SAP Commerce). Theoretische Ansätze des Konzepts sollen jedoch systemunabhängig entworfen werden, sodass eine Übernahme des Konzepts für andere Applikationen möglich ist und nur die Implementierung des Konzepts angepasst werden muss.

Das Testkonzept soll mit möglichst geringem Budgeteinsatz realisierbar und in der Wartung des Systems einfach sein. Dadurch soll es einen Einsatz in Projekten mit engen Budgets ermöglichen oder für Projekte von Start-ups oder KMUs eine Basis für automatisiertes Testen schaffen. Daher soll bei der Auswahl von externen Applikationen und Schnittstellen auf Open-Source Varianten gesetzt werden.

Als End-to-End Tests werden in diesem Kontext Systemtests bezeichnet, die die Funktionalität eines Systems aus Anwendersicht prüfen. Die Tests betrachten das System als Blackbox und stellen die Interaktionsschritte des Endnutzers mit den einzelnen Komponenten nach, wodurch Eingaben möglichst oberflächennah (GUI-Elemente) erfasst werden. Auf gleicher Ebene (GUI) werden die erwarteten Ergebnisse entsprechend geprüft. Das Konzept soll Testfälle nach dem Ansatz von Behavior Driven Testing (BDT) verarbeiten können und dadurch Prozessabläufe in allgemeiner Form realisieren. Diese erstellten End-to-End Tests sollen in weitere Folge als Regressionstests für bereits bestehende Funktionalitäten dienen, um nach einem Integrationszyklus den Zustand des Systems grundsätzlich bewerten zu können. BDT ist ein Ansatz, der Testfälle mithilfe einer definierten (allgemeinen) Textform beschreibt und dadurch eine Automatisierung dieser Testfälle ermöglicht (Kerthyayana Manuaba, 2019).

Das entwickelte Testkonzept soll Unternehmen ermöglichen, Prozessabläufe automatisiert und kostengünstig intern (vor Kundenkontakt) zu prüfen und dadurch eine höhere Qualität der entwickelten Softwarekomponenten zum Übergabezeitpunkt an den Kunden ermöglichen. Weiters soll es aufzeigen, welche Aspekte bei der Realisierung eines Testframeworks für Applikationen berücksichtigt werden müssen, und wie automatisierte Tests in allgemeiner Form für die Wiederverwendbarkeit erstellt werden können (BDT). Das entwickelte Konzept soll in Zusammenarbeit mit dem Unternehmen B4B Solutions GmbH als Prototyp für die Realisierung einer SAP E-Commerce Lösung (SAP Commerce) umgesetzt und die daraus resultierenden Ergebnisse entsprechend dokumentiert werden.

Basierend auf der Einleitung und Zielsetzung ergibt sich folgende Forschungsfrage:

- Wie können Prozessabläufe (Systemtests) in einem E-Commerce System (SAP Commerce) unter Einbeziehung der Schnittstelleninteraktion zu anderen Systemen automatisiert getestet werden?

Weiters lassen sich im Zuge der Arbeit noch folgende weitere Fragestellungen beantworten:

- Welche technischen Voraussetzungen werden benötigt, um automatisierte Regressionstests für Webapplikationen (am Beispiel von SAP Commerce) zu erstellen und auszuführen?
- Wie kann der Ansatz von Test Driven Development in Kombination mit Testautomatisierung in agilen Softwareentwicklungsprojekten eingesetzt werden?
- Welche Vor- und Nachteile ergeben sich durch die Implementierung eines solchen

Testframeworks?

- Wie kann das Testframework mit bereits eingesetzten Softwareentwicklungstools (z. B. Jenkins, Jira, ...) verknüpft und integriert werden?

1.2 Methodisches Vorgehen

Diese Masterarbeit fokussiert im praktischen Teil auf die konzeptionelle Erstellung eines Testkonzepts für automatisierte End-to-End Tests basierend auf einem Test-Driven Development Ansatz. Weiters wird dieses Konzept im Zuge eines Referenzprojektes in einer Webanwendung (E-Commerce Lösung auf Basis von SAP Commerce) der Firma B4B Solutions GmbH prototypisch realisiert und die daraus resultierenden Ergebnisse dokumentiert. Diese Dokumentation dient zur Evaluierung und Validierung des entwickelten Testkonzepts.

Mithilfe von Literaturrecherche wird im theoretischen Teil der Arbeit ein Konzept für die Entwicklung eines allgemeinen Testframeworks zur Erstellung und Ausführung von automatisierten End-to-End Tests erstellt. Dazu werden die bereits vorhandenen Konzepte im Bereich von Softwaretests erläutert und deren Einfluss in Bezug auf die Softwarequalität dargelegt sowie nach deren Vor- und Nachteile bewertet.

Auf Basis dieser Informationen wird ein allgemeines Konzept für ein Testframework entwickelt, welches möglichst Kosten- und Ressourcen-günstig umsetzbar ist, um in kleineren und mittleren Unternehmen Einsatzmöglichkeiten zu finden. Das Konzept soll bei Nutzung von externen Applikationen und Schnittstellen deren kostenlose Nutzung (Open Source) berücksichtigen. Weiters soll das Testkonzept ein möglichst vollumfängliches Qualitätssicherungsframework abbilden. Vollumfänglich bedeutet, dass das Konzept beschreibt, wie automatisierte Tests einfach und schnell entworfen und erstellt werden können, aber auch, wie diese in eine bestehende Softwareentwicklungslandschaft integriert werden können, um eine automatische Ausführung zu ermöglichen (beispielsweise nach einer Softwareänderung). Die resultierenden Testergebnisse sollen in entsprechender Form für Testberichte in einer allgemeinen API bereitgestellt werden.

Im Weiteren wird im praktischen Teil das entwickelte Konzept anhand eines Referenzprojektes der Firma B4b Solutions GmbH eingesetzt, um die theoretischen Ansätze in einer praktischen Umgebung zu validieren. Die daraus resultierenden Ergebnisse werden mithilfe von Kennzahlen (beispielsweise *Anzahl gefundener Fehler*, *Verhältnis erfolgreicher vs. durchgeführter Testläufe*, ...) bewertet. Die zu ermittelnden Kennzahlen werden im

Zuge der Konzeptentwicklung (noch vor der praktischen Anwendung im Referenzprojekt) definiert, um eine objektive Bewertung des Ergebnisses zu ermöglichen.

1.3 Aufbau

In der vorliegenden Masterarbeit werden zu Beginn die Problemstellung, Zielsetzung und Forschungsfragen dargelegt sowie das methodische Vorgehen und der grundsätzliche Aufbau des Dokuments beschrieben. Die Arbeit selbst ist in einen theoretischen und praktischen Bereich gegliedert.

Der theoretische Teil erläutert die Gründe und Testmöglichkeiten in einem Softwareentwicklungsprojekt sowie die Vor- und Nachteile des Ansatzes ”Test Driven Development”. Im Weiteren werden die Möglichkeiten einer Testautomatisierung beleuchtet und in Kontext mit dem **Test-Driven-Development** Ansatz (TDD) gebracht sowie die Testtechnik **Behavior Driven Testing** erläutert. Im Zuge einer Analyse von vorhandenen Frameworks für die Testautomatisierung werden die jeweiligen Vor- und Nachteile bewertet. Abschließend werden in Kapitel 3 Testframeworks vorhandene Testframeworks auf deren Möglichkeit zur Erstellung von neuen automatisierten Tests geprüft, um die Möglichkeiten für eine einfache Beschreibungsart von automatisierten Tests nach Behavior Driven Testing (BDT) zu ermöglichen.

Der praktische Teil beschreibt die Entwicklung des ganzheitlichen Konzepts (siehe 4 Konzeptentwicklung) und dessen Funktionsumfang und Aufbau. Hier werden die Grenzen zu anderen Frameworks erläutert sowie Unterschiede zu anderen Testframeworks dokumentiert. Im Abschnitt 4.4 Technische Umsetzung wird die technische Umsetzung des Frameworks (Integration in bestehendes Softwareprojekt) erläutert und beschrieben. Im Weiteren wird die Funktionsweise des Konzepts näher beschrieben sowie die Gründe für die gewählten jeweiligen Softwarekomponenten beleuchtet. In Kapitel 5 Konzeptvalidierung wird der Einsatz des Konzeptes beim Referenzprojekt der Firma B4B Solutions GmbH dokumentiert und die daraus resultierenden Erfahrungen in Bezug auf Implementierung, Benutzbarkeit und Testergebnisse dokumentiert. Um eine nachvollziehbare Bewertung zu ermöglichen, werden vor der Implementierungsphase Kennzahlen definiert, welche während der Testausführung erhoben werden. Diese erhobenen Kennzahlen sollen eine Aussage über den Nutzen des Systems erlauben.

Im abschließenden Kapitel 6 Schlussfolgerung werden die gesammelten Ergebnisse aus der Implementierung sowie die erzielten Kennzahlen kritisch hinterfragt und die

Erfahrungen des Konzepts in Bezug auf Einsatzmöglichkeiten beschrieben. Weiters werden die erhobenen Kennzahlen (Nutzen) dem dazu benötigen Aufwand (Kosten) gegenübergestellt, um das Testkonzept einer groben wirtschaftlichen Bewertung zu unterziehen.

2 Testen im Softwareentwicklungsprozess

Im Ansatz der agilen Softwareentwicklung gehört Testen zu einem elementaren Bestandteil des Softwareentwicklungsprozesses. Folgendes Kapitel erläutert die Gründe für die Wichtigkeit von Softwaretests und die unterschiedlichen Testarten in den jeweilig unterschiedlichen Systemebenen sowie deren Unterschiede zueinander. Weiters werden die Möglichkeiten der Testautomatisierung in den jeweiligen Stufen des V-Modells nach Böhm (B. W. Boehm, 1984) dargestellt.

2.1 Gründe für Softwaretests und das V-Modell nach Böhm

Wie im V-Modell nach Böhm beschrieben, agieren Testtätigkeiten und Entwicklung auf mehreren Ebenen als in Verbindung stehende Komponenten und sollen als gleichberechtigte Tätigkeiten gesehen werden. Das V-Modell (siehe Abbildung 2.1) zeigt diese beiden Tätigkeitsstränge in V-Form sowie die jeweiligen korrespondierenden Ebenen in einem Softwareentwicklungsprozess. Während der Weg zur Realisierung basierend auf der Anforderungsspezifikation bis zur konkreten Umsetzung (Programmierung) in der Spezifikation immer feiner granuliert definiert wird, folgt der ganzheitliche Testprozess einem Bottom-Up Prinzip. Hier beginnt die Qualitätssicherung in der Prüfung einzelner Methoden oder Funktionen im Detail durch Komponententests (Unit-Tests) und mündet in einem finalen Abnahmetest als durchgängige Prozessprüfung durch den Kunden (Spillner & Linz, 2012).

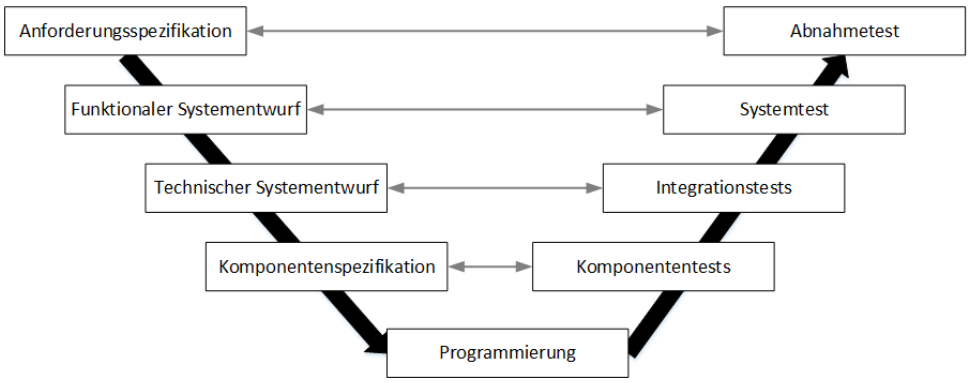

Abbildung 2.1: Das V-Modell nach Böhm. Grafik basiert auf (Spillner & Linz, 2012)

Nach Spillner beschreibt das V-Modell weiters, dass die jeweiligen korrespondierenden Ebenen (Spezifikation vs. Tests) der Validierung dienen, während die jeweils darauf folgenden Tätigkeiten die vorhergehende Spezifikation verifizieren (Spillner & Linz, 2012). Jede Teststufe prüft also, ob die erzielten Resultate den entsprechenden Anforderungen auf der jeweiligen Ebene entsprechen, während die jeweilige Entwicklungsphase selbst nur die Vollständigkeit der Umsetzung der definierten Anforderung in der vorhergehenden Phase prüft.

Die Testphasen dienen also dazu, eine Applikation auf die Erfüllung seiner Spezifikation in der jeweiligen Phase zu prüfen und sollen sicherstellen, dass Abweichungen frühzeitig erkannt werden. Im Weiteren zielen die einzelnen Testphasen auf einen unterschiedlichen Testumfang ab. Während Komponententests die Anforderungen auf einer detaillierten Ebene (Methoden oder Funktionen) prüfen, agieren Systemtests auf Prozess- oder Systemebene (Spillner & Linz, 2012).

Durch die unterschiedlichen Ziele in den jeweiligen Testebenen können unterschiedliche Abweichungen identifiziert werden: Während in der Phase der Komponententests auf einzelne Softwarebausteine (Klassen, Methoden) fokussiert wird, um Fehler in der Implementierung zu identifizieren, wird bei Integrationstests der Fokus auf die Schnittstellen der einzelnen Bausteine gelegt. Somit sollen in dieser Phase Fehler zwischen einzelnen Bausteinen identifiziert und eine korrekte Kommunikation zwischen diesen sichergestellt werden. Die Phase des Systemtests betrachtet das gesamte Systemverhalten und prüft das System inklusive der einwirkenden Umwelteinflüsse. Dadurch können Abweichungen im Systemverhalten und in den Prozessabläufen erkannt werden (Spillner & Linz, 2012).

2.2 Möglichkeiten der Testautomatisierung

Die Komplexität in Bezug auf die Automatisierung von Tests steigt mit Zunahme der Testebene nach dem V-Modell nach Böhm. Während für die Erstellung von Komponententests (Unit-Tests) bereits weit verbreitete Frameworks für diverse Programmiersprachen zur Verfügung stehen, können Systemtests meist nur mit erheblichen Aufwand und Kosten automatisiert getestet werden.

Meist wird die Funktionalitätsprüfung einzelner Softwarebausteine (Methoden, Klasse) direkt durch den Entwickler mithilfe von Unit-Tests durchgeführt. Für die Erstellung solcher Tests stehen bereits verschiedenste etablierte Frameworks für die meisten Programmier-

sprachen zur Verfügung. Beispielsweise existieren für die Programmiersprache *JAVA* die Unit-Test Frameworks **JUnit** oder **TestNG** (Karanam, 2019). Die Programmiersprache *Phyton* beinhaltet bereits im Standard eine Bibliothek zur Erstellung von Unit-Tests (Python Software Foundation, 13.06.2020). Um Interaktionen zu anderen Bausteinen zu vermeiden, werden auf der Ebene der Komponententests sogenannte Mocks eingesetzt, welche die Schnittstellen zu anderen Funktionsbausteinen (Methoden) simulieren, um so eine isolierte Testumgebung zu schaffen. Dieser Mockingansatz kann auch für die Realisierung des Ansatzes Test Driven Development verwendet werden, um noch nicht vorhandene Systemkomponenten (Funktionen) zu simulieren (Karlesky, Williams, Bereza & Fletcher, April 2017).

Auf Ebene der Integrations- und Systemtests steigt die Komplexität für die Erstellung, da diese Tests eine Interaktion von einzelnen Komponenten miteinander voraussetzen. In der Ebene der Integrationstests wird geprüft, ob das *Zusammenspiel aller Einzelteile miteinander richtig funktioniert* ((Spillner & Linz, 2012)). Für die Automatisierung von Integrationstests kann auf die Testobjekte aus den Komponententests zurückgegriffen werden. Die im Komponententest implementierten Mocks zur isolierten Prüfung werden durch die konkreten Softwarekomponenten ersetzt, wodurch eine Interaktion zwischen den isolierten Komponenten hergestellt wird. Dadurch interagieren die einzelnen Komponenten nicht mehr isoliert, sondern agieren als geschlossenes System. Um den Aufwand für diese Vorgehensweise möglichst gering zu halten, wird jedoch eine gut strukturierte Komponententestebene vorausgesetzt. Weiters muss der Datentransfer zwischen den Komponenten überwacht und geprüft werden, was einen zusätzlichen Aufwand für die Automatisierung von Integrationstests bedeutet (Spillner & Linz, 2012).

Die Ebene der Systemtests fokussiert auf die Erfüllung der Anforderungen an das System aus Nutzer- und Prozesssicht. Während die vorangegangen Ebenen auf die technische Umsetzung und deren korrekte Funktionsweise geprüft haben, zielen Tests auf Ebene der Systemtests auf die Nutzung der Applikation aus Prozesssicht ab und prüfen die Erfüllung des logischen Ablaufs des Prozesses. Diese Ebene fokussiert auf das Zusammenspiel des Gesamtsystems und der Interaktion mit deren Umwelt (Spillner & Linz, 2012). Im Gegensatz zum Abnahmetest, entspricht die Umwelt im Systemtest zwar immer noch einer Testumgebung, diese sollte aber dem Produktivumfeld bereits weitestgehend entsprechen (beispielsweise ähnliche Hardware, Netzwerkumgebung, ...). Die Tests selbst betrachten das Gesamtsystem (wie im Abnahmetest) als Blackbox und bewerten lediglich die produzierten Ergebnisse auf Basis der vorhergehenden Inputs. Die Interaktion (Eingabe sowie Prüfung) mit dem System während des Tests erfolgt am selben Level wie der spätere Nutzer (Spillner & Linz, 2012). Damit wird bei

Systemtests nicht der Prozess selbst, sondern nur der produzierte Output basierend auf den gegebenen Input geprüft.

Für die Automatisierung von Systemtests bedeutet dies, dass die Interaktion auf möglichst naher Benutzerebene erfolgt und die Testtreiber das Verhalten des Anwenders simulieren. Beispielsweise simuliert ein Testtreiber einen Klick auf einen Button oder eine Testeingabe in einem Eingabefeld und prüft das erwartete Ergebnis ebenfalls an der Oberfläche. Das beschriebene Konzept in Kapitel 4 erläutert eine automatisierte Testmöglichkeit von Systemtests sowie die Integration in unterstützenden Softwareentwicklungsapplikationen.

Welche Testframeworks für die Automatisierung in den jeweiligen Stufen am Markt bereits existieren sowie die dahinterliegenden Konzepte werden in Kapitel 3 Testframeworks beschrieben.

2.3 Test Driven Development und Testautomatisierung

Wird ein Entwicklungsprozess nach dem Wasserfallprinzip realisiert, erfolgt die Prüfung der realisierten Lösung (System- bzw. Abnahmetest) nach Abschluss der Implementierungsphase (Fritzsche & Keil, Juni 2007). Wie in Abbildung 2.2 ersichtlich, besteht das Wasserfallmodell aus aufeinanderfolgende Phasen, wobei der Übergang der Phasen erst nach Abschluss erfolgt und ein nachträgliches Wechseln auf die vorherige Phase nicht möglich ist (Sommerville, 2016).

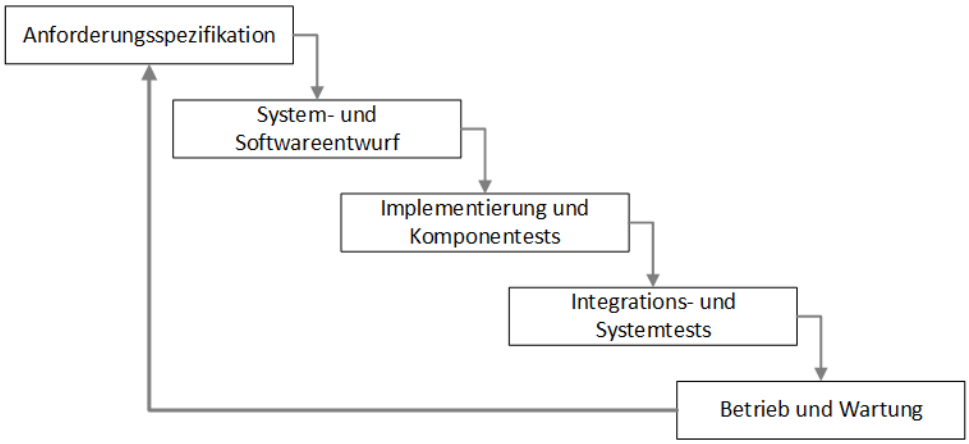

Abbildung 2.2: Das Wasserfallmodell. Grafik basiert auf (Sommerville, 2016)

Diese Herangehensweise erschwert jedoch die Integration von Kundenfeedback und Anpassungen während der Entwicklungsphase, welche mithilfe des Ansatzes der konti-

nuierlichen Integration (Continuous Integration) und dem damit verbunden agilen Ansatz der Softwareentwicklung (beispielsweise SCRUM) erleichtert wird. Durch die Einbindung des Kunden in den Entwicklungsprozess kann eine schnellere Rückmeldung ermöglicht werden. Dieses Feedback ermöglicht dem Entwicklungsteam eine schnellere Behebung von Abweichungen von Spezifikationen oder Fehlverhalten der Software. Es kann durch den agilen Ansatz jedoch nicht die Problematik behoben werden, dass nach Integration von neuer Funktionalität eine Prüfung der bereits bestehenden Funktionen (Regressionstests) notwendig ist, um sicherzustellen, dass diese nicht aufgrund ungewollter Seiteneffekte ein Fehlverhalten aufweist. Mithilfe des Entwicklungsansatzes TDD können diese Seiteneffekte frühzeitig erkannt und vermieden werden. Weiters kann mithilfe von TDD eine höhere Testüberdeckung der entwickelten Funktionalität erreicht werden (Astels, 2003).

2.3.1 Definition Test Driven Development

Janzen (Janzen & Saiedian, 2005) definiert den Begriff *Test Driven Development* als iterativen Ansatz, um Software mit minimalen Vorab-Design zu erstellen. Die dahinterliegende Strategie forciert das Schreiben von automatisierten Tests, bevor der Funktionscode in kleinen und schnellen Iterationen erstellt wird. Weiters bezeichnet er diese Art der Entwicklungsstrategie als eine der wichtigsten Programmierpraktiken (Janzen & Saiedian, 2005).

Bissi (Bissi, Serra Seca Neto & Emer, 2016) bezeichnet Test Driven Development als agile Praxis, die speziell in *Extreme Programming (XP)* an Popularität gewonnen hat und dort als grundlegender Bestandteil definiert wurde (Bissi et al., 2016). Unter *Extreme Programming (XP)* kombiniert man testgetriebene Entwicklung (TDD), Paarprogrammierung und kurze Iterationen, um schnelle Entwicklungsergebnisse in hoher Qualität zu erhalten. (Fojtik, 2011).

2.3.2 Einsatz von Test Driven Development

Der Ansatz Test Driven Development (TDD) kann unabhängig der gewählten Methode des Entwicklungsprozesses (Wasserfall oder Agil) eingesetzt werden. Das Konzept von TDD sieht vor, dass ein Testfall für eine Funktionalität vor dessen konkreter Implementierung erstellt wird. Die Implementierung der Funktionalität endet erst mit dem fehlerfreien Durchlauf des dazugehörigen Testfalls. Der Testfall wird auf Basis der entsprechenden

Anforderung (beispielsweise Komponentenspezifikation) erstellt und schlägt zu Beginn der Entwicklung fehl, da noch keine Implementierung der Funktionalität vorliegt. Der Funktionsumfang der Entwicklung folgt in weiterer Folge dem Umfang des Testfalls, wodurch für jede entwickelte Funktionalität ein entsprechender Testfall vorliegt (Astels, 2003) und eine nachträgliche Anpassung von Codeblöcken erleichtert wird. Im Weiteren ergeben sich nach Astels folgende Vorteile durch TDD:

- Programmierertests anstelle von Unit-Tests: Anstelle implementierten Code auf deren Funktionalität zu prüfen (Unit-Tests) werden Testfälle geschrieben, die definieren, was ein *funktionierender* Code bedeutet.
- Keine Funktionalität geht ohne dazugehörigen Testfall in den Produktionsbetrieb. Mithilfe von TDD kann eine vollständige Codeüberdeckung durch Tests gewährleistet werden.
- Der Inhalt der Testfälle definiert, welcher Code implementiert werden muss. Die Testfälle leiten sich aus den Anforderungen ab und bestimmen daher den Funktionalitätsumfang des zu entwickelnden Codes.

Mithilfe von TDD kann über den Kompiliervorgang festgestellt werden, welche Methoden, Klassen oder Funktionen noch implementiert werden müssen. Der Compiler repräsentiert daher die TODO-List für die Umsetzung. Ähnlich wie bei agilen Softwareentwicklungsmethoden (User-Stories) fördert TDD den Ansatz **Konzeption vor Entwicklung** und stellt das erwartete Handeln in den Vordergrund. Während bei agiler Softwareentwicklung User-Stories die erwartete Funktionalität definieren, definieren die Tests bei TDD diesen Funktionalitätsumfang. Dadurch interagieren agile Softwareentwicklung und testgetriebene Entwicklung (TDD) Hand in Hand (Astels, 2003). Abbildung 2.2 zeigt die jeweiligen Schritte innerhalb eines TDD-Zyklus.

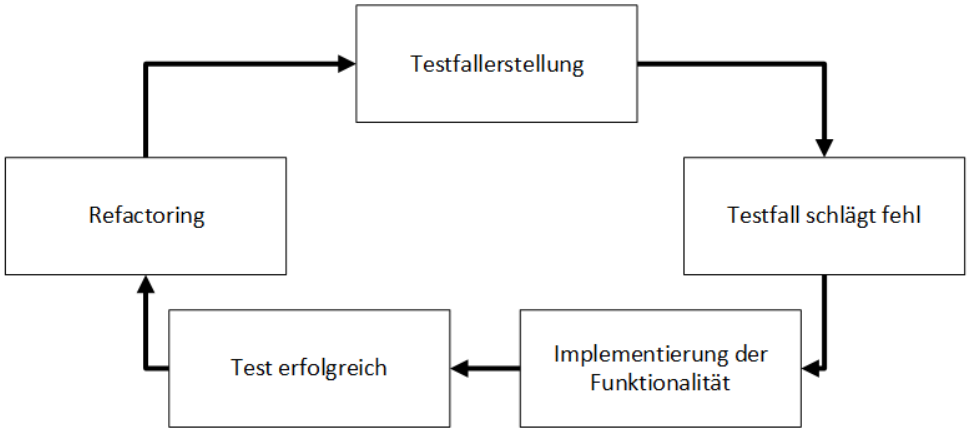

Abbildung 2.3: Der Test-Driven-Development Zyklus. Grafik basiert auf (Suryawanshi, Mai 2019)

Ein weiterer Vorteil von TDD bietet die Möglichkeit, bestehende Implementierungen ohne Risiken nachträglich bearbeiten zu können (Refactoring zu betreiben). Die vorhandenen Tests dienen hierbei als Sicherheitsnetz, da diese das erwartete Ergebnis einer Funktionalität repräsentieren und nicht die Implementierung selbst. Dadurch kann beispielsweise ein entstandener duplizierter Code einfach entfernt und durch eine generalisierte Form ersetzt werden. Nach Ausführung der Tests wird automatisch geprüft, ob die Änderung unerwartete Auswirkungen auf das Systemverhalten zeigt. Durch TDD kann implementierte Funktionalität nach dessen erstmaliger Implementierung leichter in wartbare Blöcke aufgesplittet werden und so zu einer höheren Codequalität beitragen (Astels, 2003).

Der Ansatz von TDD kann auch bei der Automatisierung von Systemtests verwendet werden, da die Testfälle Prozessabläufe des Gesamtsystems abbilden und damit den erwarteten Funktionsumfang eines Prozesses repräsentieren. Erfolgt die Definition der automatisierten Testfälle vor Implementierung der eigentlichen Funktionalität, wird der Zyklus von TDD realisiert und ermöglicht so eine zielgerichtetere Implementierung nach Prozessen. Kapitel 3.4 TDD und Testautomatisierung für Systemtests beschreibt, welche Vor- und Nachteile sich durch die Kombination von TDD und Testautomatisierung ergeben.

2.4 Behavior Driven Testing und Testautomatisierung

Behavior Driven Testing fokussiert in der Testdefinition nicht auf den implementierten Code (wie beispielsweise Unit-Tests), sondern folgt der Prozessbeschreibung aus den Anforderungen (beispielsweise User-Stories aus der Sprintbeschreibung). Dadurch können Testfälle parallel (idealerweise vorher - siehe Abschnitt 2.3 Test Driven Development und Testautomatisierung) zur Implementierung erstellt werden und eine Überprüfung der implementierten Resultate kann automatisiert erfolgen. Abbildung 2.4 zeigt den Unterschied zwischen BTD und einem herkömmlichen Sprintablauf (Kerthyayana Manuaba, 2019).

Während im traditionellen Ablauf automatisierte Tests erst auf Basis des entwickelten Codes realisiert werden, werden mithilfe von BTD die Testfälle basierend auf den Anforderungen erstellt. Dadurch wird explizit das (Prozess)verhalten des Systems getestet und nicht die implementierte Funktionalität, wodurch die Tests unabhängig zur Implementierung werden. Dadurch können Testfälle leichter organisiert werden und erreichen eine höhere Wiederverwendbarkeit, da geänderter Code (Refactoring) keinen

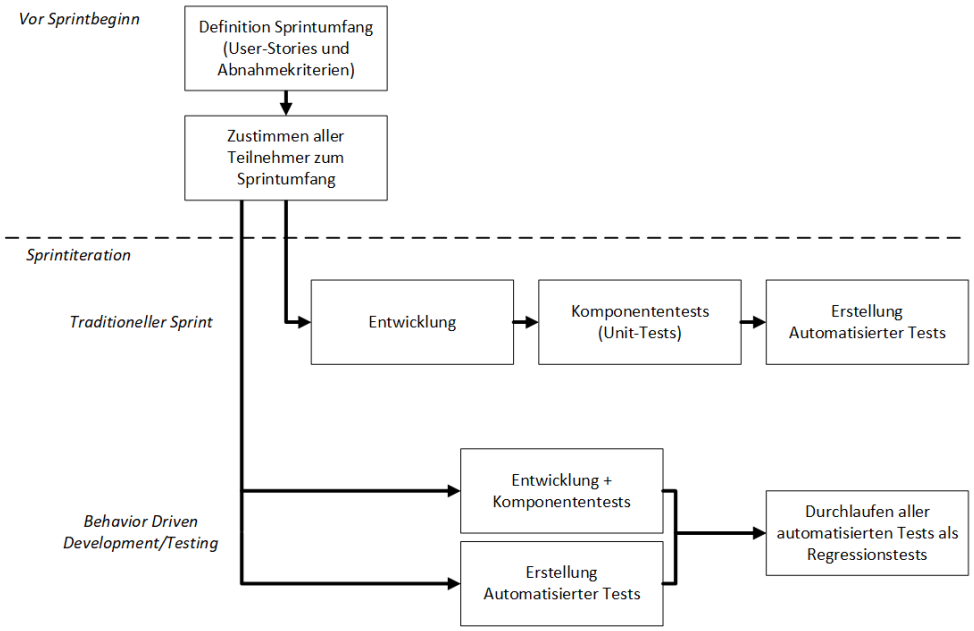

Abbildung 2.4: Ablauf von Sprints (Traditionell vs. BDT). Grafik basiert auf (Kerthyayana Manuaba, 2019)

Einfluss auf das Testergebnis hat (Kerthyayana Manuaba, 2019).

Weiters besagt der BTD Ansatz, dass Anforderungen in einer klar strukturierten Spezifikation erstellt werden sollen, beispielsweise **Given**, **When**, **Then**. Der *Given-When-Then* Ansatz unterteilt eine Spezifikation in drei Abschnitte. Der Erste Abschnitt (Given) beschreibt die Vorbedingungen für den Anwendungsfall (Beispiel: User muss eingeloggt sein). Der zweite Abschnitt (When) beschreibt die Aktion (Verhalten), welche danach ausgelöst wird (Beispiel: Benutzer legt Artikel in Warenkorb). Der letzte Abschnitt (Then) beschreibt das erwartete Ergebnis durch die zuvor ausgelöste(n) Aktion(en) (Beispiel: Anzahl der Artikel im Warenkorb um 1 erhöht.) (Kerthyayana Manuaba, 2019), (Fowler Martin, 2013). Der beispielhafte Code aus 2.1 zeigt eine Spezifikation auf Basis von BTD. Die beiden Schlüsselwörter *FEATURE* und *SCENARIO* dienen zur allgemeinen Beschreibung der Spezifikation, um einen Gesamtüberblick über den zu testenden Prozessablauf zu ermöglichen. Die einzelnen Aktionen werden in den Sektionen *GIVEN*, *WHEN* und *THEN* beschrieben.

Listing 2.1: Aufbau einer Spezifikation nach BTD. Beispiel basiert auf (Fowler Martin, 2013).

```
1 FEATURE: Benutzer befüllt Warenkorb
2   SCENARIO: Artikel dem Warenkorb hinzufügen.
3     GIVEN Es sind 10 Artikel im Produktkatalog verfügbar
4       AND Der Benutzer hat einen registrierten Account
```

```
5       AND Im Warenkorb befinden sich keine Artikel
6
7     WHEN Der Benutzer wählt einen Artikel aus und klickt auf das Symbol '
          Warenkorb hinzufügen', um Artikel dem Warenkorb hinzuzufügen.
8      AND Der Benutzer klickt auf das Symbol 'Warenkorb', um den Warenkorb
          anzuzeigen.
9
10     THEN Der Inhalt des Warenkorbs wird angezeigt
11      AND Die Summe der Artikel im Warenkorb ist 1
12      AND Die Gesamtsumme des Warenkorbs wird angezeigt
```

Viele bestehende Testframeworks unterstützen die Anforderungsspezifikation im Stil von *Given-When-Then*. Testframeworks wie **Cucumber** erweitern den Ansatz um weitere Schlüsselwörter, um Testfälle noch besser für eine Automatisierung beschreiben zu können (Wynne & Hellesøy, 2012) (Fowler Martin, 2013). Neben *Cucumber* gibt es jedoch noch weitere Frameworks und APIs, die den Ansatz von BTD und Given-When-Then unterstützen. Dazu zählt das Framework **Selenium** [1], welches sich speziell für die Testautomatisierung von Webanwendungen eignet oder auch die kommerzielle Umgebung des Herstellers **Ranorex** [2].

Der Ansatz BDT erlaubt also, dass die Spezifikation bereits als Testfallbeschreibung für Systemtests automatisiert herangezogen werden kann, da der Prozessablauf klar strukturiert ist. Die Herausforderung in der vollständigen automatischen Erstellung von Tests liegt darin, die textuellen Beschreibungen aus Vorbedingungen, Aktionen und Resultaten mit implementierten Methoden zu verknüpfen. Dieser Aspekt muss im zu entwerfenden Testkonzept berücksichtigt werden und wird in Abschnitt 4 Konzeptentwicklung erläutert.

[1] https://www.selenium.dev/
[2] https://www.ranorex.com/de/

3 Testframeworks

In folgendem Kapitel werden die gängigsten bereits existierenden Testumgebungen für Testautomatisierung beschrieben und deren Vor- und Nachteile beleuchtet. Abschnitt 3.1 Bestehende Testframeworks für Testautomatisierung beschreibt die jeweiligen Testframeworks und zeigt die Unterschiede zwischen den jeweiligen Systemen auf. In Abschnitt 3.3 Erstellung von automatisierten Tests werden Konzepte zur Erstellung von automatisierten Tests näher beschrieben. Die Erstellung von Tests ist die Schnittstelle zwischen Frameworks mit dem Endanwender und nimmt daher eine zentrale Stelle für die Bedienbarkeit eines Testframeworks ein. Je einfacher neue Testfälle erstellt und integriert werden können, desto eher können Testfälle durch Endanwender (Kunden) erstellt werden, welche meist den gesamten Systemablauf aus Prozesssicht kennen. Im Abschnitt 2.3 Test Driven Development und Testautomatisierung werden Testframeworks mit dem Entwicklungsansatz **Test Driven Development** in Verbindung gesetzt und erläutert, wie Testautomatisierung bei der Umsetzung des Ansatzes unterstützen kann.

3.1 Bestehende Testframeworks für Testautomatisierung

In folgendem Abschnitt werden einige bereits existierende Testumgebungen (Testframeworks) auf deren Möglichkeiten für den Ansatz von **Behavior Diven Testing** sowie **Test Driven Development** analysiert und bewertet. Weiters werden die untersuchten Frameworks näher beschrieben sowie auf Vor- und Nachteile untersucht. Dieser Schritt dient als Basis für die Auswahl des passendsten Frameworks für das Testkonzept aus Kapitel 4 Konzeptentwicklung.

Da das Testkonzept den Fokus auf Webanwendungen (insbesondere auf SAP COMMERCE) legt, müssen die ausgewählten Konzepte die Programmiersprache *JAVA* oder verwandte Scriptsprachen (beispielsweise **JavaScript** oder **Ruby**) unterstützen. Zwar wird die Testausführung und Testabwicklung für End-to-End Tests losgelöst vom *zu testenden System* ausgeführt und betrachtet dieses als Blackbox, jedoch können durch die Nutzung einer einheitlichen Programmiersprache die Aufwände für das initiale Setup sowie die Wartungszeiten reduziert werden. Da SAP COMMERCE auf der Program-

miersprache *JAVA* basiert, kann fundiertes Wissen über diese Programmiersprache vorausgesetzt werden (SAP SE, 2020). Weiters ist laut PYPL Index (Juli 2020) die Programmiersprache *JAVA* nach *Python* die am weit verbreiteste Sprache (Carbonnelle, 2020). Abbildung 3.1 zeigt die zehn beliebesten Programmiersprachen nach PYPL.

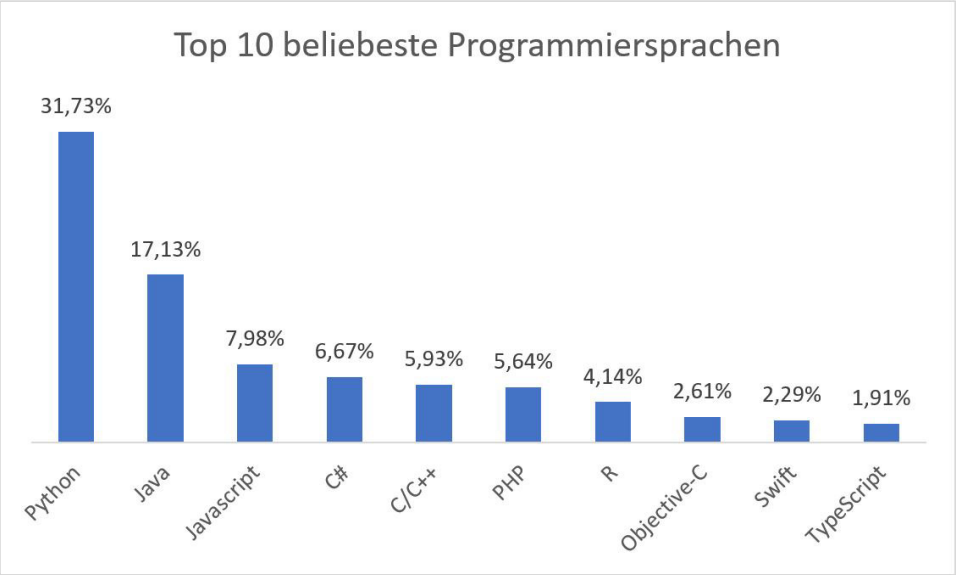

Abbildung 3.1: Die zehn beliebtesten Programmiersprachen nach JYPL. Grafik basiert auf Daten des PYPL Index (Carbonnelle, 2020)

.

Weiters muss bei der Auswahl des Testsystems neben dem Funktionsumfang auch das Lizenzkonzept und der damit verbundene Kostenfaktor berücksichtigt werden. Hier sollen Open Source Lösungen bevorzugt werden, um die Gesamtaufwände für die Systemtests gering zu halten.

3.2 Analyse bestehender Testframeworks für Systemtests

Das internationale Testing Board (Chece, 2016) bietet auf https://www.testing-board.com/testautomatisierung-tools/ eine Sammlung von verbreiteten Tools für Testautomatisierung. Basierend auf den zuvor festgelegten einschränkenden Kriterien wurden die dargelegten Frameworks analysiert und folgende Frameworks für die nähere Analyse ausgewählt:

- Selenium
- CasperJS

- Protractor
- Watir / Watir WebDriver

Die vier aufgelisteten Frameworks werden als Open Source angeboten und unterstützen automatisiertes Testen von End-to-End Prozessen bei Webanwendungen. Auf der Website werden noch weitere kostenpflichtige Frameworks gelistet (beispielsweise **HP Unified Testing**, **Ranorex** oder **Silk Test**), welche die technischen Anforderungen erfüllen. Diese werden jedoch aufgrund der Lizenzkosten in der weiteren Analyse nicht weiter berücksichtigt.

3.2.1 Selenium

Selenium ist ein Framework, um Interaktionen im Browser automatisiert ablaufen zu lassen. Dadurch können Webseiten aus Endanwendersicht (End-to-End) auf unterschiedlichen Browserplattformen (Internet Explorer, Google Chrome, Firefox, ...) getestet werden (Bruns, Kornstadt & Wichmann, 2009).

Weiteres können Benutzerinteraktionen emuliert werden, um Abläufe nachzustellen oder Mehrfachzugriffe (Last/Performancetests) auf das System zu simulieren. Neben dem automatisierten Ablauf der Tests bietet das Framework mit *Selenium IDE* bereits eine Applikation, mit der Abläufe (beispielsweise den Klick auf einen Button oder die Navigation durch Seiten, ...) aufgezeichnet und als Testfall abgespeichert werden können (Selenium HQ, o. J.). So können nach Implementierung einer Funktion auf einfache Art und Weise Regressionstests erstellt und wiedergegeben werden. Nach Checе zählt das Framework zu den populärsten und weit verbreitesten Tools zur Automatisierung von Abläufen in Webanwendungen (Checе, 2016).

Zu den größten Vorteilen des Selenium Frameworks zählt die Unterstützung mehrerer Webdriver, um die gängigen Browser (Google Chrome, Internet Explorer/Microsoft Edge, Firefox, Safari) zu unterstützten sowie die Anbindung an mehrere Programmiersprachen (beispielsweise Java, Phyton, JavaScript, ...). Weiters wird das Framework kostenlos als Open Source Projekt bereitgestellt (Selenium HQ, o. J.).

Als Nachteil kann die Testfallerstellung angeführt werden. Die Testfälle müssen über eine unterstützte Programmiersprache (ähnlich wie bei Unit-Tests) erstellt werden und bedürfen daher Kenntnisse über Softwareprogrammierung. Es ist nicht möglich, Testfälle textuell zu definieren (Beispielsweise über eine bestimmte Form der Anforderungsspezifikation), sondern es bedarf einer expliziten Implementierung der Abläufe. Mithilfe des

Teilprojekts *Selenium IDE* wird ein Addon für Browser bereitgestellt, mit dessen Hilfe Abläufe auf der Website aufgezeichnet und zu einem späteren Zeitpunkt als Regressionstests wiedergegeben werden können.

3.2.2 CasperJS

CasperJS ist eine JavaScript Bibliothek, die automatisiertes Testen von Webanwendungen ermöglicht. Die einzelnen Schritte von Testfällen müssen mithilfe von JavaScript implementiert werden und werden in einem JavaScript basierten Browser (https://phantomjs.org/) ausgeführt. Während JavaScript Unit-Tests die Qualität der JavaScript Funktionen prüfen, kann mithilfe von CasperJS die gesamte Anwendung (Zusammenspiel von JavaScript und dazugehörigem HTML-Seiten) validiert werden (Perriault, o. J.) (Bréhault, 2014).

Da für die Tests eine clientseitige Scriptsprache verwendet wird, kann auf einzelnen Seitenelemente (beispielsweise Überschrift, Bild, Paragraph) direkt mittels JavaScript oder JQuery über den HTML-DOM darauf zugegriffen werden. CasperJS bietet darüber hinaus weitere Funktionen um Tests abzubilden. Der Einsatz von JavaScript für die Implementierung von Testfällen ist der größte Vorteil sowie Nachteil des Frameworks. Einerseits ist das Framework aufgrund der expliziten Verwendung von JavaScript auf Webanwendungen limitiert, anderseits können aufgrund der Nutzung einer Scriptsprache schnell Testfälle erstellt werden. Ein weiterer Vorteil von CasperJS ist die plattformunabhängige Einsatzmöglichkeit. Der Einsatz von CapserJS ermöglicht auf einfache Art das Erstellen von funktionalen Tests für Webanwendungen (Bréhault, 2014).

Wie bereits beim Testframework *Selenium* müssen die Tests explizit implementiert werden und können nicht über eine Metasprache textuell erfasst werden.

3.2.3 Protractor

Das Framework Protractor ermöglicht die Erstellung von End-to-End Tests für Angular oder AngularJS Webanwendungen. Für die Tests nutzt Protractor die Selenium Webdriver API, um Userinteraktionen zu simulieren (Protractor, o. J.).

Protractor dient also als Schnittstelle zwischen Angular und Selenium und übernimmt den Datenaustausch zwischen den beiden Applikationen. Die Daten werden über HTTP mittels JSON-Format übertragen und entsprechend interpretiert. Der Selenium Server

interpretiert die Befehle aus dem übermittelten Test, um die Steuerung des Browsers durchzuführen und sendet die resultierenden Ergebnisse zurück an das Protractor Framework (Protractor, o. J.).

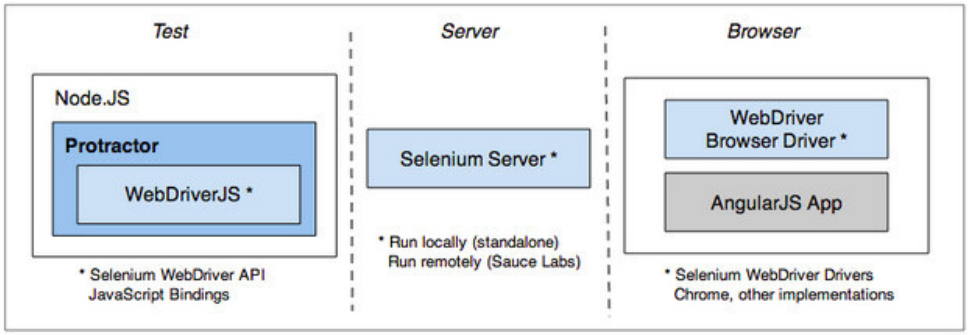

Abbildung 3.2: Aufteilung der Funktionen zwischen Protractor und Selenium (Protractor, o. J.)

.

Abbildung 3.2 zeigt die Aufteilung der Funktionen zwischen *Protractor* und *Selenium*. Protractor dient als Wrapper für WebDriverJS, welcher die Kommunikation mit dem Selenium Server und der tatsächlichen Testausführung übernimmt. Dieser Wrapper ist notwendig, da der Aufruf des Selenium Servers asynchron erfolgt und die Rückmeldungen von Protractor synchronisiert werden müssen (Protractor, o. J.).

Protractor kann also als Erweiterung von *Selenium* für Angular oder AngularJS angesehen werden. Protractor übernimmt lediglich die Übersetzung der definierten Teställe in Angular und leitet diese zur Testausführung an den Selenium Server weiter. Die Testausführung sowie die Interaktion mit dem Browser selbst wird durch den *Selenium Webdriver* ausgeführt und gesteuert. Die Ergebnisse werden von Protractor synchronisiert und gesammelt als Testergebnis ausgewertet.

3.2.4 Watir / Watir WebDriver

Watir ist ein Open Source Projekt, um automatisierte Tests mithilfe der Scriptsprache Ruby zu erstellen. Ähnlich wie *Protractor* dient Watir als Erweiterung von Selenium für die Scriptsprache Ruby. Die Testausführung (automatisierte Interaktion am Browser) wird mithilfe der Selenium API durchgeführt und das Ergebnis durch Watir in der Scriptsprache Ruby zurückgeben (Watir, 2018).

Der Ablauf der Testausführung in Watir wird in Abbildung 3.3 dargestellt. Initial werden die Tests in einem sogenannten **Test Runner** erstellt. Dieser Test Runner enthält die

Beschreibung des Testfalls sowie die einzelnen Ausführungsschritte. Diese Ausführungsschritte referenzieren auf Funktionen oder Methoden (**Test Code**), welche auf dem jeweiligen Testobjekt (**Page Object**) ausgeführt werden sollen. Die Übersetzung des Testfalls in automatisiert ausführbaren Code erfolgt im Abschnitt **Watir Code**, welcher die Ausführungsschritte in automatisierbare Aufrufe für Selenium und JavaScript umwandelt und an die **Selenium API** weitergibt. Die **Selenium API** leitet den Code an den Browser weiter und übernimmt die automatisierte Bedienung des Browser und die damit verbundene automatisierte Testausführung im **Browser** (Watir, 2018)

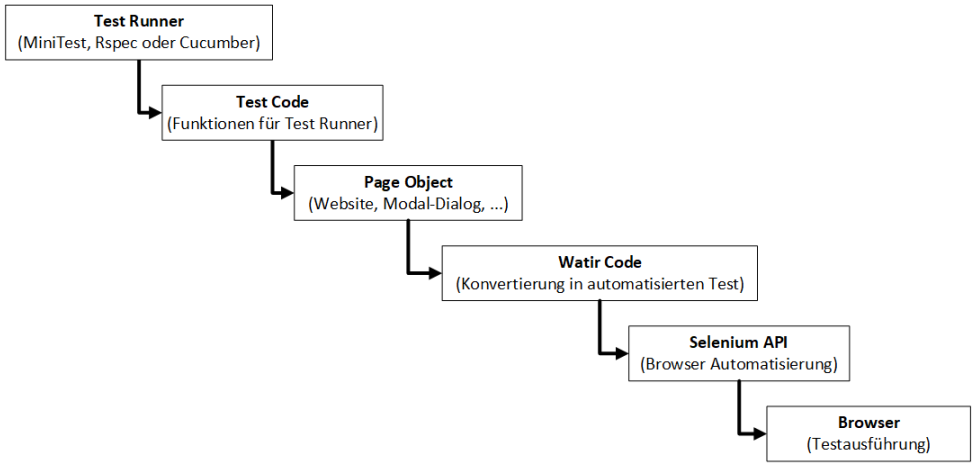

Abbildung 3.3: Testausführung mittels Watir. Grafik basiert auf (Watir, 2018)

.

Ähnlich wie *Protractor* dient Watir als Erweiterung von *Selenium*, wobei Watir noch einen Schritt weiter geht und die Testerstellung abstrahiert und die Erstellung dieser in Frameworks wie *Cucumber* oder *Rspec* ermöglicht. Damit müssen Testfälle nicht mehr explizit via Code implementiert, sondern können auch ohne Programmierkenntnisse erstellt werden, wenn entsprechende implementierte Funktionen vorliegen. Dadurch können Testszenarien direkt vom Kunden/Endanwender definiert und die Konzepte von Behavior Driven Testing sowie Test Driven Development mit Testautomatisierung verknüpft werden.

3.3 Erstellung von automatisierten Tests

Sämtliche in Abschnitt 3.2 Analyse bestehender Testframeworks für Systemtests genannten Frameworks ermöglichen automatisiertes Testen für Webanwendungen. Die Erstellung der automatisierten Tests muss jedoch meist programmiertechnisch via Code erfolgen, wodurch die Tests eher der implementierten Funktionalität folgen als der Anfor-

derungsspezifikation. Um Tests basierend auf der Anforderungsspezifikation (Behavior Driven Testing) zu ermöglichen und damit auch den Ansatz von Test Driven Development umzusetzen, müssen Testfälle in einfacher Sprache textuell formuliert werden können. Diese Testspezifikationen müssen im weiteren Schritt maschinell übersetzt werden, um automatisiert ausführbar zu sein. Diese textuelle Beschreibung und Übersetzung kann mithilfe des Frameworks **Cucumber Open** erfolgen (Wynne & Hellesøy, 2012) (SmartBear Software, 2020).

Cucumber ermöglicht die Testspezifikation nach Behavior Driven Development in textueller Form nach spezifischer Syntax (Gherkin) und verknüpft diese mit implementierten Funktionsbausteinen zur Testausführung. Diese Implementierung kann in diversen Programmiersprachen (Java, JavaScript, Ruby, ...) erfolgen und beinhaltet die Verknüpfung zu den, in Abschnitt 3.2 Analyse bestehender Testframeworks für Systemtests genannten, Frameworks.

Die Syntax von *Gherkin* dient dazu, die (in unstrukturierten Text) vorliegende Testfallbeschreibung so zu strukturieren, um diese für eine automatisierte Testausführung zu verwenden. Mithilfe von Gherkin wird der Text mit Unterstützung von Schlüsselwörtern in Abschnitte (Vorbedingung, Ausführung, Testergebnis) unterteilt. Die darin befindlichen Textdefinitionen einzelner Schritte (*Step definitions*) werden im Code über *Marker* verknüpft. Bei Testausführung wird diese Verknüpfung aufgelöst und der dahinterliegende Code ausgeführt (SmartBear Software, 2020).

Listing 3.1 zeigt ein Beispiel einer *Step Definition* mithilfe von Gherkin. Das Schlüsselwort *GIVEN* beschreibt eine Vorbedingung, welche dem Schlüsselwort in textueller beschreibender Form nachgestellt ist. Diese Beschreibung wird als *Marker* für die Verknüpfung des dahinterliegenden Codes verwendet.

Listing 3.1: Aufbau eines Schrittes mittels Gherkin-Syntax. Beispiel basiert auf (SmartBear Software, 2020).

```
1    GIVEN Es sind 10 Artikel im Produktkatalog verfügbar
```

Die Verknüpfung des Testschritts mit dem dazugehörigen Code wird in Listing 3.2 dargestellt. Die Schrittbeschreibung (*Es sind 10 Artikel im Produktkatalog verfügbar*) wird als Verknüpfungstext zwischen Beschreibung und Testmethode genutzt.

Listing 3.2: Verknüpfung des Testschritts mit Ausführungscode. Beispiel basiert auf (SmartBear Software, 2020).

```
1  package com.example;
2  import io.cucumber.java8.En;
3
4  public class StepDefinitions implements En {
5      public StepDefinitions() {
6        @Given( "Es sind {int} Artikel im Produktkatalog verfügbar",
7                (Integer article) -> {
8                  //TODO: Anzahl der Artikel prüfen
9              }
10       );
11     }
12 }
```

Das grundlegende Konzept der Verknüpfung von textueller Testbeschreibung und dahinterliegendem ausführbaren Code kann als Konzeptansatz für die Konzeptfindung in Kapitel 4 dienen. Diese Verknüpfung ermöglicht die Realisierung von Test Driven Development, da Anforderungen bereits als Testfälle im Zuge der Sprint-Planung definiert werden können. Da auch der Code zur Testfallausführung losgelöst von der Programmlogik existieren kann (beispielsweise *Seite öffnen*, *Klick auf Button*, *Eingabe in Textfeld*) kann TDD sowie BDT konzeptionell umgesetzt werden. Grundvoraussetzung dafür ist jedoch eine abstrahierte Sicht auf die eigentliche Funktionalität, sodass Ausführungsschritte möglichst atomar gehalten werden. So bedarf beispielsweise die Funktion *Seite öffnen* lediglich den Funktionsumfang des Ladens der übergebenen URL und nicht etwa zusätzlich die Prüfung, ob eine Überschrift gesetzt ist. Nur durch diese atomare Funktionsweise können Funktionen häufig wiederverwendet werden.

3.4 TDD und Testautomatisierung für Systemtests

Wie bereits in Abschnitt 3.3 Erstellung von automatisierten Tests dargelegt, kann der Entwicklungsansatz *Test Driven Development* mit Frameworks zur Testautomatisierung interagieren. Für eine Umsetzung von TDD bei Testautomatisierung müssen folgende Kriterien erfüllt werden:

- Testfälle müssen textuell erstellt werden können und die einzelnen Testschritte mit ausführbarem Code hinterlegt werden. Dadurch können die Testfälle vorab als Anforderungsbeschreibungen erstellt sowie als spätere Funktionsdokumentation verwendet werden.
- Die Testschritte und der dazugehörige ausführbare Code muss allgemein und im

Funktionsumfang atomar gehalten werden. Dadurch können implementierte Funktionen häufig wiederverwendet werden und die Formulierung der Testbeschreibung bleibt über alle Testfälle hinweg annähernd ident.

- Der Code zur Testausführung muss vom Code der Applikation getrennt werden. Dadurch kann der Testfall vorab implementiert und der Ansatz von **TDD** realisiert werden.

Um *Test Driven Development* effizient umsetzen zu können, bedarf es jedoch einer klaren Struktur über den gewünschten Funktionsumfang einer Funktionalität, um die Testfallbeschreibung entsprechend genau zu erstellen. Diese Beschreibung ist essenziell, um einerseits den Testfall entsprechend genau definieren zu können und andererseits auch später die produzierte Funktionalität (Implementierung) entsprechend zu prüfen. Die automatische Ausführung der Testfälle hilft im Zuge der Entwicklung als Prüfung der Funktionalität und nach Fertigstellung als Regressionstests bei der Weiterentwicklung oder bei Wartungsarbeiten.

3.5 Zusammenfassung

Die Analyse von bestehenden Testframeworks zeigt, dass Ansätze für eine automatisierte Testausführung für Webanwendungen existieren. Diese Frameworks können als Basis für ein ganzheitliches Testkonzept verwendet werden, um das definierte User-Verhalten in den Tests automatisiert simulieren zu können. Basierend auf der Analyse von bestehenden Testframeworks für End-To-End Tests können folgende Schlüssel für die Konzeptentwicklung gezogen werden:

- Die Aufschlüsselung des Frameworks **Watir** kann als Vorlage für die Strukturierung des Testkonzepts dienen. Eine Trennung der textuellen Testbeschreibung sowie die Umwandlung der Testschritte in ausführbaren Code ermöglicht die Realisierung von BDD und TDD.
- Mit **Cucumber** gibt es ein Framework, um das Testkonzept BDD umzusetzen. Mithilfe der Syntax *Gherkin* können Testfallbeschreibungen mit ausführbarem Code verknüpft und so textuelle Testschrittbeschreibungen in ausführbare Testfälle umgewandelt werden.
- Für die Automatisierung der Interaktion im Browser wird hauptsächlich auf **Selenium API** gesetzt. In Kombination mit dem **Selenium Webdriver** können die bekanntesten Browser (Google Chrome, Mozilla Firefox, Internet Explorer/Micro-

soft Edge) automatisiert bedient werden.

- Es existieren bereits einige Frameworks für Testautomatisierung für verschiedenste Programmiersprachen. Sämtliche untersuchte Frameworks setzen im Backend auf die Schnittstellen der **Selenium API** und erweitern diese API lediglich um die spezifischen Eigenschaften der Programmiersprachen.

Im Zuge der Analyse konnte noch keine vollständige Schnittstelle zur Bereitstellung der Testergebnisse identifiziert werden. Einzelne Frameworks resultieren die Testergebnisse direkt nach der Ausführung ohne Persistierung. Die Bereitstellung der Testresultate zu einem späteren Zeitpunkt (beispielsweise als REST-Schnittstelle) ist jedoch von essenzieller Bedeutung, um die Qualität des Projektes zu überwachen und entsprechende Maßnahmen abzuleiten. Nur eine aktive und regelmäßige Kontrolle der Testergebnisse ermöglicht eine Qualitätsverbesserung. In der Entwicklung des Testkonzepts muss also darauf geachtet werden, eine Schnittstelle zur übersichtlichen Darstellung der Testergebnisse an zentraler Stelle (beispielsweise in einem Dashboard) zu ermöglichen.

Weiters muss das Testkonzept eine regelmäßige Testausführung gewährleisten. Nur durch die regelmäßige Ausführung der Testfälle und der damit verbundenen Überprüfung können Abweichungen im System identifiziert werden. Dazu muss eine Schnittstelle geschaffen werden, um die Tests über ein Build-Tool (beispielsweise Jenkins) zu starten.

3.6 Nutzwertanalyse

Um eine Auswahl der untersuchten Frameworks für die technische Umsetzung zu treffen, wurden die untersuchten Frameworks mithilfe einer Nutzwertanalyse nach ausgewählten Kriterien bewertet und verglichen. Dadurch werden die einzelnen Vor- und Nachteile dargelegt. Als zu bewertende Kriterien wurden folgende Punkte ausgewählt:

- **Funktionsumfang**: Welchen Umfang bietet das Framework? Da das entwickelte Testkonzept flexibel in der Anwendung sein soll, muss das darunterliegende Framework über einen hohen Funktionsumfang verfügen.
- **Dokumentation**: Wie gut ist die Dokumentation der API des Frameworks? Nur mithilfe gut dokumentierter APIs kann ein effizienter Einsatz des Frameworks erfolgen.
- **Aktualität**: Wird das untersuchte Framework noch aktiv weiterentwickelt oder wurde die Entwicklung eingestellt?

- **Verbreitung**: Wie oft wird das Framework eingesetzt? Gibt es, abseits der Dokumentation, Foren oder Hilfe im Web?
- **Unterstützte Programmiersprachen**: In wie vielen Programmiersprachen kann das Framework eingesetzt werden? Diese sind besonders wichtig, um eine weite Verbreitung des Frameworks zu ermöglichen.

Basierend auf diesen Kriterien erfolgt eine Bewertung der Wichtigkeit der einzelnen Punkte pro Framework. Die Wichtigkeit wird in 5 Stufen unterteilt, wobei 5 die vollständige Erfüllung und 0 keine Erfüllung des Kriteriums bedeutet. Das Framework mit der höchsten Gesamtpunktzahl verfügt über den höchsten Nutzen der untersuchten APIs für das definierte Ziel (Ganzheitliches Testframework). Würde ein Framework sämtliche Punkte vollständig erfüllen (5 Wichtigkeit), ergäbe die erreichte Punktzahl 100. Somit spiegelt die absolut erreichte Punktzahl in der Nutzwertanalyse zugleich auch den Prozentsatz der Erfüllung der Gesamtkriterien.

Die Tabelle in Abbildung 3.1 zeigt die erstellte Nutzwertanalyse, welche zur Entscheidungsfindung für die technische Umsetzung (siehe 4.4 Technische Umsetzung) als Basis dient. In den einzelnen Zellen der Bewertung repräsentiert die Zahl die erreichte Punktzahl und die Zahl in den Klammern den summierten Wert mit der Wichtigkeit in der Nutzwertanalyse.

Tabelle 3.1: Nutzwertanalyse der untersuchten Frameworks

Kriterium	**Wichtigkeit**	**Selenium API**	**CasperJS**	**Protractor**	**Watir** **Watir WebDriver**
Funktionsumfang	5	4 (20)	3 (15)	3 (15)	3 (15)
Dokumentation	4	4 (16)	4 (16)	4 (16)	2 (8)
Aktualität	5	5 (25)	5 (25)	4 (20)	5 (25)
Verbreitung	3	5(15)	4 (12)	3 (9)	3 (9)
Programmiersprachen	3	4 (12)	3 (9)	1 (3)	1 (3)
Nutzwert		**88 %**	77 %	63 %	60 %

Basierend auf den Ergebnissen der Nutzwertanalyse wird für die technische Umsetzung auf das Testframework **Selenium API** gesetzt. Dieses Framework bietet den größten Funktionsumfang und verfügt weiters über eine gute Dokumentation und einer aktiven Nutzergemeinschaft. Weiters wird das Framework **Cucumber Open** eingesetzt, um den Ansatz von *Behavior Test Driven* zu ermöglichen und Testfälle in einfacher textuell beschreibender Form abzubilden.

4 Konzeptentwicklung

Basierend auf den Ergebnissen in den Kapiteln 2 Testen im Softwareentwicklungsprozess und 3 Testframeworks wurde ein Konzept zur Implementierung einer Umgebung für automatisierte Tests entworfen, welches bereits etablierte Open-Source Tools in der Softwareentwicklung verknüpft und dadurch ein Testframework bereitstellt, welches folgende Funktionalitäten umfasst:

- Erstellung von durchgängigen Testfällen in textueller Form zur Umsetzung von **Behavior Driven Testing** für End-to-End Tests
- Unabhängige Codebasis der Testschritte zur Umsetzung von **Test Driven Development**
- **Automatisiertes Ausführen der Testfälle** und damit verbundene Emulation der Benutzerinteraktionen bei Webanwendungen
- Bereitstellen einer **Schnittstelle zur Bereitstellung der Testresultate** zur Darstellung in externen Analyse- (beispielsweise Dashboards) oder Managementsystemen.

Das erstellte Konzept fokussiert auf das Testen von Webanwendungen, insbesondere auf SAP Commerce. Nichtsdestotrotz können die grundlegenden Überlegungen auf andere Anwendungen übernommen werden, lediglich die darunterliegenden Frameworks müssen entsprechend auf die Bedürfnisse des zu testenden Systems angepasst werden. Der detaillierte Funktionsumfang des ganzheitlichen Testkonzepts wird in den folgenden Abschnitten beschrieben.

4.1 Funktionsweise und Funktionsumfang

Dieser Abschnitt beschreibt den Gesamtumfang des Testkonzepts in den jeweiligen Phasen der Testausführung. Die einzelnen Teilbereiche können unabhängig betrieben werden. Die Ergebnisse (Daten) der jeweiligen Phase werden an die jeweilig nachfolgende Phase weitergereicht und dienen als Input für die Prozessverarbeitung. Dadurch können die jeweiligen Phasen organisatorisch unabhängig voneinander betrieben werden und an die Phasen einer Projektplanung nach Scrum angelehnt werden. Die agile

Projektplanung nach Scrum umfasst ebenfalls drei große iterative Phasen für einen Sprint. Die Sprintplanung beginnt mit der *Erhebung des Funktionsumfangs* und dem Aufbau des Sprint-Backlogs aus dem Produkt-Backlog. Diese Phase folgt der Umsetzung der definierten Funktionen in der *Sprintausführung* und endet mit der Abnahme der implementierten Funktionen im *Sprint-Review*. Meist erfolgt die Abnahme des aktuellen Sprints zeitgleich mit der nächsten Sprintplanung des nachfolgenden Sprints (Wirdemann & Mainusch, 2017).

Abbildung 4.1 zeigt die drei großen Phasen von Scrum und die mögliche Aufteilung des Funktionsumfangs des Testkonzepts auf die jeweilige Phase. Während der Sprintausführung können neben der Testausführung zur Validierung und Verifikation auch Testergebnisse ausgewertet oder neue Testfälle vor Entwicklung einer Funktionalität erstellt werden. Die Abbildung zeigt lediglich eine grobe Zuordnung der Funktionalität zu den Phasen im agilen Projektmanagementkonzept nach SCRUM.

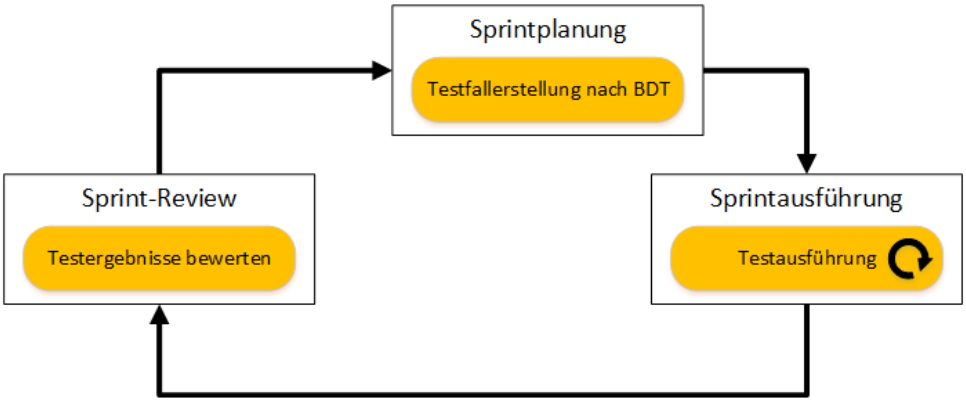

Abbildung 4.1: Phasen der Projektplanung nach Scrum in Verbindung mit Teilbereichen des Testkonzepts

Durch die Trennung der Funktionalität kann diese organisatorische Verantwortung auf mehrere Personen (Gruppen) verteilt werden. Beispielsweise kann die Testfallerstellung im Zuge der Sprintplanung durch den Kunden erfolgen, während die technische Testausführung in Verantwortung des Entwicklungsteams liegt. Die Testergebnisse können im Sprint-Review durch das gesamte Team analysiert und bewertet werden.

4.1.1 Testfallerstellung

Die Testfallerstellung soll über eine Webapplikation an zentraler Stelle möglich sein und sich an die Syntax von *Gherkin* des BDT-Framework *Cucumber* anlehnen. Die Oberfläche soll intuitiv bedienbar sein, um eine Testfallerstellung auch aus nicht technischer

Sicht zu ermöglichen (Beispielsweise Kundensicht). Die Webapplikation kann über den bereits vorhandenen Webserver der zu testenden Webanwendung (beispielsweise SAP Commerce) zentral zur Verfügung gestellt werden.

Die einzelnen Testfälle sollen in Testsets(Test-Suite) nach logischen Prozessabläufen gruppiert und als Datei im Flat-File Format abgespeichert werden können. Die erstellte Datei kann danach mittels Versionskontrollsystemen (beispielsweise GIT, SVN, CVS) verwaltet (veröffentlicht) werden. In der Phase der Testausführung kann die veröffentlichte Datei geladen und ausgeführt werden, wodurch die Aktualität der Testfälle gewährleistet wird.

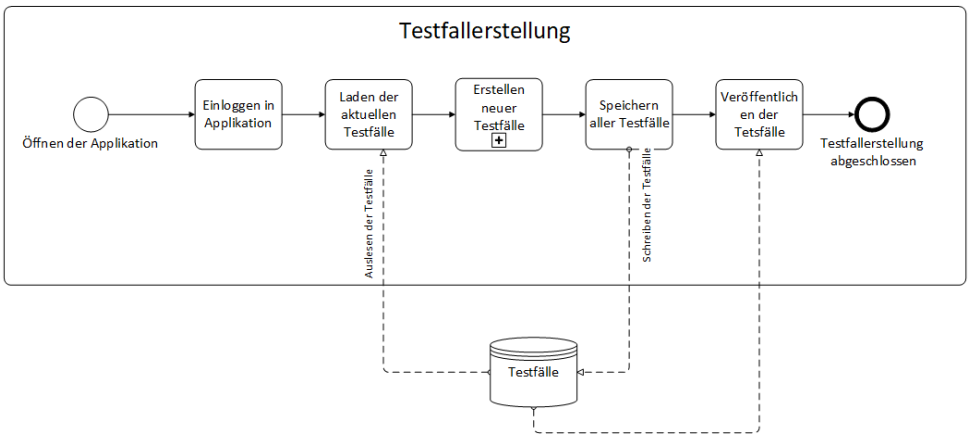

Abbildung 4.2: Prozessablauf der Testfallerstellung als BPMN Prozessmodell

Abbildung 4.2 zeigt den Prozessablauf zur Erstellung eines Testfalls in BPMN Notation. Der Ablauf soll eine Übersicht über die jeweiligen Schritte innerhalb der Testfallerstellung zeigen. Die detaillierten Ablaufschritte zur Testfallerstellung wurden im Unterprozess *Erstellen neuer Testfälle* gruppiert.

Die Webapplikation liefert als Ergebnis eine Datei, die dem Format des Frameworks **Cucumber Open** entspricht. Dies ist notwendig, da die Validierung der erstellten Testfälle sowie die Umwandlung in ausführbare Tests in der Phase *Testausführung* über **Cucumber Open** erfolgt. Die Webapplikation dient also als Interface zur einfachen Erzeugung von Testfällen in textueller Form, welche mithilfe des Frameworks **Cucumber Open** in der Phase *Testausführung* in ausführbaren Code umgewandelt werden. Die detaillierte Dokumentation zur technischen Umsetzung wird in Abschnitt 4.4.4 Erstellen automatisierter Tests (AWATAR-Creator) beschrieben.

4.1.2 Testausführung

Die Testausführung ist grundsätzlich in zwei Prozessschritte untergliedert. Im ersten Schritt werden die (textuell vorliegenden) Testfälle eingelesen, validiert und mit dem dazugehörigen ausführbaren Code verknüpft. Bei der Validierung wird für die einzelnen Testschritte jeweils eine ausführbare Methode im Testcode identifiziert zugeordnet.

Ist eine Verknüpfung nicht möglich, wird ein Standard-Baustein mit dem Testfall verknüpft, um eine Testausführung dennoch zu gewährleisten. Dieser Standard-Baustein verursacht zwingend einen Fehlschlag des Tests, um eine Analyse des Testfalls im Zuge der Auswertung der Testergebnisse zu forcieren. Durch diese Vorgehensweise kann die textuelle Beschreibung des Testfalls von der technischen Implementierung losgelöst erfolgen, wodurch das Konzept von **Test Driven Development** auch in der Testfallerstellung einfließt. Die Definition des Testfalls nach **Behavior Driven Testing** erfolgt in der vorhergehenden Phase der Testfallerstellung, während die Implementierung des Testfalls nachträglich erfolgt (sofern die aufzurufende Funktionalität nicht bereits existiert).

Danach werden die einzelnen Bausteine zu einem automatisierten Testfall zusammengefügt und für die Ausführung bereitgestellt. Nach Transformation sämtlicher Testfälle werden die erstellten Testfälle auf Ausführbarkeit geprüft und nur zur Ausführung an den *Test Runner* weitergegeben, sofern dies fehlerfrei möglich ist. Abbildung 4.3 zeigt die Prozessschritte für die Transformation. Die technische Umsetzung erfolgt über das Framework *Cucumber Open*, welches bereits die Funktionalität zur Verknüpfung von textuellen Testfällen mit ausführbarem Code bereitstellt. Die Applikation erweitert dieses Framework um die Funktionalität des Einlesens von Testfällen sowie der Optimierung in der Erstellung der ausführbaren Testfälle.

Im zweiten Schritt werden die transformierten Testfälle an den *Test-Runner* zur tatsächlichen Testausführung weitergegeben. Dieser liest die transformierten Testfälle ein und führt die einzelnen Schritte aus. Dazu wird das Framework der *Selenium API*, insbesondere der *Selenium WebDriver*, verwendet. Diese Applikation liest automatisierbare Testfälle ein und leitet die Aktionen an den Browser zur Automatisierung weiter. Somit werden die einzelnen Schritte des Testfalls im Browser ausgeführt. Das daraus resultierende Ergebnis wird mit dem erwarteten Ergebnis aus dem Testfall verglichen und entsprechend ein Testergebnis protokolliert. Im Fall von Webanwendungen und im vorliegenden Testkonzept übernimmt diesen Teil das Framework der *Selenium API*.

Die gesamte technische Dokumentation zur Umsetzung der Applikation zur automati-

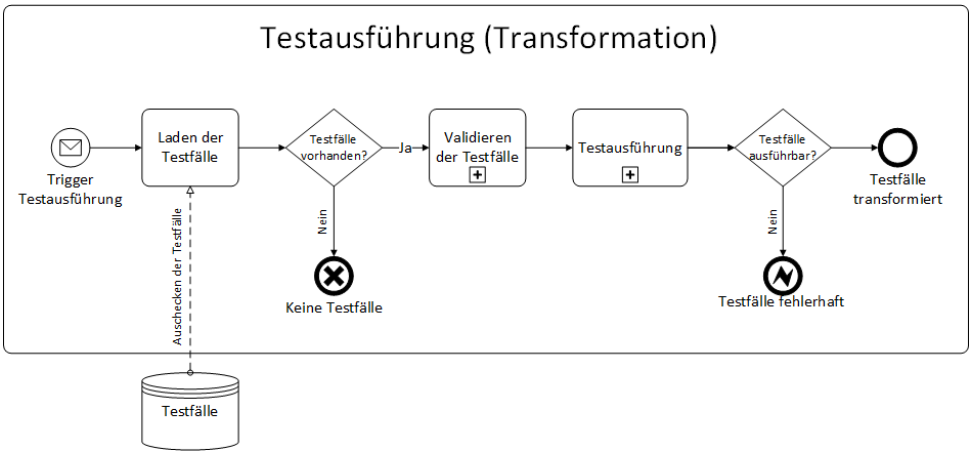

Abbildung 4.3: Prozessablauf der Testfalltransformation als BPMN Prozessmodell

sierten Testausführung wird in Abschnitt 4.4.5 Testausführung (AWATAR-Runner) beschrieben.

4.1.3 Bereitstellung der Testergebnisse

In der dritten Phase der automatisierten Testausführung werden die gesammelten Testergebnisse aufbereitet und für eine übersichtliche Darstellung bereitgestellt. Dazu werden die Ergebnisse aus der Testausführung gesammelt und in ein JSON-Format umgewandelt. Das resultierende JSON-Format wird über eine REST-Schnittstelle über HTTP für externe Systeme bereitgestellt und kann so in die spezifische Systemlandschaft des Unternehmens eingebunden werden. Das bereitgestellte Format JSON wird gewählt, da es sich dabei um ein weitverbreitete Datenformat für den Datenaustausch nach derzeitigem Technologiestand handelt.

Abbildung 4.4 zeigt den Prozessablauf der Sammlung der Testergebnisse und Transformation in JSON-Format und die Bereitstellung über eine REST-Schnittstelle. Die erstellten Ergebnisse werden persistiert und registrierte Systeme über den Inhalt neuer Testergebnisse informiert. Mithilfe eines *Public/Subscribe*-Konzepts können die Ergebnisse sofort nach Bereitstellung dieser an Drittsysteme weitergeben werden.

Das dazu erstellte Service liest die Ergebnisse aus der Testausführung aus, und wandelt diese in ein JSON-Format für die weitere Bereitstellung über eine HTTP-Schnittstelle um. Die Applikation der Testausführung triggert nach Ausführung der Tests das Service, welches die eingehenden Daten ausliest, aufbereitet und archiviert. Danach werden Interessenten (Systeme) über neue vorliegende Ergebnisse informiert.

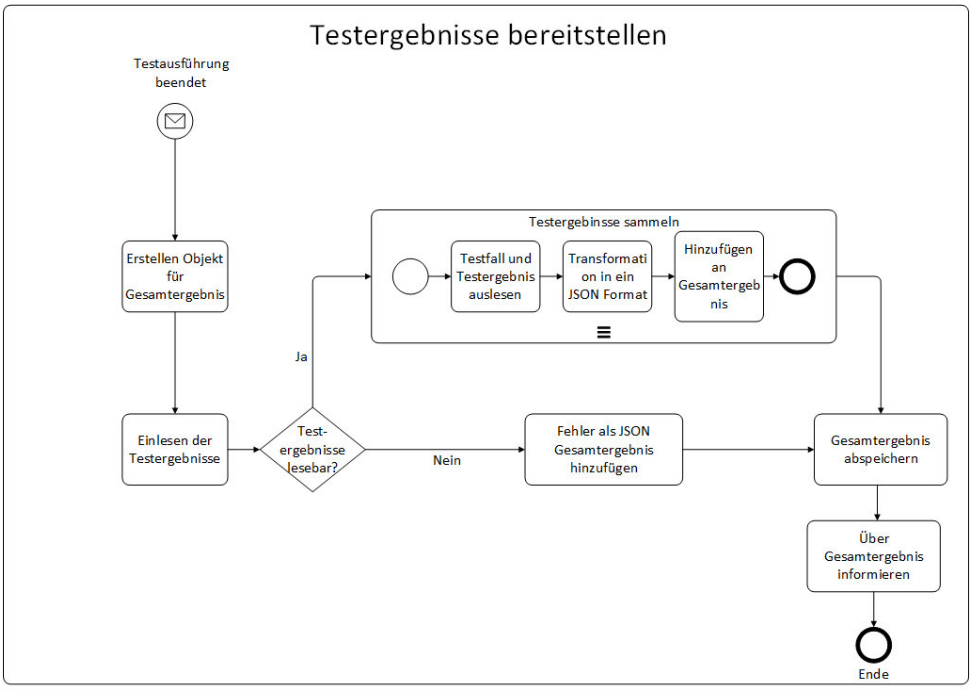

Abbildung 4.4: Prozessablauf der Bereitstellung der Testergebnisse als BPMN Prozessmodell

4.2 Abgrenzung zu alternativen Testframeworks

Das beschriebene Testkonzept unterscheidet sich zu alternativen Frameworks dadurch, dass eine ganzheitliche Integration ermöglicht wird. Während einzelne Frameworks wie **Selenium**, **Watir** oder **CasperJS** lediglich den Teil der Testausführung bereitstellen, können mit dem beschriebenen Konzept Testfälle erstellt, ausgeführt und die Ergebnisse entsprechend verarbeitet werden. Es verknüpft also die Funktionalität mehrerer bereits vorhandener Frameworks und ermöglicht durch Schnittstellenimplementierung die durchgängige Nutzung innerhalb des Testprozesses (von der Erstellung bis zur Ergebnisverarbeitung).

Als Basis wird das Framework **Cucumber Open** verwendet, welches die Funktionalität der Testfallerstellung nach BDT und eine Testausführung ermöglicht. Diese Funktionalität wird um eine grafische Benutzeroberfläche zur Testfallerfassung erweitert, um diese auch aus nicht technischer Sicht zu ermöglichen. Weiters wird mithilfe eines Webservices eine Schnittstelle geschaffen, die die Testergebnisse für externe Systeme bereitstellt.

Es handelt sich bei dem erstellten Testkonzept also um eine Verknüpfung bestehender Frameworks und Funktionalitäten. Diese Verknüpfung schafft ein ganzheitliches System, um automatisierte End-to-End Testfälle nach BTD zu erstellen. Durch die Trennung der Testfallerstellung und -ausführung kann weiters das Entwicklungskonzept Test Driven

Development realisiert werden.

4.3 Integration in bestehende Infrastruktur

Durch die Entwicklung der Schnittstelle kann das entwickelte Testkonzept in die bereits vorhandene Infrastruktur eines Projektes integriert werden. Für die Testerstellung wird eine eigenständige Webapplikation bereitgestellt, welche jedoch durch eigene Applikationen ersetzt werden kann. Diese Applikationen müssen lediglich den Export der Testfälle nach dem Gherkin-Format des Frameworks *Cucumber Open* untersützen. Zur Testausführung werden die erstellen Testfälle im entsprechenden Format eingelesen und mit individuell implementiertem Testcode verknüpft und ausgeführt.

Die daraus resultierenden Testergebnisse werden über eine standardisierte REST-Schnittstelle bereitgestellt, sodass diese von jedem externen System eingelesen werden können. Die interessierten Systeme werden von einem Webservice über neue Ergebnisse informiert und können diese zu jedweglichen Zeitpunkt entsprechend abholen.

4.4 Technische Umsetzung

Folgender Abschnitt beschreibt die technische Realisierung des zuvor beschriebenen Konzepts. Das entwickelte System wurde unter dem Projektnamen **AWATAR** (*A Web Application Test Automation Resource*) entwickelt und wird in den Folgekapiteln und Abschnitten entsprechend bezeichnet. Die folgenden einzelnen Unterabschnitte beschreiben die drei Teilbereiche des Gesamtkonzepts von *AWATAR* und zeigen die Verknüpfung der einzelnen Applikationen, um ein ganzheitliches Testsystem bereitzustellen. Weiters wird der dahinterliegende Aufbau und die Architektur des entwickelnden Systems dargestellt.

Bei **AWATAR** handelt es sich um eine prototypenhafte Anwendung, welche das Testkonzept anhand eines Beispiels zeigt. Es beinhaltet sämtliche wesentliche Inhalte des Konzepts. Der gesamte entwickelte Code des Prototyps kann auf Github unter der URL `https://github.com/hoedlale16/AWATAR` eingesehen werden.

4.4.1 Konzeptionelle Systemarchitektur

Grundsätzlich basiert das entwickelte Testsystem auf einem modularen Aufbau, bei dem jedes Subsystem als eigenständige Applikation agieren kann. Durch diese Architektur kann das System für spezifische Anpassungen (beispielsweise für andere Projekte oder Systemlandschaften) modular angepasst werden.

Im Zentrum des Systems steht ein Webservice, welches als **AwatarCore** bezeichnet wird. AwatarCore ist ein in JAVA implementiertes Webservices, welches HTTP-Schnittstellen für die Funktionalitäten der drei Subsysteme bereitstellt. Es beinhaltet die Geschäftslogik im Backend und agiert als zentrales Steuerungselement. Die modulare Bauweise ermöglicht ein mehrfaches Betreiben der Applikation und eine damit verbundene Lastverteilung. Der entwickelte Prototyp beinhaltet keine vollständig entwickelte Architektur zur redundanten Betreibung - eine Synchronisierung zwischen den einzelnen Services wurde im Prototypen nicht realisiert. Das Testkonzept umfasst neben dem **AwatarCore** noch im Weiterem die Submodule **AwatarCreator**, **AwatarRunner** und **AwatarResultant**. Abbildung 4.5 Softwarearchitektur von AWATR zeigt den schematischen Datenaustausch zwischen den Systemen sowie die eingesetzte Technologie für die Entwicklungen der Teilbereiche.

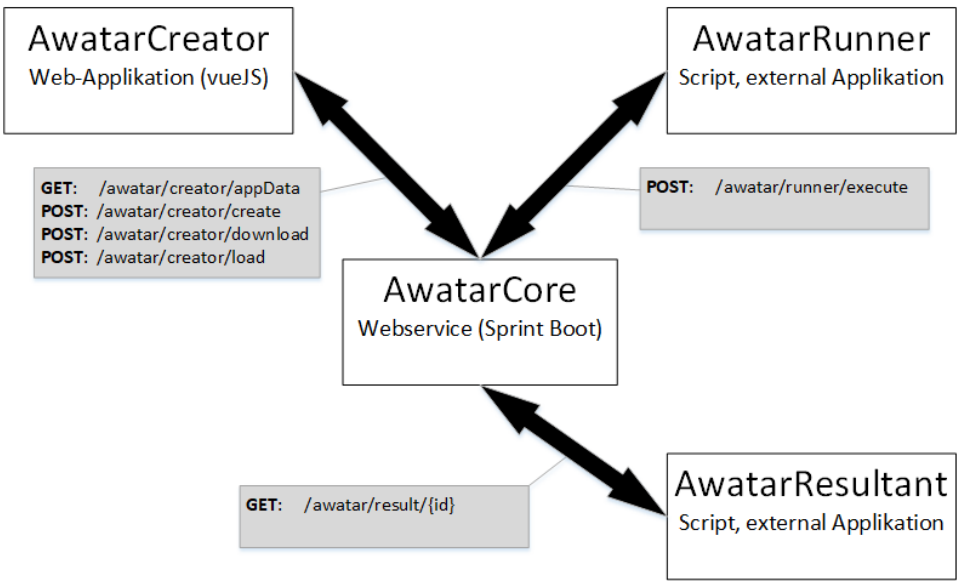

Abbildung 4.5: Softwarearchitektur von AWATR

Das Submodul **AwatarCreator** dient zur Erstellung von Testfällen in textueller Form. Dabei handelt es sich um eine Webapplikation, welche mit dem Framework von vueJS Realisiert wird. VueJS ist ein JavaScript Framework um Progressive Webapplicationen zu entwickeln (You, 2014). Der Datenaustausch mit *AwatarCore* erfolgt über HTTP-

Requests, welche Daten zwischen der Applikation und dem Service austauschen.

AwatarCore bietet neben der Schnittstelle für die Applikation *AwatarCreator* auch Schnittstellen um die Testausführung (*AwatarRunner* zu starten und die resultierenden Testergebnisse über einen HTTP-Request abzufragen (*AwatarResultant*. Die Logik der beiden Submodule befindet sich im Service von *AwatarCore*. Die Schnittstellen können also einfach von andren externen Systemen aufgerufen werden, wodurch diese Services ersetzt werden können.

Bei **AwatarRunner** und **AwatarResultant** handelt es sich jeweils um Schnittstellen von *AwatarCore*. **AwatarRunner** repräsentiert eine HTTP-Schnittstelle am Service *AwatarCore*, welche die erstellten textuellen Testfälle einliest, in ausführbaren Code transformiert und ausführt. Nach erfolgter Testausführung wird eine eindeutige ID des Testlaufs zurückgeliefert. Diese ID kann in weiterer Folge für die HTTP-Schnittstelle *AwatarResultant* genutzt werden, um die Testergebnisse in JSON-Format abzufragen. Die Trennung der beiden Funktionalitäten ermöglicht ein asynchrones aufrufen der jeweiligen Services.

Die detaillierte Dokumentation der einzelnen Submodule wird in Abschnitten 4.4.3, 4.4.4, 4.4.5 sowie 4.4.6 beschrieben.

4.4.2 Technische Systemarchitektur

Als grundlegende Programmiersprachen für das Framwork wurde **JAVA**, **JavaScript** und **TypeScript** verwendet. Die Wahl der Programmiersprachen wurde aufgrund der Verwendung in der E-Commerce Lösung von SAP (SAP Commerce) eingegrenzt und entsprechend angelehnt. Der entwickelte Prototyp wurde als eigenständige Applikation realisiert und explizit nicht in die Webshop-Applikation von SAP integriert. Dadurch kann die Entwicklung in einem eigenen Releasezyklus erfolgen und ist unabhängig von der Weiterentwicklung des Webshops bei Kundenprojekten. Durch den modularen Aufbau können die einzelnen Webapplikation via Link in die Infrastruktur des Webshops (beispielsweise SAP Commerce Backoffice) eingebunden werden, sodass für den Endanwender eine zentrale Sicht auf die beiden Systeme entsteht.

Da es sich bei dem entwickelten Prototyp bei *AWATAR Core*, *AWATAR Creator* und *AWATAR Runner* um Webapplikationen handelt, wurden für das Betreiben der Applikationen unterschiedliche Ports definiert. Dadurch kann das gesamte Framework auf einem Webserver verwaltet und betrieben werden. Um größtmögliche Netzwerksicherheit zu gewährleisten wurde darauf geachtet, dass sämtlicher Datenverkehr über die jeweiligen

Applikationsports erfolgt. Somit kann eine entsprechende Firewall-Einstellung für die jeweiligen Applikationen definiert werden.

- **Port 4000**: Die Webapplikation *AWATAR-Core* wird auf dem Port 4000 betrieben. Sämtliche Schnittstellenanfragen werden über diesen Port abgewickelt.
- **Port 4001**: Die Webanwendung zur Testfallerstellung *AWATAR-Creator* wird auf Port 4001 betrieben und kann über diesen mit der URL des Servers aufgerufen werden. Um die bereits vorhandenen Test-Methoden abzufragen wird eine Verbindung zu *AWATAR-Core* auf Port 4000 benötigt.
- **Port 4002**: Der Prototyp zur Testfallausführung *AWATAR-Runner* wird auf Port 4002 betrieben. Die Testausführung kann auch alternativ über alternative Applikationen (beispielsweise eigen entwickelte Anwendungen)erfolgen. Für die Testausführung ist eine Verbindung zu *AWATAR-Core* notwendig.

4.4.3 Systemsteuerung (AWATAR-Core)

Die zentrale Verwaltung der Applikationen und der Datenaustausch zwischen den einzelnen Submodulen erfolgt über das Modul **AWATAR-Core**. Dabei handelt es sich um eine *JAVA Spring Boot* Webapplikation, welche die Grundfunktionalitäten des Testframeworks via einer HTTP-REST Schnittstelle bereitstellt. Dazu gehören neben den Schnittstellen zur Testausführung und Abholen der Testergebnisse auch das Bereitstellen der verfügbaren Test-Methoden.

Die Tabelle 4.1 listet sämtliche verfügbare HHTP-Request des implementierten Prototyps. Mithilfe dieser Requests können Testfälle erstellt und ausgeführt werden.

Tabelle 4.1: Verfügbare HTTP-Request-Methoden von *AWATAR-Core*

HTTP-Request	Request-Typ			Beschreibung
	GET	POST	PUT	
/awatar/creator/stepDefinitions	X			Liefert sämtliche ausführbare Testmethoden in strukturierter Form nach dem BTD-Ansatz
/awatar/runner/execute		X		Triggert das Ausführen der übertragenen Testfälle in JSON Format. Das Ergebnis wird in entsprechendem JSON-Format zurückgeliefert.
/awatar/runner/execute		X		Triggert das Ausführen der übertragenne Testfälle als binäres File-Format. Das Ergebnis wird in entsprechendem JSON-Format zurückgeliefert.
/awatar/resultant/{id}	X			Liefert das Testergebnis der geforderten Testausführung. Diese Funktionalität ist im Prototyp nicht enthalten, da die Testergebnisse nicht persistiert werden.

REST HTTP-Schnittstellendefinition

In folgendem Abschnitt werden die jeweilig verfügbaren HTTP-Requests aus Tabelle 4.1 näher beschrieben. Beispiele für die Datenübertragung wurden als *POSTMAN-Collection* im Verzeichnis *resources* im Submodule **AWATAR-Core** abgelegt. Dadurch können Daten mithilfe der Applikation *POSTMAN* testweise an eine laufende Instanz von **AWATAR-Core** übertragen werden.

/awatar/creator/stepDefinition: Liefert sämtliche verfügbare Test-Methoden für die Testfallerstellung.

- **HTTP Requesttyp**: GET
- **Eingangsparameter**: Keine
- **Rückgabewerte**: Die vorhandene Test-Methoden in gruppierter Form nach *GIVEN*, *WHEN* und *THEN*. Listing 4.1 zeigt den Aufbau des zurückgemeldeten JSON-Objektes.

Listing 4.1: Aufbau des zurückgemeldente JSON für verfügbare Test-Methoden.

```
{
    "stepDefinition": {
        "WHEN": [
            {
                "methodName": "
                   user_with_username_and_password_log_into",
                "stepDefinition": "User with username {string}
                   and password {string} log into",
                "parameters": [
                    "string",
                    "string"
                ]
            },
            {
                "methodName": "open_website_with_url",
                "stepDefinition": "Open website with URL {
                   string} ",
                "parameters": [
                    "string"
                ]
```

```
18              }
19          ],
20          "GIVEN": [
21              {
22                  "methodName": "login_page_with_url_is_shown",
23                  "stepDefinition": "Login page with url {string}
                        is shown",
24                  "parameters": [
25                      "string"
26                  ]
27              }
28          ],
29          "THEN": [
30              {
31                  "methodName": "dashboard_is_shown",
32                  "stepDefinition": "Dashboard is shown",
33                  "parameters": []
34              }
35          ]
36      }
37  }
```

In jeder Sektion (GIVEN,WHEN,THEN) wird eine Liste von Objekten zurückgemeldet, welche den aufzurufenden Methodennamen (*methodName*, die textuelle Beschreibung (*stepDefinition)* sowie eine Liste von Parametern (*parameters*) mit den jeweils erwarteten Datentyp.

/awatar/runner/execute: Triggert die Testausführung.

- **HTTP Requesttyp**: POST
- **Eingangsparameter**: Testfalldefinition als JSON-Object. Listing 4.2 zeigt den aufbau des JSON-Objects. Alternativ kann ein binäres File mit dem Inhalt nach dem *GHWERKIN* Format übertragen werden. Die Struktur des Fileinhalts wird in Listing 4.3 dargestellt.
- **Rückgabewerte**: Die erzielten Testergebnisse als JSON Objekt. Listing 4.4 zeigt den Aufbau des zurückgemeldeten JSON-Objektes.

Listing 4.2: Aufbau des JSON-OBjekts zur Testausführung.

```
{
    "feature": "Login successful?",
    "scenario": "Log into Dashboard",
    "given": [
      "Login page with url \"https://www.myexample.at\" is
         shown"
    ],
    "when": [
      "User with username \"admin\" and password \"pa44w0rd\"
         log into",
      "User opens page \"Dashboard\""
    ],
    "then":[
      "Dashboard is shown"
    ]
}
```

Listing 4.3: Aufbau des Fileinhalts zur Testausführung mittels binärer Fileübertragung.

```
Feature: Login successful?
Scenario: Log into Dashboard
Given Login page with url "https://www.myexample.at" is shown
When User with username "admin" and password "pa44w0rd" log
   into
And User opens page "Dashboard"
Then Dashboard is shown
```

/awatar/resultant/{id}: Abholen der Testergebnisse.

- **HTTP Requesttyp**: GET
- **Eingangsparameter**: ID der Testausführung via URL
- **Rückgabewerte**: Die erzielten Testergebnisse als JSON Objekt. Listing 4.4 zeigt den Aufbau des zurückgemeldeten JSON-Objektes.

Listing 4.4: Aufbau des rückgemeldeten JSON-Objekts nach Testausführung

```
{
    "feature": "Login successful?",
    "scenario": "Log into Dashboard",
    "given": [
        "today is Sunday"
    ],
    "when": [
        "User with name \"admin\" and password \"1234\" log in"
    ],
    "then": [
        "Dashboard is shown"
    ],
    "exitState": 1,
    "result": "F---\n\nFailed scenarios:
    ...
    You can implement missing steps with the snippets below:
    @Given(\"today is Sunday\")
    public void today_is_sunday() {
        // Write code here that turns the phrase above into
           concrete actions
        throw new io.cucumber.java.PendingException();
    }"
}
```

Das zurückgemeldete Objekt beinhaltet den ausgeführten Testfall, das Gesamtergebnis (*exitState* sowie in der Eigenschaft *result* den detaillierten Logeintrag der Ausführung. Der Wert des Gesamtergebnisses wird als Nummer zurückgeliefert, wobei 0 als *Erfolg* und 1 als *Fehlschlag* zu interpretieren ist.

Bereitstellung verfügbarer Test-Methoden

Das Bereitstellen der verfügbaren Test-Methoden ist dahingehend wichtig, um bei der Testfallerstellung auf die bereits vorhandenen Testmethoden zurückgreifen zu können um redundante Test-Methoden zu verhindern. Weiters kann dadurch bereits in der Testfallerstellung (Awatar-Creator) das Mapping auf Test-Methoden erfolgen. In dieser Applikation können dadurch die vorhandenen Test-Methoden ausgelesen und entsprechend dargestellt werden. Dadurch kann ein direktes Mapping der textuellen Beschreibung eines Testfallschrittes sowie der ausführbaren Methode im Code erzeugt werden.

Listing 4.5: Codeausschnitt zum Zurückmelden von bereits existierenden Test-Methoden

```
1  @RequestMapping(value = "/stepDefinitions", method = RequestMethod.GET,
2                  produces = MediaType.APPLICATION_JSON_VALUE)
3  @ResponseBody
4  public Map<String, Object> getStepDefinitions() {
5    Map<String, Object> applicationData = new HashMap<>();
6
7    //Add available executable step definitions
8    Map<String, List<CucumberStepDefinitionDTO>>  stepDefs = new HashMap<>();
9    stepDefs.put("GIVEN", collectStepDefintion(Given.class.getName()));
10   stepDefs.put("WHEN", collectStepDefintion(When.class.getName()));
11   stepDefs.put("THEN", collectStepDefintion(Then.class.getName()));
12   applicationData.put("stepDefinition", stepDefs);
13
14   return applicationData;
15 }
```

Listing 4.5 zeigt die implementierte Methode für den Schnittstellenaufruf. Die Methode Reagiert auf den HTTP-Requesttyp *GET* und produziert ein JSON-Objekt als Rückgabeobjekt. Dieses Objekt beinhalet jeweils eine Liste von Methoden für die jeweiligen GHERKIN-Definitionen *GIVEN*, *WHEN* und *THEN*.

Für die Datenverarbeitung wird eine interne Methode **collectStepDefinition** aufgerufen, welche zur Laufzeit sämtliche vorhandene Klassen im package **at.campus02.itw.awatar.atwatarcore.awatarcore.cucumber** mach ausführbaren Testmethoden durchsucht, filtert und zurückmeldet. Der Code der Methode ist in Listing 4.6 ersichtlich.

Listing 4.6: Codeausschnitt zum Zurückmelden von bereits existierenden Test-Methoden

```
1  private List<CucumberStepDefinitionDTO> collectStepDefintion(final String
       className) {
2
3    final List<CucumberStepDefinitionDTO> stepDefs = new ArrayList<>();
4
5    ClassPathScanningCandidateComponentProvider scanner = new
       ClassPathScanningCandidateComponentProvider(false);
6    scanner.addIncludeFilter(new AnnotationTypeFilter(CucumberOptions.class))
       ;
7    for (BeanDefinition beanDefinition : scanner.findCandidateComponents("at.
       campus02.itw.awatar.atwatarcore.awatarcore.cucumber")){
8
9      final AnnotationMetadata controllerAnnotationMetaData = (((
         ScannedGenericBeanDefinition) beanDefinition).getMetadata());
10
11     //Show available Methods
12     final Set<MethodMetadata> methodMetadataSet =
         controllerAnnotationMetaData.getAnnotatedMethods(className);
13     if(methodMetadataSet != null && ! methodMetadataSet.isEmpty()) {
14       for(MethodMetadata m : methodMetadataSet) {
15         Map<String, Object> methodAttr = m.getAnnotationAttributes(
             className);
16         if(methodAttr != null && ! methodAttr.isEmpty()) {
17           CucumberStepDefinitionDTO sd = new CucumberStepDefinitionDTO();
18           final String description = (String)methodAttr.get("value");
19
20           sd.setMethodName(m.getMethodName());
21           sd.setStepDefinition(description);
22
23           List<String> parameters = new ArrayList<>();
24           Matcher matcher = Pattern.compile("\\{([^}]+)\\}").matcher(
               description);
25           while(matcher.find()) {
26             parameters.add(matcher.group(1));
27           }
28           sd.setParameters(parameters);
29           stepDefs.add(sd);
30         }
31       }
32     }
33   }
34   return stepDefs;
35 }
```

Die definierte Schnittstelle kann von jedem HTTP-Client aufgerufen werden und ist somit unabhängig von der entwickelten Applikation *AWATAR-Creator*. So kann das System in bereits bestehende Applikationen bei Unternehmen eingebunden werden. Abbildung 4.6 zeigt den Aufruf mit der Applikation **POSTMAN**. Diese Applikation simuliert HTTP-Requests und ermöglicht so das Testen von HTTP-Schnittstellen (Postman, o. J.).

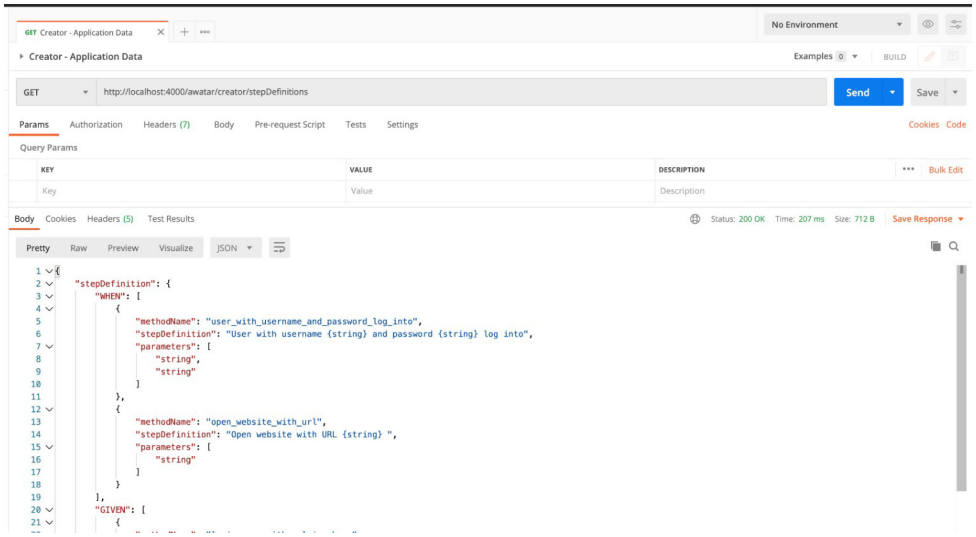

Abbildung 4.6: Aufruf der Rest-Schnittstelle für die verfügbaren Methoden der Testschritte via POSTMAN

Wie in Zeile fünf bis neun ersichtlich, werden zur Laufzeit sämtliche registrierte Klassen aus dem übergebenen Package nach der Annotation **CucumberOptions** geladen. Damit werden sämtliche ausführbare Test-Klassen identifiziert. In weiterer Folge werden von diesen Klassen sämtliche implementierte Methode gefiltert, welche mit dem übergebenen Klassennamen annotiert wurden. Dadurch können diese den jeweiligen Testschritten (GIVEN, WHEN,THEN) zugeordnet werden. Zur Veranschaulichung wird in Listing 4.7 ein Beispiel einer ausführbaren Testklasse dargestellt.

Listing 4.7: Beispiel einer Klasse mit definierten Test-Methoden

```
1  @RunWith(Cucumber.class)
2  @CucumberOptions(publish = false)
3  public class GivenDefinitions {
4
5    @Given("Login page with url {string} is shown")
6    public void login_page_with_url_is_shown(String url) {
7        WebDriver driver = new ChromeDriver();
8        driver.get(url);
9        ...
10   }
11 }
```

Wie in Zeile zwei im Listing 4.7 ersichtlich, wird die Klasse mit der Annotation *Cucumber-Option* definiert, wodurch die Klasse im zuvor beschriebenen Filtervorgang gefunden wird. In weiterer Folge wird über die Annotation *Given* (siehe Zeile fünf) die jeweilige Methode als Test-Methode definiert. Im zuvor beschriebenen Filtervorgang wird diese Annotation genutzt, um die Methode der entsprechend korrekt zuzuordnen.

Testausführung

Für die Testausführung wird über die Schnittstelle der auszuführende Testfall entweder als JSON-Objekt oder mithilfe eines Files übergeben. Der Inhalt des übergebenen Files muss dem dokumentierten GHERKIN-Format (siehe https://cucumber.io/docs/gherkin/reference/) entsprechen und wird auch in Listing 4.3 dargestellt. Das übergebene JSON-Format muss dem Format aus Listing 4.2 entsprechen und wird als initialen Schritt auf Serverseite in ein entsprechendes File zur Ausführung umgewandelt.

Listing 4.8: Code der Schnittstelle zur Testausführung

```
1  @RequestMapping(value = "execute", method = RequestMethod.POST, produces =
       MediaType.APPLICATION_JSON_VALUE)
2  @ResponseBody
3  public ResponseEntity executeTestFeature(@RequestBody
       CucumberTestExecutionDTO executionDTO) {
4      try {
5          //Create temporary feature file with given content to execute
               cucumber runtime...
6          final String featureText = parseFeatureContent(executionDTO);
7          final File featureFile = File.createTempFile("cucumber-", "-
               execution.feature");
8          FileUtils.writeStringToFile(featureFile, featureText,
9                                      Charset.defaultCharset(), false);
10         featureFile.deleteOnExit();
11
12         //Execute cucumber with given file...
13         executeCucumber(executionDTO, featureFile);
14         final HttpStatus status = (executionDTO.getExitState() != 0) ?
               HttpStatus.BAD_REQUEST : HttpStatus.OK;
15         return new ResponseEntity(executionDTO, status);
16     } catch (Exception e) {
17         return new ResponseEntity(e.getMessage(), HttpStatus.BAD_REQUEST);
18     }
19 }
```

Der Codeausschnitt in 4.8 zeigt die Verarbeitung eines übergebenen JSON-Objekts.

Initial wird das übergebene Objekt in ein File zur Ausführung umgewandelt. Dazu wird die interne Methode **parseFeatureContent** aufgerufen, welche den Inhalt des JSON-Objekts in einen validen Text für das File umwandelt. In weiterer Folge wird die Methode **exeucteCumber()** aufgerufen, welche das erzeugte File einliest und über die *Cucumber API* zur Ausführung bringt. Listing 4.9 zeigt die Umwandlung des JSON-Objekts sowie das Parsen der einzelnen Ausführungsschritte.

Listing 4.9: Code zur Umwandlung des Testfalls in JSON-Format zur Testausführung

```
1  private String parseFeatureContent(final CucumberTestExecutionDTO dto) {
2      final StringBuilder sb = new StringBuilder();
3      sb.append("Feature: ").append(dto.getFeature()).append("\n")
4        .append("Scenario: ").append(dto.getScenario()).append("\n");
5
6      buildStepDefinitions(sb, dto.getGiven(), "Given");
7      buildStepDefinitions(sb, dto.getWhen(), "When");
8      buildStepDefinitions(sb, dto.getThen(), "Then");
9
10      return sb.toString();
11 }
12
13 private void buildStepDefinitions(final StringBuilder stringBuilder, List<
       String> stepDefinitions, final String identifier ) {
14     final List<String> sdList = new ArrayList<>(stepDefinitions);
15
16     if (CollectionUtils.isNotEmpty(sdList)) {
17         final String firstStepDefinition= sdList.get(0);
18         stringBuilder.append(identifier).append(" ").append(
               firstStepDefinition).append("\n");
19         sdList.remove(0);
20
21         //If further elements available append with 'And'
22         if(! sdList.isEmpty()) {
23             sdList.forEach(sd -> stringBuilder.append("And ").append(sd).
                   append("\n") );
24         }
25     }
26 }
```

Die Methode **buildStepDefinitions** wandelt die übergebenen Testschritte der jeweiligen Testphase (GIVEN, WHEN, THEN) um. Wichtig dabei ist, dass ab dem zweiten Testschritt, diese mit dem Verbindungswort *And* angefügt werden.

Die eigentliche Testausführung wird über den Aufruf der internen Methode **execute-**

Cucumber ausgeführt (siehe Zeile 13 in Codeausschnitt 4.8). In dieser Methode wird die Testausführung über die *Cucumber API* angesteuert. Die externe API übernimmt in weiterer Folge das Auslesen des übergebenen Testfalls und Aufrufen der hinterlegten Testmethoden. Weiters werden die einzelnen implementierten Schritte ausgeführt (z.B. Öffnen einer Webseite) und der dazu gehörende Logeintrag erstellt. Diese Ausführung wird aus Performancegründen in einem eigenen Thread ausgeführt. Damit wird die Ausführung von der Schnittstelle entkoppelt, wodurch mehrere Ausführungen parallel abgearbeitet werden können. Der dazugehörige Code wird in Listing 4.10 dargestellt.

Listing 4.10: Hilfsmethode zur Testausführung

```
1  private void executeCucumber(final CucumberTestExecutionDTO executionDTO,
       final File featureFile) {
2    Runnable task = () -> {
3      // Redirect System.OUT of Cucumber Execution to response
4      ByteArrayOutputStream buffer = new ByteArrayOutputStream();
5      System.setOut(new PrintStream(buffer));
6
7      //Execute Cucumber runtime
8      String[] args = { featureFile.getAbsolutePath(), "--glue", "at.campus02
           .itw.awatar.atwatarcore.awatarcore.cucumber" };
9      final int exitState = io.cucumber.core.cli.Main.run(args, Thread.
           currentThread().getContextClassLoader());
10     System.setOut(new PrintStream(new FileOutputStream(FileDescriptor.out))
           );
11
12     // Use captured content for Response
13     final String executionResult = buffer.toString();
14     final String executionResultNoColor = executionResult.replaceAll("\
           u001B\\[[;\\d]*m", "");
15
16     executionDTO.setResult(executionResultNoColor);
17     executionDTO.setExitState(exitState);
18     buffer.reset();
19   };
20
21   //Execute Cucumber Runtime in seperate thread...
22   task.run();
23   Thread thread = new Thread(task);
24   thread.start();
25 }
```

Die Testausführung erfolgt über den Aufruf der *Main-Klasse* der API. Diese kann auch

von externen Applikationen über die Commandozeile aufgerufen werden, wodurch die Testausführung nicht zwingend über die REST-Schnittstelle von **AWATAR-CORE** erfolgen muss. Bei der Testausführung fungiert AWATAR-CORE also lediglich als Interface zur Testausführung via einer REST-Schnittstelle. Die Schnittstelle sammelt die ausgeführten Testergebnisse und liefert diese entsprechend zurück.

Im Abschnitt Abholen der Testergebnisse wird ein alternatives Konzept zur Ausführung dargelegt. Dieses Konzept beschreibt einen theoretischen Ansatz, wie eine Testausführung komplett losgelöst von der Auswertung des Testergebnisses erfolgen kann. Im Zuge des entwickelten Prototypen wurde dieser Ansatz nicht realisiert.

Abholen der Testergebnisse

Diese Funktionalität wurde im Prototypen nicht vollständig realisiert, da keine Datenbank für die Persistierung der Testergebnisse vorliegt. Die Schnittstelle wurde initial erstellt, verfügt jedoch über nicht funktionierenden Code (aufgrund der fehlenden Anbindung einer Datenbank). Im folgenden Abschnitt wird die konzeptionelle Umsetzung für diese Funktionalität beschrieben.

Um die Testausführung und das Abholen der Testergebnisse zu entkoppeln, muss die Methode **executeTestFeature** aus dem Listing 4.8 dahingehend umgebaut werden, dass das zurückgemeldete Ergebnis der Methode **executeCucumber** in einer Datenbank gespeichert wird. Für jede Testausführung muss ein eindeutiger Schlüssel (Identifier) definiert werden, um die Testergebnisse in weiterer Folge auslesen zu können. Dieser Identifier wird bei der Testausführung anstelle des Testergebnisses zurückgeliefert.

Wird die Schnittstelle **/awatar/resultant/{id}** aufgerufen, kann das gespeicherte Testergebnis für den übergebenen Identifier ausgelesen und zurückgemeldet werden. Dadurch können zwei unterschiedliche Systeme die Testausführung bzw. die Verwaltung der Testergebnisse angesteuert werden. Abbildung 4.7 zeigt, wie die Rückmeldung des Testergebnisses aussehen könnte. Die Abbildung zeigt das zurückgelieferte Ergebnis bei direkter Testausführung.

Die Persistierung der Testergebnisse kann entweder auf Datenbankebene oder direkt am Filesystem des Servers erfolgen. Wichtig bei der Sicherung ist die eindeutige Identifikation und Zuordnung der Testergebnisse zur Testausführung. Da der gewählte Identifier eindeutig sein muss, kann dieser bei einer Datenbanklösung auch als **PRIMARY KEY** gewählt werden.

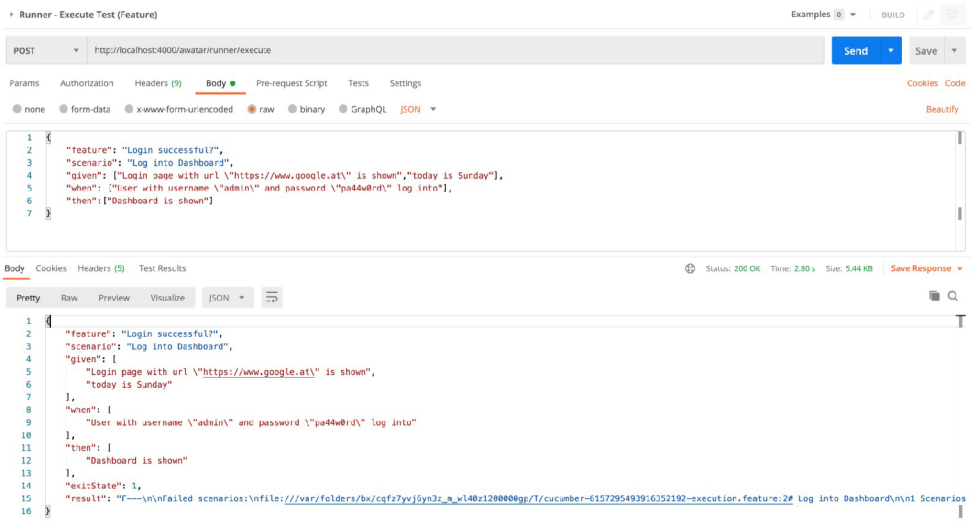

Abbildung 4.7: Aufruf der Rest-Schnittstelle zur Testausführung mit direkter Ergebnisrückmeldung

4.4.4 Erstellen automatisierter Tests (AWATAR-Creator)

Die Applikation **AWATAR-Creator** wurde als Webapplication mithilfe des Frameworks *vueJS* realisiert und dient zur Erstellung von automatisierten Tests in textueller Form. Für die eigenständige die Webapplikation wird der Port **4001** für den eingebundenen Webserver verwendet. Die Web-Applikation kann jedoch auch in bestehende Webapplikationen eingebunden werden, wodurch der Zugriff über diese erfolgt.

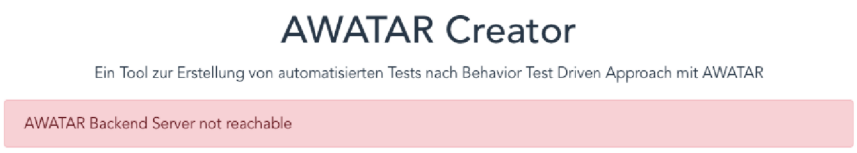

Abbildung 4.8: Aufruf der Rest-Schnittstelle zur Testausführung mit direkter Ergebnisrückmeldung

Beim Start der Applikation (Erster initialer Seitenaufruf) werden die bereits vorhandenen Definitionen für Testmethoden aus dem **AWATAR-Core** geladen. Dadurch soll verhindert werden, dass Methoden doppelt erstellt und weiters dem Benutzer eine Vorauswahl an bereits implementierten Methoden zur Verfügung gestellt werden. Listing 4.11 zeigt die Implementierung des Schnittstellenaufrufs, sowie das Handling im Fehlerfall. Ist

eine Verbindung zum Backend (AWATAR-Core) nicht möglich, wird eine entsprechende Fehlermeldung angezeigt (siehe Abbildung 4.8

Listing 4.11: Hilfsmethode zum Laden der verfügbaren Testmethoden in AWATAR-Creator

```
initializeAppData () {
  this.showLoadingScreen = true
  RequestService.webRequest(RequestService.contextPath + '/creator/
      stepDefinitions')
  .then(function (res) {
    this.showLoadingScreen = false
    this.applicationData = res.data
    if(this.applicationData) {
      this.availableStepDefinitions = (this.applicationData.stepDefinition)
          ? this.applicationData.stepDefinition : []
    } else {
      this.noBackendAvailable = true
    }
  }.bind(this))
  .catch(function (error) {
    this.showLoadingScreen = false
    this.noBackendAvailable = true
    if (error.response) {
      const data = JSON.parse(error.response.data)
      console.log(data.errorMsg)
    }else if (error.request) {
      console.log(error.request);
    } else {
      console.log(error.message);
    }
  }.bind(this))
}
```

Konnten die Daten geladen werden, wird eine Oberfläche zur Erfassung eines neuen Testfalls angezeigt. Für jeden Testfall ist ein Titel (Funktion), sowie eine Szenariobeschreibung zwingend erforderlich. Weiters können über eine Liste die Vorbedingungen (GIVEN), die Aktionen (WHEN) und die erwarteten Ergebnisse (THEN) erfasst werden. Abbildung 4.9 zeigt die Oberfläche mit bereits erfassten Beispieldaten.

Die Applikation bietet die Möglichkeit einen bereits gespeicherten Testfall zu laden, um diesen zu bearbeiten oder den erstellten Testfall in einem File zu speichern. Die Zuordnung einer Aktion (Vorbedingung, Aktion oder erwartetes Ergebnis) erfolgt über

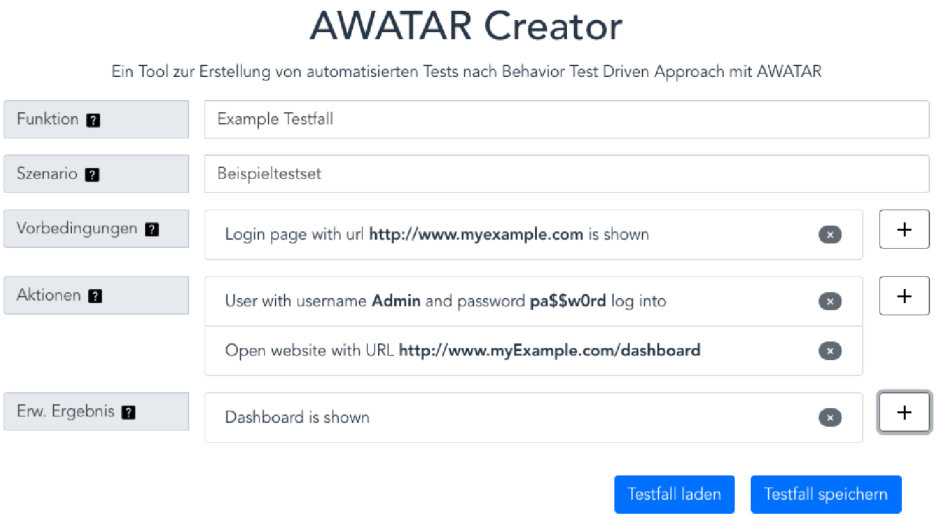

Abbildung 4.9: Aufruf der Rest-Schnittstelle zur Testausführung mit direkter Ergebnisrückmeldung

einen eigenen Dialog. Dieser listet die verfügbaren ausführbaren Aktionen je nach Gruppe in einem Dropdrown-Menü auf. Je nach Auswahl der Aktion werden die dafür benötigen Parameter mit dem jeweils erwarteten Datentyp angezeigt (siehe Abbildungen 4.10 und 4.11).

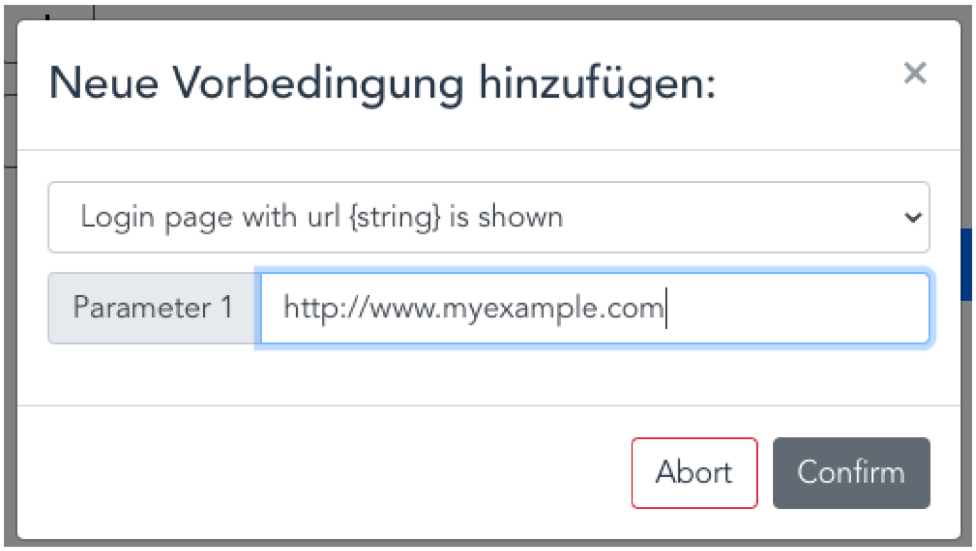

Abbildung 4.10: Auswahl einer Testmethode mit einem Parameter

Die eingegebenen Parameter werden in der Übersicht in **fett** dargestellt, um diese entsprechend hervorzuheben. Bereits zugeordnete Aktionen können aus der Liste wieder entfernt werden, ein Bearbeiten von bereits zugeordneten Aktionen ist nicht möglich. Nachdem der Testfall erstellt wurde, kann dieser über den Button *Testfall speichern* gespeichert werden. Im Zuge des Speichervorgangs werden die Daten in das GHERKIN-

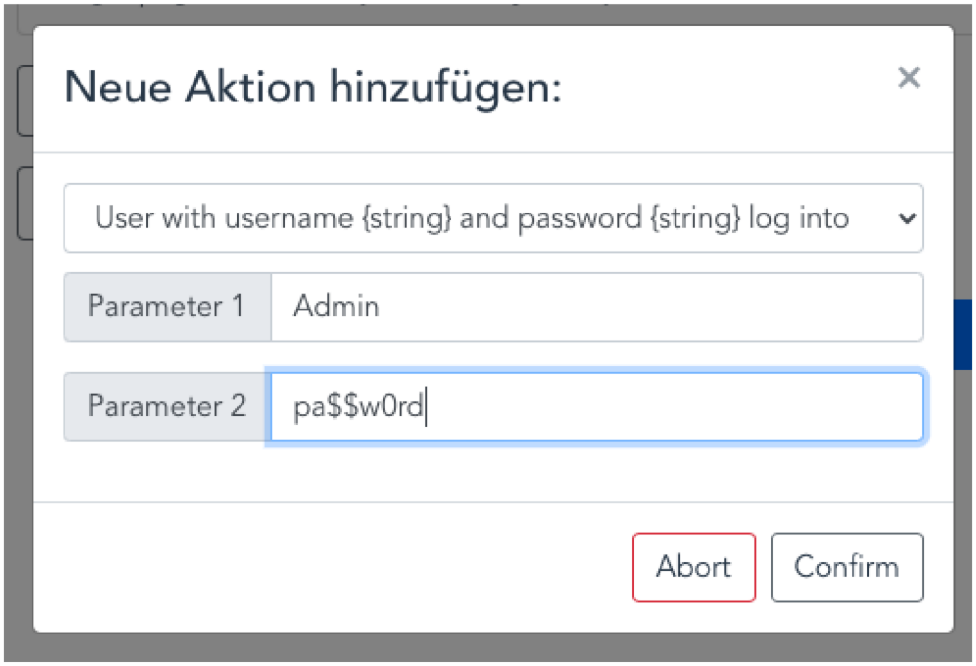

Abbildung 4.11: Auswahl einer Testmethode mit mehreren Eingabeparametern

Format (siehe https://cucumber.io/docs/gherkin/reference/) umgewandelt und entsprechend abgespeichert. Danach wird eine entsprechende Rückmeldung angezeigt (siehe Abbildung 4.12).

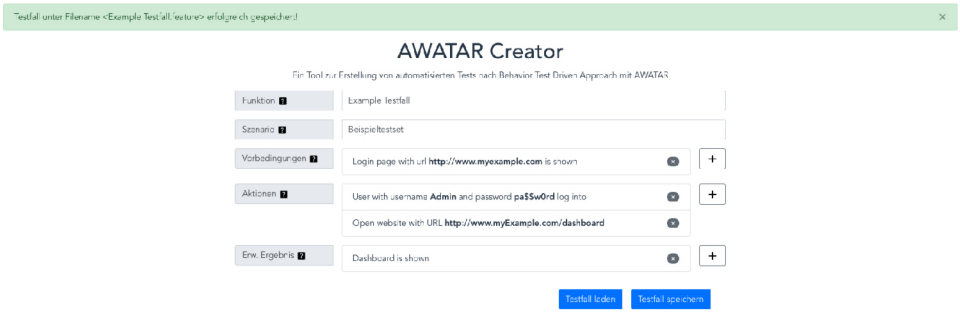

Abbildung 4.12: Speichern eines Testfalls in Awatar-Creator

Der Prototyp könnte noch um weitere Funktionalitäten erweitert werden, sodass Testfälle in ein Testmanagementsystem (beispielsweise Testlink oder JIRA) exportiert werden können. Diese Funktionalitäten wurden im Zuge der Prototypentwicklung aus Zeitgründen nicht realisiert.

Zur Bearbeitung von bestehenden Testfällen kann über den Button *Testfall laden* ein Testfall geladen werden. Der Testfall muss dafür in dem GHERKIN-Format vorliegen und wird im Zuge des Einleseprozesses entsprechend validiert. Der dafür implementierte Lade-Dialog zeigt den geparsten Inhalt als Vorschau, bevor die Daten in die Appli-

kation übernommen werden. Abbildung 4.13 zeigt den Dialog und die Vorschau des Dateiinhalts.

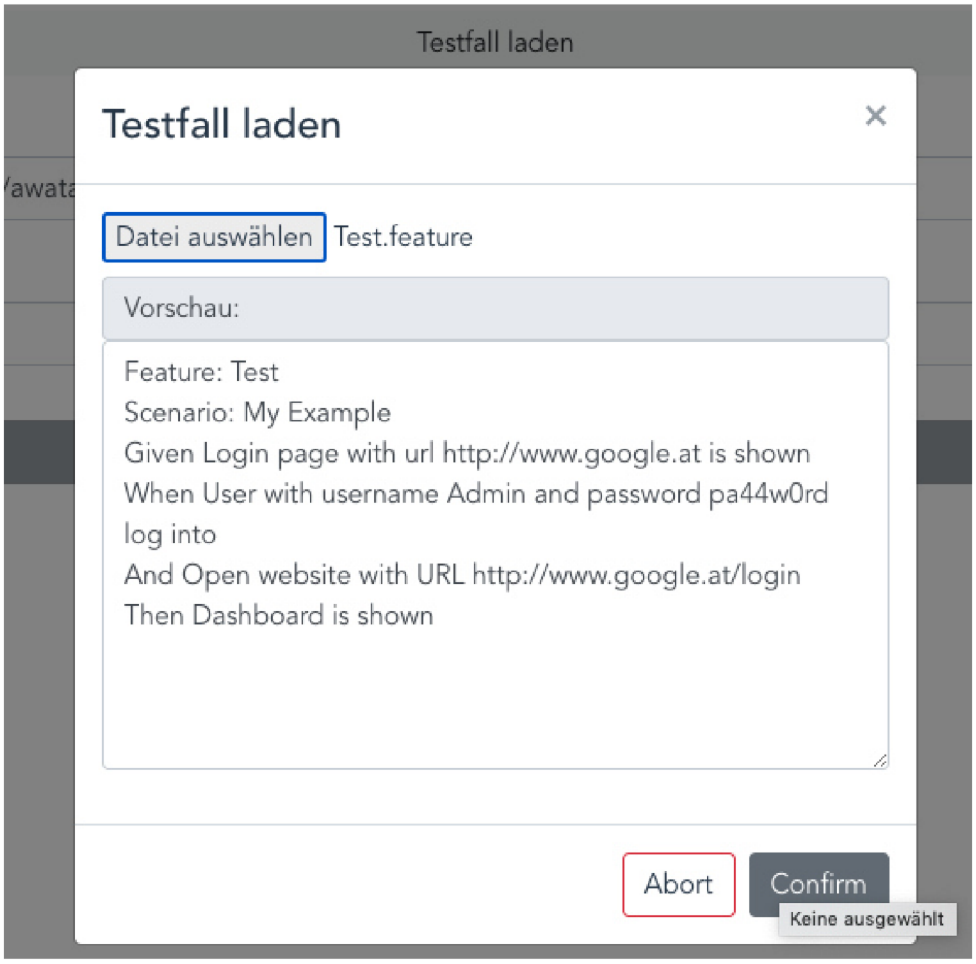

Abbildung 4.13: Laden eines Testfalls in Awatar-Creator

4.4.5 Testausführung (AWATAR-Runner)

Die Applikation AWATAR-Runner ist eine prototypenhafte Webapplikation, die eine Testausführung ermöglicht und das Ergebnis der Testausführung entsprechend anzeigt. Die Applikation nutzt die REST-Schnittstelle von *AWATAR-Core* zur Testausführung und zeigt auf der Oberfläche dem Benutzer den Aufbau abgesendeten des HTTP-Requests an. In Abbildung 4.14 wird die Testausführung eines erfolgreich ausgeführten Testfalls dargestellt.

Tritt während der Testausführung ein Fehler auf, wird der Log des Testfalls in Rot hinterlegt und eine entsprechende Fehlermeldung ausgegeben (siehe Abbildung 4.15).

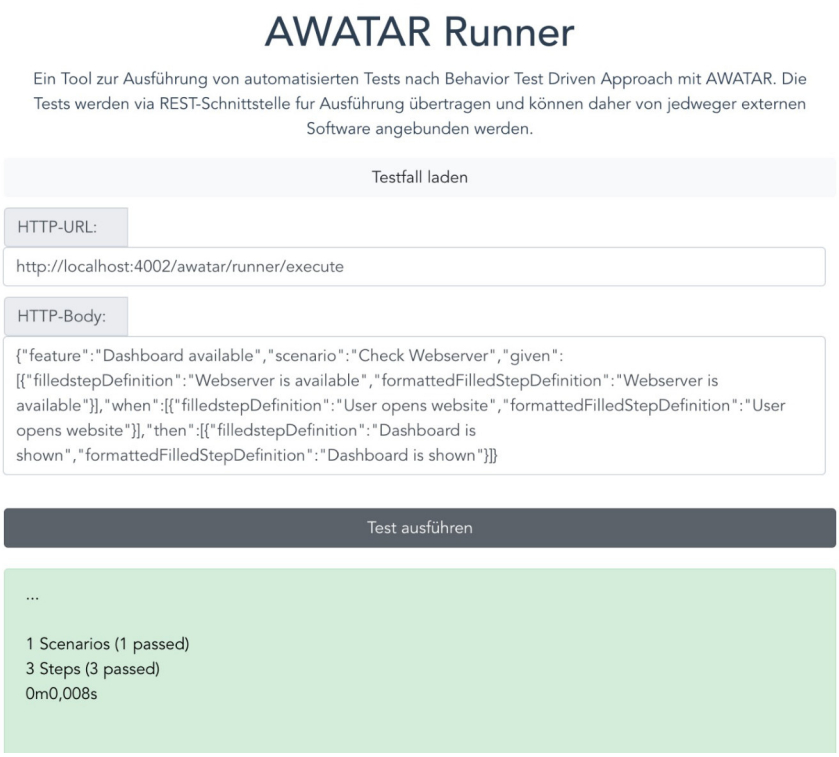

Abbildung 4.14: Testausführung mittels der Webapplikation Awatar-Runner

Die Testausführung erfolgt über das Backend *AWATAR-Core*, welche zur Ausführung die API **Cucumber Open** aufruft. Wie bereits in Abschnitt 4.4.3 erläutert, kann die Testausführung auch ohne der Webapplikation erfolgen. Die Webapplikation dient lediglich als Schnittstelle, um die Testausführung via HTTP-Requests anzustoßen. Dadurch kann die Testausführung in bestehende Webanwendungen einfach integriert werden.

Die Webapplikation *AWATAR-Runner* validiert vor der Ausführung des Testfalls diesen auf syntaktische Korrektheit. Danach wird die eingelesene Datenstruktur in das JSON-Format für die Schnittstelle umgewandelt und an das Backend (*AWATAR-Core*) gesendet. Das dafür notwendige Format ist in Unterabschnitt REST HTTP-Schnittstellendefinition des Abschnitts 4.4.3 Systemsteuerung (AWATAR-Core) beschrieben.

Der Prototyp könnte hierbei noch um die Funktionalität der Persistierung der Ergebnisse oder das Weiterreichen an weitere Systeme erweitert werden. Beispielsweise kann das zurückgemeldete Ergebnis an das Testmanagementsystem zur Dokumentation und weiterer Verarbeitung (beispielsweise das Antriggern von E-Mailbenachrichtigungen) weitergereicht werden. Diese Funktionalitäten wurden im Zuge der Prototypenentwicklung nicht realisiert.

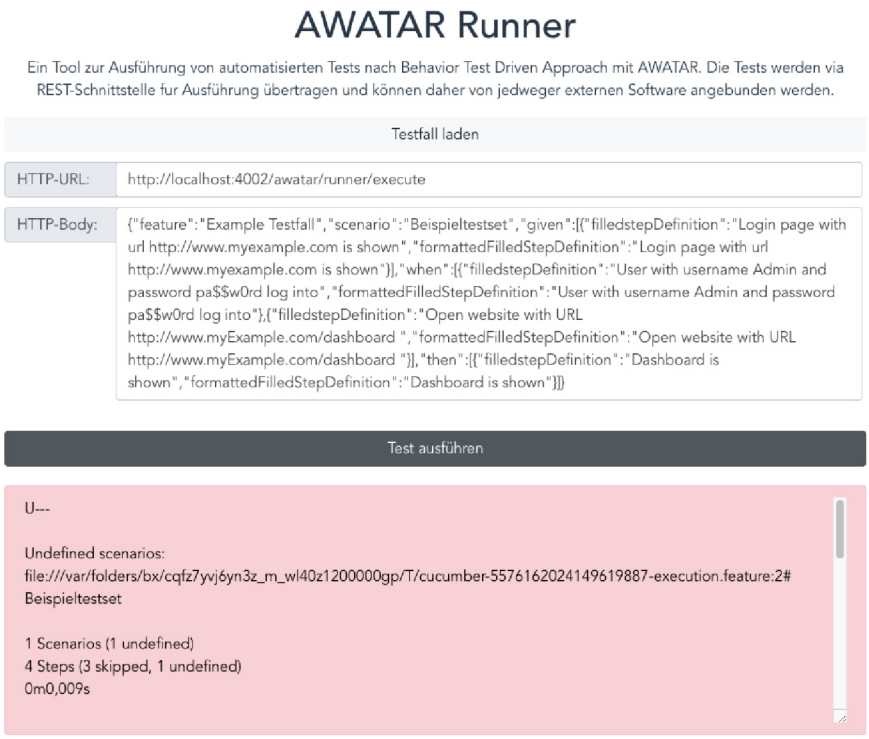

Abbildung 4.15: Fehler während der Ausführung eines Testfalls mittels Awatar-Runner

Listing 4.12: Code zur Ausführung eines Testfalls mittels AWATAR-Runner

```
executeTest() {
  if (this.validTestFeature) {
    this.showLoadingScreen = true
    let executableTestFeature = this.buildExecutableTest()
    RequestService.postRequest.post(this.executeUrl, executableTestFeature)
    .then(function (res) {
      var respone = res.data
      if (respone) {
        this.infoBox = {
          msg: respone.result,
          closable: true,
          type: (respone.exitState == 0) ? 'success' : 'error'
        }
        this.showTestExecutionInfoBox = true
      }
      this.showLoadingScreen = false
    }.bind(this))
    .catch(function (error) {
      this.showLoadingScreen = false
      ...
    }.bind(this))
 }
}
```

In Listing 4.12 wird der dafür implementierte Code dargestellt. Beim zurückgemeldeten Ergebnis wird initial der *exitState* ausgelesen. Entspricht dieser 0 wird der Log in einer grünen Box dargestellt. Ansonsten wird die Rückmeldung als fehlerhaft interpretiert und das Ergebnis in einer rot hinterlegten Informationsbox dargestellt.

4.4.6 Reporting der Testergebnisse (AWATAR-Resultant)

Im Zuge der Entwicklung des Prototyps wurde das Abholen und Darstellen der Testergebnisse mit der Applikation zur Testausführung zusammengelegt. Diese dazu entwickelte Webapplikation wird in Abschnitt 4.4.5 Testausführung (AWATAR-Runner) näher erläutert. Wie bereits in Unterabschnitt Abholen der Testergebnisse des Teilbereichs 4.4.3 beschrieben, kann die Trennung von Testausführung und Verarbeitung der Testergebnisse nur erfolgen, wenn die Ausführungen in einer Datenbank oder direkt am Filesystem permanent gespeichert werden. Dadurch kann über einen asynchronen Prozess auf die Ergebnisse zur weiteren Verarbeitung zugegriffen werden. Der vorliegende Prototyp beinhaltet die vorbereitete Schnittstelle, um Testdaten anhand eines eindeutigen Identifieres abzuholen. Eine Persistenz der Testergebnisse wurde jedoch nicht realisiert.

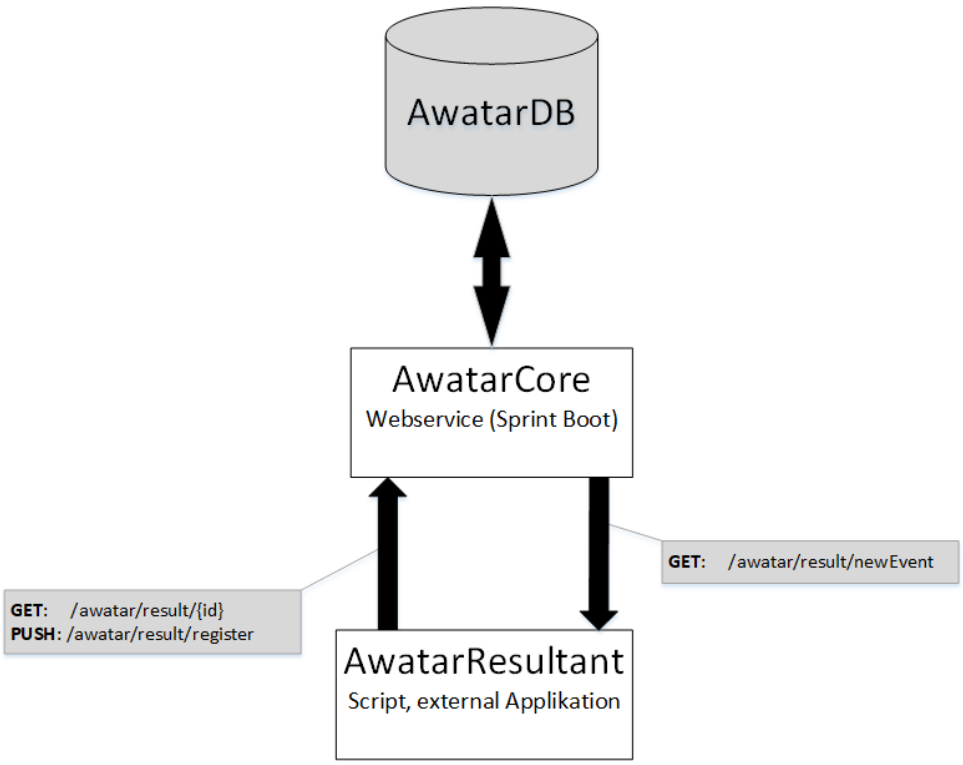

Abbildung 4.16: Konzept zur asynchronen Implementierung der Ergebnisverarbeitung (Awatar-Resultant)

Wird diese asynchrone Schnittstelle realisiert, muss sichergestellt werden, dass auch ein Mechanismus zur Benachrichtigung neuer Ergebnisse implementiert wird. Dieser Benachrichtigungsprozess kann mithilfe eines **Public/Subscribe-Mechanismus** erreicht werden. Interessierte Anwendungen können sich auf neue Ereignisse registrieren und werden bei Vorliegen eines neuen Testergebnisses benachrichtigt. Dadurch muss von den interessierten Anwendungen kein permanentes Abfragen (Polling) erfolgen. Mithilfe dieses **Public/Subscribe-Mechanismus** können externe Applikationen wie Testmanagementsysteme, Bugtracking-Systeme oder Dashboards angebunden werden. Diese können die Testergebnisse entsprechend nach Benachrichtigung abholen und für den Endanwender aufbereiten.

Abbildung 4.16 zeigt das Konzept zur Registrierung von externen Interessenten. Über die Http-Schnittstelle **/awatar/result/register** können sich Anwendungen für neue Events registrieren. Für die Verarbeitung neuer Events muss die Anwendung eine Schnittstelle implementieren, welche HTTP-Anfragen des Types **GET** verarbeitet. Auf diesem überträgt *AWATAR-Core* die Informationen, welche neuen Testergebnisse abgeholt werden können. Diese können dann über die vorgesehene Schnittstelle **/awatar/result/{id}** abgeholt und verarbeitet werden.

Awatar-Core muss für jede registrierte Applikation die Verbindungsdaten abspeichern, um bei neuen Testergebnissen diese benachrichtigen zu können. Bei jeder Testausführung werden die Ergebnisse in einer Datenbank abgelegt und die Interessenten ausgelesen. Diese Interessenten werden über das neu abgelegte Testergebnis informiert. Eine direkte Übertragung des Testergebnisses erscheint nicht sinnvoll, da zum Zeitpunkt der Erstellung nicht bekannt ist, ob die interessierten Systeme für eine Informationsverarbeitung bereitstehen. Entsprechend fungiert die Datenbank als Warteschlange der Testergebnisse. Um bei einer etwaigen Datenunterbrechung keine Daten zu verlieren, müssen für die Schnittstellen eigene Warteschlangen realisiert werden. Dadurch können die einzelnen Systeme in ihrer Datenverarbeitung entkoppelt werden, ohne Daten zu verlieren.

5 Konzeptvalidierung

Das in Kapitel 4 Konzeptentwicklung beschriebene Konzept, wurde im Zuge eines Referenzprojektes der Firma B4B Solutions GmbH für ein E-Commerce System prototypisch realisiert und damit einem entsprechenden Praxistest unterzogen. Folgendes Kapitel beschreibt die daraus resultierenden Ergebnisse und Erfahrungen. Die erzielten Ergebnisse beziehen sich stark auf das E-Commerce System von SAP mit der Bezeichnung SAP Commerce 19.11, welches in der OnPremise-Variante bei einem Kundenprojekt eingesetzt wurde.

Das entwickelte Testkonzept wurde als eigenständige Webapplikation entworfen, um einen unabhängigen Releasezyklus zum Hauptprojekt realisieren zu können. Weiters beinhaltet der vorgestellte Prototyp nicht alle realisierten Funktionalitäten des eingesetzten Projektes. Dies betrifft hauptsächlich kundenspezifische Entwicklungsschritte sowie die entwickelten Tests der Prozessabläufe. Der Prototyp beinhaltet jedoch sämtliche Grundfunktionalitäten, um ein Testkonzept für ein ähnliches Referenzprojekt realisieren zu können.

5.1 Kennzahlen zur Validierung

Im Zuge des eingesetzten Referenzprojektes wurden zu Beginn Kennzahlen zur Messung der Benutzbarkeit und Effizienzsteigerung mithilfe des entworfenen Testkonzepts festgelegt. Basierend auf diesen erhobenen Erfahrungswerten soll eine Aussage getroffen werden, inwieweit die erstellten automatisierten Tests zur Qualitätssteigerung der Webanwendung beigetragen haben. Die Kennzahlen sind insofern wichtig, da in der Definition Qualitätskriterien eine große subjektive Komponente einfließt. So können etwa Endanwender eine differenziertere Sichtweise auf eine Anwendung haben, als die Entwickler oder der Projektmanager.

Da keine Daten aus Referenzprojekten als Vergleichswerte herangezogen werden können, wurde als wichtigste Kennzahl die Anzahl der durchgeführten Tests und der hiervon die Anzahl prozentual erfolgreich durchgeführter Tests im Testlauf als Kennzahl definiert. Die untenstehende Tabelle (siehe 5.1) zeigt die im Zeitraum 1. Oktober 2020

Tabelle 5.1: Übersicht durchgeführter Testfälle je Sprint (Zeitraum Oktober bis November 2020)

Sprint	Anzahl durchgeführter Tests	Erfolgreich durchgeführt	Nicht Erfolgreich durchgeführt
06.10. - 20.10.	3	**1 (33,3 %)**	2 (66,6 %)
20.10. - 03.11.	5	**3 (60,0 %)**	2 (40,0 %)
03.11. - 17.11.	5	**4 (80,0 %)**	1 (20,0 %)
17.11. - 24.11.	6	**4 (66,6 %)**	2 (33,3 %)

bis 31. November 2020 durchgeführten vollständigen Sprints. Weiters wird für jeden Sprint die Anzahl der Testfälle, sowie der Prozentsatz der erfolgreich durchgeführten Tests dargestellt.

Im Verlauf der Testausführung konnte neben der Steigerung der durchgeführten Tests auch ein positiver Trend der erfolgreich ausgeführten Testungen erkannt werden. Die Erstellung von automatisierten Tests konnte durch die Wiederverwendbarkeit von Testmethoden durch die Webanwendung *AWATAR-Creator* vereinfacht werden und kann auch von nicht technikaffinen Personen durchgeführt werden. Die Komplexität der automatisierten Tests liegt jedoch weiterhin in der Entwicklung der Testmethoden und der Integration einzelnen Prozessschritte.

Um die Entwicklungszeit der Testfälle in Summe in einem akzeptablen Bereich zu halten, müssen die Testszenarien weiterhin vorab klar durchdacht und strukturiert werden. Dabei kann die Webanwendung **Awatar Creator** unterstützen, sofern die einzelnen Testmethoden atomar genug realisiert wurden und wiederverwendet wer den können. Nichtsdestotrotz sollten automatisierte Tests lediglich für wiederkehrende Prozessabläufe erstellt werden und nicht zur Validierung sämtlicher Grundfunktionalität verwendet werden. Manuelle Tests können im Vergleich zu automatisierten Tests viel schneller umgesetzt werden, wodurch der Einsatz des automatisierten Testframeworks wohl überlegt sein will.

Eine weitere Erfahrung aus dem praktischen Referenzprojekt liegt in der vorhandenen Datenbasis von Testmethoden. Je mehr atomar erstellte Testmethoden zur Verfügung stehen, desto einfacher können neue Testfälle abgebildet werden. Hierbei ist jedoch bei der Entwicklung darauf zu achten, dass Prüfungen (beispielsweise ob ein Element angezeigt wird oder nicht) immer über den selben Identifier angesprochen werden können. Bei Webanwendungen lassen sich diese Abfragen gut über speziell definierte Klassennamen via CSS lösen. Über direkt vergebene Element-Ids können einzelne Elemente zwar gut identifiziert, aber diese im Code nicht öfter wiederverwendet werden.

Dadurch schränkt diese Art der Identifizierung ein.

5.2 Integration in das Softwareprojekt

Durch die Realisierung des Testkonzepts als eigenständiges System erfolgt die Integration des Testkonzepts in ein bestehendes Softwareprojekt ohne großen zeitlichen Aufwand. Hierbei muss bei der Installation lediglich auf die Vergabe der freien Ports sowie das Herstellen der Infrastruktur (Webserver) geachtet werden.

Die größte Herausforderung bei der initialen Erstellung der Infrastruktur liegt in der korrekt verwendeten Versionierung der einzelnen benötigten Komponenten (beispielsweise der ChromeDriver für die Cucumber API). Da für das SAP Commerce Projekt bereits ein Webserver bereitgestellt wurde, konnte dieser um das Testkonzept entsprechend erweitert werden. Durch die unterschiedliche Nutzung der Port-Bereiche konnte auch ein Überschneiden dieser verhindert werden.

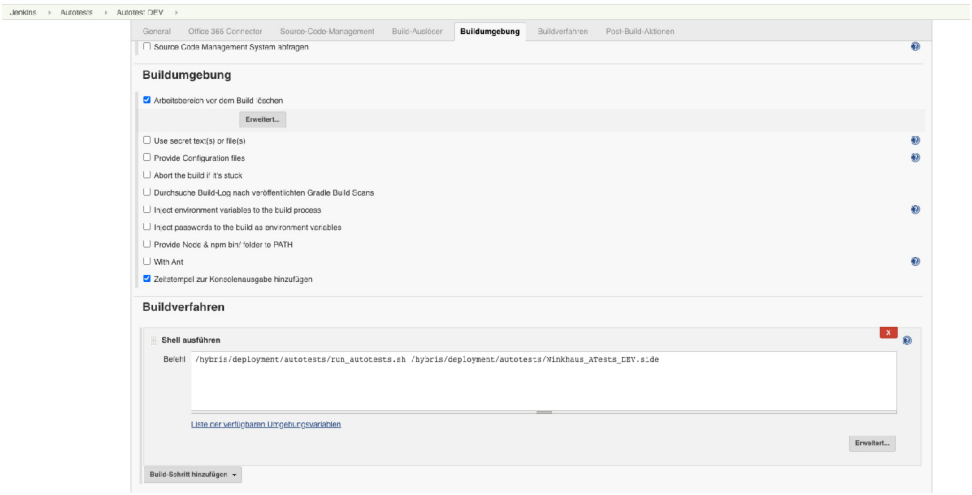

Abbildung 5.1: Beispielhafte Einbindung der automatischen Testausführung in das Tool **Jenkins**.

Neben der Erstellung der Infrastruktur wurde auch eine mögliche Integration des entwickelten Prototyps mit vorhandenen Entwicklungstools (beispielsweise JIRA, Jenkins oder GitHub) geprüft. Durch die flexible REST-Schnittstelle konnte ein automatischer Start der Testausführung via Jenkins implementiert werden. Dafür wurde ein Bash-Script erstellt, welche die Main-Klasse der Cucumber-API aufruft. Die technische Dokumentation für diese Implementierung wird in Teilabschnitt Testausführung des Kapitels 4.4.3 beschrieben. Die erstellten Testfälle wurden in weiterer Folge mithilfe von Git versioniert, wodurch Änderungen an diesen nachvollziehbar dokumentiert werden konnten.

Zusätzlich wurde sichergestellt, dass zur Ausführung immer die aktuellste Version der Testfälle verwendet wurde.

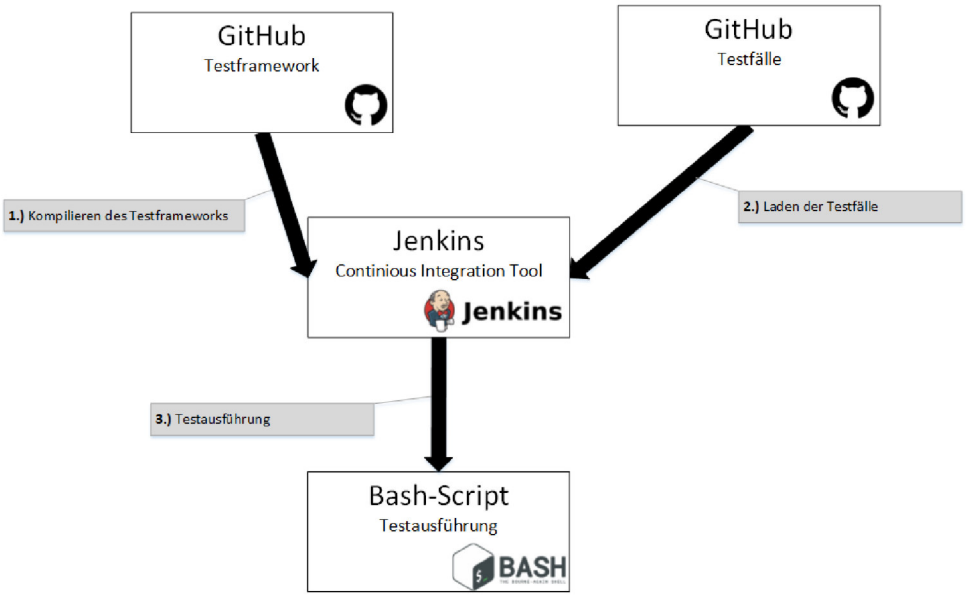

Abbildung 5.2: Schematische Darstellung der Integration des Testkonzept in die bestehende Softwarearchitektur

Die Integration des Testkonzepts in die bestehende Infrastruktur des Softwareprojekts wird in Abbildung 5.2 dargestellt. Bei jeder Ausführung werden initial die aktuellste Version der Sourcen des Testframeworks geladen und neu kompiliert. Danach folgt das Laden der aktuellsten Testfälle aus dem GitHub-Repository. Erst nachdem die beiden Schritte erfolgreich ausgeführt wurden, erfolgt die eigentliche Testausführung mittels eines Bash-Scripts. Das Logging sowie die Benachrichtigung (Erfolgreich oder nicht) erfolgt über das Jenkins-Framework. Abbildung 5.1 zeigt die beispielhafte Konfiguration zur Einbindung in Jenkins. Weiters wird in Abbildung 5.3 ein Logauszug einer Testausführung im Referenzprojekt dargestellt.

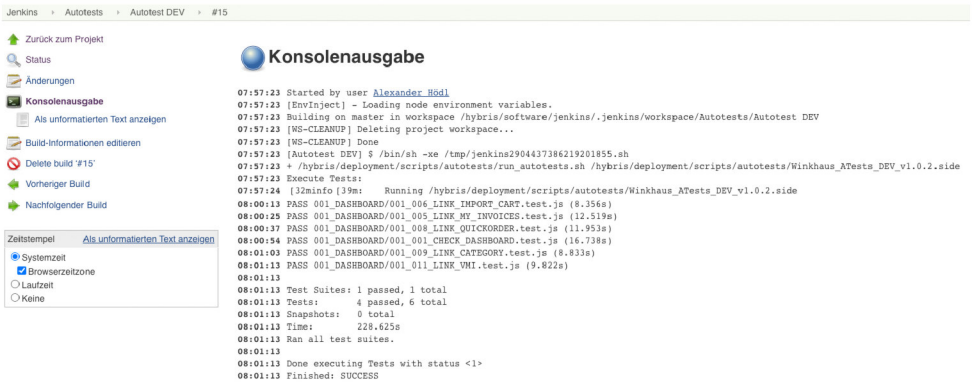

Abbildung 5.3: Beispielhafter Logauszug einer Testausführung via das Tool **Jenkins**.

Eine Integration in ein Testmanagementtool wurde nicht durchgeführt, da im Zuge des Kundenprojekts kein entsprechendes Tooling eingesetzt wurde. Diese Integration würde ein automatisiertes Testreporting ermöglichen, indem definierte Testfälle mithilfe des Testframeworks auf *erfolgreich durchgeführt* oder *nicht erfolgreich durchgeführt* gesetzt werden könnten. Für diese Implementierung wird jedoch die, in Kapitel 4.4.3 beschriebene Schnittstelle für das Abholen von Testergebnissen benötigt.

5.3 Erstellung automatisierter Tests

Die Erstellung der automatisierten Tests konnte mithilfe der Webapplikation **AWATAR-Creator** im Zuge der Sprintplanung durch Mitarbeiter aus dem jeweiligen Fachbereich durchgeführt werden. Dadurch konnte die Testfalldefinition an Personen weitergegeben werden, welche die Prozessabläufe gut kennen und entsprechende Testfälle konzipieren konnten.

Die größte Herausforderung hierbei war die Zurverfügungstellung von ausführbaren Testmethoden, sodass die einzelnen Testschritte vorab bereits abgebildet werden konnten. Nur durch eine große Palette atomar ausführbarer Testmethoden können nicht technisch affine Personen Prozessabläufe des Arbeitsalltags gut beschreiben und dokumentieren. Hierfür ist auch eine kurze Einschulung in die Konzepte von *Behavior Test Driven* und *Test Driven Development* notwendig, um die Notwendigkeit und die Herangehensweise bei der Testfallerstellung zu erläutern.

Im Zuge des Referenzprojektes wurde nur eine geringe Anzahl an Testmethoden im Zuge des Pilotprojektes erstellt, wodurch keine komplexen Prozessabläufe abgebildet werden konnten. Beispielsweise wurde mithilfe der automatisierten Tests das Navigationsverhalten auf den jeweiligen Webapplikationen der Fachbereiche geprüft. Eine vollständige Abbildung eines Bestellprozesses konnte jedoch aufgrund der fehlenden Testmethoden nicht abgebildet werden (Von der Produktsuche über die Produktauswahl bis hin zur Ablage im Warenkorb und Durchführung der Bestellung mit Interaktion des ERP-Systems).

Nichtsdestotrotz konnten positive Rückmeldungen durch die Fachbereiche identifiziert werden. Durch die frühe Einbindung der Fachbereiche in den Entwicklungsprozess konnte neben einer höheren Akzeptanz der entwickelten Softwarebausteine im Fachbereich auch eine genauere Spezifikation von Abläufen erzielt werden. Weiters wurde durch die Testfalldefinition in den jeweiligen Fachbereichen bereits eine indirekte Vorabschulung

für die neu entwickelten Abläufe im System erreicht. Dadurch konnte ein schnelleres Feedback im Entwicklungsprojekt einfließen. Als negative Rückmeldung musste der dafür notwendige Zeitaufwand im Fachbereich dokumentiert werden. Durch das Verschieben der Testfallerstellung in die Fachbereiche werden zusätzliche Tätigkeiten zum Tagesgeschäft in den Bereichen generiert, wodurch eine erhöhte Mehrbelastung der Mitarbeiter entsteht.

Durch die Integration des Testkonzepts in die bestehenden Entwicklertools konnte die Rückmeldung von Fehlern in bereits bestehenden (und vermeintlich funktionierenden) Softwarebausteinen schneller erkannt werden. Sämtliche bereits vorhandene Testfälle wurden bei jedem Deployment des Systems (Update der bestehenden Software auf neuesten Stand) erneut ausgeführt. Dadurch konnte bereits im Zuge dieses Vorgangs eine entsprechende Prüfung über die Funktionstüchtigkeit der Software durchgeführt werden, wodurch neue Abweichungen in bereits abgenommenen Softwarebausteinen frühzeitig erkannt und behoben werden konnten. Aufgrund der geringen Testfallanzahl konnte dieser Effekt nur in geringen Ausmaß im Pilotprojekt gezeigt werden, da die Anzahl der Testfälle für eine flächendeckende Abdeckung der bereits bestehenden Softwarefunktionalität zeitlich nicht möglich war (siehe Tabelle 5.1). Da der Aufwand in der Testfallerstellung generell höher ist, müssen hierbei bei jedem Projekt individuell die dafür anfallenden Kosten und der erwartete Nutzen gegenübergestellt werden.

5.4 Zusammenfassung

Im Zuge des Pilotprojektes konnten sechs Testfälle durch Mitarbeiter aus den Fachabteilungen ohne technischen Hintergrund mit dem neu entwickelten Testkonzept erstellt werden. Diese Testfälle konnten im Zuge der jeweiligen Sprintplanung erstellt werden, wodurch der Ansatz von *Test Driven Development* gezeigt werden konnte. Die Testfälle wurden vor der Entwicklung erstellt und im Zuge der Entwicklungstätigkeiten im Sprint als automatische Prüfung auf Korrektheit herangezogen. Bereits erstellte Testfälle aus den vorherigen Sprints konnten als Regressionstests verwendet werden, um Fehler in bereits implementierter Software zu identifizieren.

Wie in der Tabelle 5.1 ersichtlich, konnte dadurch die Anzahl der positiv durchgeführten Testfälle mit Fortdauer des Einsatzes erhöht werden, wodurch eine Verringerung von Fehlern in bereits bestehender Software erzielt werden konnte. Aufgrund von fehlenden Kennzahlen als Referenzwert kann hier jedoch keine deutliche Aussage getroffen werden!

Durch die Erstellung der Testfälle mit der Webapplikation *AWATAR-Creator* durch die Fachabteilungen konnte weiters ein Schulungseffekt und eine damit höhere Akzeptanz der neu entwickelten Software in den Bereichen erzielt werden. Durch das aktive Erstellen der Testfälle, konnten die Fachbereiche das Funktionsverhalten durch die Testfalldefinition indirekt steuern, wodurch ein positives Feedback aus den jeweiligen Bereichen zu vernehmen war. Als negative Auswirkung musste die Mehrbelastung der Fachbereiche dokumentiert werden. Die Ablaufspezifikation (durch die Testfälle) wurde durch die Einbindung der Fachbereiche in diese verlagert, wodurch zusätzlicher Aufwand zum Tagesgeschäft in diese Bereiche gelangte.

Des Weiteren konnte im Pilotprojekt die Integration des erstellten Testkonzepts in eine bestehende Softwarelandschaft für die Softwareentwicklung gezeigt werden. Wie in Abschnitt 5.2 beschrieben, wurde die Testausführung über das Softwaretool *Jenkins* automatisiert. Durch die versionsbasierte Verwaltung der Testfälle mithilfe von *GIT* wurde bei der Testausführung immer das aktuell gültige Testset verwendet. Durch die Integration des Testkonzepts in das Entwicklungstool *Jenkins* wurden die Tests bei jedem Deploymentvorgang (Update der bestehenden Software auf aktuellsten Stand) durchgeführt. Dadurch konnte zeitnah eine Rückmeldung über den Erfolg der Tests erreicht werden und bei Fehlern die Fehlerbehebung rasch erfolgen.

6 Schlussfolgerung

Ziel dieser Arbeit ist die Erstellung eines Testkonzepts zur Einführung von End-to-End Tests im Konzept von *Test Driven Development* und *Behavior Test Driven Design*. Durch die Entwicklung des Konzepts soll Entwicklungsteams mit geringen finanziellen Mitteln ein Werkzeug zur Verfügung gestellt werden, mit dem ganzheitliche Prozessabläufe einer Webapplikation (beispielsweise eines E-Commerce Systems) abgebildet werden können. Durch den Einsatz des entworfenen Konzepts in einem Pilotprojekt mithilfe der Firma B4B Solutions GmbH soll das Konzept in einem Projekt in der Praxis eingesetzt werden und die daraus resultierenden Ergebnisse als Erfahrungsbericht aufbereitet werden.

Weiters soll durch die Definition der Begriffe *Test Driven Development* und *Behavior Test Driven* eine einheitliche Basis für den Verwendungszweck des Testkonzepts geschaffen werden. Im Zuge der Erstellung des theoretischen Konzepts werden bestehende Testframeworks auf deren Vor- und Nachteile geprüft und bewertet. Diese Analyse soll dazu dienen, einen allgemeinen Ansatz zu beschreiben, der eine durchgängige Prüfung von Prozessabläufen (End-to-End Tests) in Webapplikationen ermöglicht. In diesem Ansatz sollen gut funktionierende Konzepte von bereits vorhandenen Testframeworks einfließen und integriert werden, sodass ein ganzheitliches Testsystem entsteht.

Basierend auf den durchgeführten Recherchetätigkeiten und der praktischen Umsetzung des Testkonzepts sowie dessen Anwendung in einem Pilotprojekt lassen sich folgende Aussagen als Quintessenz zur Beantwortung der Forschungsfrage in Abschnitt 1.1 Zielsetzung und Forschungsfragen des Kapitels 1 Einleitung und Problemstellung ableiten:

Prozessabläufe innerhalb einer Webanwendung können mithilfe von automatisierten Tests geprüft werden, wenn die Prüfung auf der Ebene der Oberfläche erfolgt. Durch die Prüfung der Eingabemasken und Darstellung von vorhandenen Daten (beispielsweise Produkte im Produktkatalog oder Preisinformationen auf der Produktdetailseite) erfolgt die indirekte Prüfung der dahinterliegenden Schnittstellen automatisch. Dadurch können Prozessabläufe aus Anwendersicht durchgeführt und das erwartete Ergebnis analysiert werden. Mithilfe dieser Tests kann keine qualitätssichernde Maßnahme auf atomarer Ebene, beispielsweise eine isolierte Funktionalitätsprüfung von Methoden

oder Schnittstellen, erfolgen. Dies ist aber auch nicht das Ziel von End-to-End Tests und wird im V-Modell nach Böhm beschrieben (Spillner & Linz, 2012). Abbildung 2.1 zeigt das Modell in dem die jeweilige Phase des Entwicklungszyklus mit der Testphase in Verbindung gebracht wird.

Für den Einsatz von End-to-End Tests eignet sich der Ansatz von *Test Driven Development*, da Prozessabläufe meist bereits vor der Entwicklung von Softwarekomponenten bestehen und die neu entwickelte Software in diesem Prozess eine unterstützende Komponente einnimmt. Dadurch sollte der bestehende Prozessablauf als Definition für die erwartete Funktionalität verwendet werden, um Entwicklungen in eine falsche Richtung zu minimieren. Unter *Test Driven Development* versteht man nach Janzen (Janzen & Saiedian, 2005) einen iterativen Ansatz, um Software ohne großen *Vorab-Design* zu erstellen. Der Testfall spiegelt das erwartete Ergebnis wider und treibt die Entwicklung bis zur Erfüllung des Testfalls in die entsprechend korrekte Richtung. Konkretisiert auf Prozessabläufe in einem Webshop könnte dies ein korrekt durchgeführter Bestellprozess sein, von der Produktsuche bis über die Erstellung des Warenkorbs und der daraus resultierenden Bestellung. Innerhalb dieses Prozesses gibt es viele Schnittstellen zwischen Systemen (beispielsweise das Auslesen der Preisinformationen aus dem ERP-System oder das Abrufen der Lieferadresse aus dem CRM-System), welche als einzelne Unterprozesse oder Prozessschritte definiert werden. Mithilfe dieser Prozessbeschreibungen und dem Einsatz des Ansatzes von *Test Driven Development* können Entwicklung nah am erwarteten Kundennutzen realisiert und das implementierte Ergebnis schnell verifiziert werden.

Dieser Aspekt ist ein zentraler Punkt in der Entwicklung des Testkonzepts und stellt eine wichtige Anforderung an dieses dar. Das Testkonzept muss die Testfallerstellung auf Prozessebene ermöglichen und unabhängig von der zu entwickelnden Software funktionieren, um die Testfälle bereits vorab (beispielsweise in der Sprintplanung) zu ermöglichen. Eine weitere Säule in der Entwicklung des Konzepts stellt der Ansatz *Behavior Test Driven* dar. Dieser Ansatz beschreibt die klare Definition von Anforderungen in Aktionen: Es gibt klare Vorbedingungen (**GIVEN**, ausgeführte Aktionen **WHEN** und erwartete Ergebnisse **THEN**. Diese drei Gruppen bilden das zentrale Element eines Testfalls und können bei Bedarf noch um weitere Felder erweitert werden. Neben der klaren Formgebung ermöglicht der Ansatz auch die Wiederverwendbarkeit von Testschritten. Durch die Kombination von *Test Driven Development* und dem Ansatz von *Behavior Test Driven* können automatisierte Tests auf Prozessebene definiert werden.

Für die technische Umsetzung von automatisierten Tests für Webanwendungen gibt es bereits einige Frameworks, die je nach Anwendungsfall eingesetzt werden können. Die

einzelnen untersuchten Anwendungen sind in Kapitel 3 Testframeworks beschrieben. Die untersuchten Frameworks wurden mithilfe einer Nutzwertanalyse (siehe Abschnitt 3.6) bewertet, um eine fundierte Entscheidung zur Auswahl eines Frameworks zu treffen. Für die technische Umsetzung des Testkonzepts wurde die API **Selenium API** verwendet, da diese von allen anderen untersuchten Frameworks bereits als Basis verwendet wird und die größte Flexibilität und Individualisierbarkeit bietet. Weiters verfügt dieses Framework über eine sehr aktuelle Dokumentation und wird ständig weiterentwickelt. Weiters wurde für die technische Realisierung des Ansatzes *Behavior Test Driven* das Framework **Cucumber API** eingesetzt. Dieses Framework ermöglicht die textuelle Erfassung von Testfällen ohne technisches Wissen und bietet eine automatisierte Verknüpfung der Testfallbeschreibung mit ausführbarem Code. Dadurch wird ermöglicht, dass Testschritte in textueller Form beschrieben werden und somit von Mitarbeitern aus Fachabteilungen mit Wissen über die Prozessabläufe innerhalb des Unternehmens beschrieben werden können.

Mithilfe der Kombination der beiden Frameworks wurde ein neues Testkonzept entwickelt, welches in modularer Bauweise realisiert wurde. Dadurch können die einzelnen Funktionalitäten individuell eingesetzt und dem jeweiligen Entwicklungsprojekt angepasst werden. Das Konzept sieht vier Subsysteme vor:

- **AWATAR-Core**: Eine zentrale Serverapplikation zur Bereitstellung von bereits existierenden Testmethoden und Ausführung von Testfällen.
- **AWATAR-Creator**: Eine Webapplikation zur Erstellung von Testfällen nach dem *Behavior Test Driven*-Ansatz in textueller Form.
- **AWATAR-Runner**: Eine Webanwendung zur Ausführung von erstellten Testfällen.
- **AWATAR-Resultant**: Eine Anwendung zum Abholen und Verwalten von ausgeführten Testfällen.

Im Zuge der Prototypentwicklung wurde die Applikation *AWATAR-Resultant* nicht entworfen, da die Testergebnisse direkt mithilfe der Applikation *AWATAR-Runner* bereitgestellt wurden und keine Datenbank zur Persistierung der Testergebnisse realisiert wurde. In Unterabschnitt 4.4.6 des Abschnitts 4.4 Technische Umsetzung wird jedoch die theoretische Umsetzung der Applikation beschrieben. Die Applikation *AWATAR-Core* verwaltet sämtliche bereits implementierte Testmethoden und stellt diese über eine HTTP-REST-Schnittstelle zur Verfügung.

Neben der Bereitstellung der verfügbaren Testmethoden erfolgt die Testausführung ebenfalls über dieses Subsystem. Es wird eine eigene REST-Schnittstelle für die Test-

ausführung angeboten, welche von der Webapplikation *AWATAR-Runner* genutzt wird. Diese kann jedoch auch von anderen Systemen verwendet werden, beispielsweise von einem Testmanagementsystem. Die bereitgestellten Informationen über die vorhandenen Testmethoden werden von der Webapplikation *AWATAR-Creator* genutzt, mit der Testfälle im Ansatz von *Behavior Test Driven* erstellt werden können. Diese Testfälle können vor Implementierung der Funktionalität im zu testenden System erfolgen, wodurch auch das Konzept von *Test Driven Development* realisiert werden kann.

Durch den Einsatz des entwickelten Prototyps in einem Referenzprojekt konnten Erfahrungen in Bezug auf die Erstellung der Testfälle im Ansatz von BDT und TDD gesammelt werden. Die Testfallerstellung konnte durch Mitarbeiter aus den Fachabteilungen ohne technischen Hintergrundwissen erfolgen, wodurch diese Prozessabläufe widerspiegeln konnten. Weiters konnten diese Testfälle bereits in der Phase der Sprintplanung erstellt werden, wodurch die Entwicklung der Funktionalität durch die Tests getrieben wurde. Dadurch konnte mit dem entwickelten Testkonzept der Entwicklungsansatz *Test Driven Devlopment* in das Praxisprojekt eingeführt werden. Durch die klar spezifizierten Testfälle konnte die Anforderung an die Entwicklung spezifischer definiert und das Produkt in Summe näher am bestehenden Prozess realisiert werden.

Weiters wurde die Testausführung in das bereits eingesetzte Entwicklungstool *Jenkins* integriert, um diese automatisiert und kontinuierlich auszuführen (Continious Integration). Dadurch konnten die bereits erstellten Testfälle aus vorangegangen Sprints als Regressionstests genutzt werden, um die bereits bestehende Funktionalität auf Funktionstüchtigkeit zu prüfen und Seiteneffekte frühzeitig zu erkennen. Aufgrund eines fehlenden Testmanagementsystems im Referenzprojekt konnte die beispielhafte Integration in dieses System nicht gezeigt werden. Durch die Integration in das Entwicklungstool *Jenkins* konnte jedoch die Möglichkeit geschaffen werden, dass sämtliche Mitarbeiter mit Zugang jederzeit eine Prüfung des Systems (mit den bestehenden Testfällen) anstoßen und so eine Qualitätsbestimmung jederzeit möglich gemacht wird.

Mit Fortdauer der eingesetzten Testfälle konnte gezeigt werden, dass die Anzahl der positiv durchgeführten Tests steigt. Aufgrund der geringen Anzahl an Testfällen und der fehlenden Referenzkennzahlen kann hier jedoch nur ein Trend über den Nutzen der Tests abgelesen werden. Die Rückmeldungen im Zuge des Prototypenprojektes können jedoch als positiv gewertet werden. Insbesondere durch die dadurch frühzeitige Einbeziehung der Fachabteilungen in den Entwicklungsprozess konnte die Akzeptanz von neuer Funktionalität in den einzelnen Bereichen gesteigert werden. Weiters wurde als Feedback zurückgemeldet, dass eine indirekte Schulung des Personals innerhalb der Fachabteilungen erfolgte, da diese sich bereits vor der Fertigstellung der neuen

Funktionalität beschäftigten mussten und somit besseres Feedback in die Sprintplanung einbringen konnten.

In Bezug auf die Einsatzmöglichkeiten des entwickelten Testkonzepts lässt sich im Allgemeinen sagen, dass dieses den Ansatz von *Test Driven Development* bei der Entwicklung von Webanwendungen (insbesondere SAP Commerce) unterstützt und zusätzlich das Konzept von *Behavior Test Driven* ermöglicht. Durch die modulare Architektur und der implementierten HTTP-REST-Schnittstelle von *AWATAR* kann das System leicht in eine bestehende Systemlandschaft integriert werden. Mithilfe des Subsystems *AWATAR-Creator* ist es möglich, die Testfallerstellung durch Personal ohne technischen Hintergrund durchführen zu lassen. Dadurch können Personen mit gutem Prozesswissen die Testfälle spezifizieren, welche für die technische Umsetzung genutzt werden können. Das Testkonzept soll als zusätzliche Ebene der Qualitätssicherung angesehen werden und ersetzt nicht die Implementierung von Unit-Tests oder Integrationstests. Eine detaillierte Prüfung von Funktionalität (beispielsweise auf Methoden-Ebene) ist mit dem entwickelten System nicht möglich, da die Prüfungen immer auf der Weboberfläche der Applikation erfolgt.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
AWATAR	A Web Applicatoin Test Automation Resource
BDT	Behavior Driven Testing
BPMN	Business Process Modell Notation
CRM	Customer Relationshop Managementsstem
ERP	Enterprise Ressource Planing System
GUI	Graphical User Interface
KMU	Kleinere und Mittele Unternehmen
TDD	Test Driven Development

Listings

Literaturverzeichnis

Astels, D. (2003). *Test-driven development: A practical guide*. Upper Saddle River, N.J.: Prentice Hall PTR. Zugriff auf http://proquest.tech.safaribooksonline.de/0131016490

B. W. Boehm. (1984). Verifying and validating software requirements and design specifications. *IEEE Software*, *1* (1), 75–88.

Bissi, W., Serra Seca Neto, A. G. & Emer, M. C. F. P. (2016). The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, *74*, 45–54. doi: 10.1016/j.infsof.2016.02.004

Bréhault, É. (2014). *Instant testing with casperjs: Create advanced and efficient casperjs tests for your web development projects*. Birmingham, UK: Packt Pub. Zugriff auf http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=691846

Bruns, A., Kornstadt, A. & Wichmann, D. (2009). Web application tests with selenium. *IEEE Software*, *26* (5), 88–91. doi: 10.1109/MS.2009.144

Carbonnelle, P. (2020). *Pypl popularity of programming language.* Zugriff am 30.07.2020 auf https://pypl.github.io/PYPL.html

Chece, S. (2016). *Testautomatisierung-tools.* Zugriff am 30.07.2020 auf https://www.testing-board.com/testautomatisierung-tools/

Fojtik, R. (2011). Extreme programming in development of specific software. *Procedia Computer Science*, *3*, 1464–1468. doi: 10.1016/j.procs.2011.01.032

Fowler Martin. (2013). *Givenwhenthen.* Zugriff am 19.06.2020 auf https://martinfowler.com/bliki/GivenWhenThen.html

Fritzsche, M. & Keil, P. (Juni 2007). *Kategorisierung etablierter vorgehensmodelle und ihre verbreitung in der deutschen software-industrie.* Zugriff am 10.06.2020 auf https://mediatum.ub.tum.de/doc/1094277/file.pdf

George, B. & Williams, L. (2004). A structured experiment of test-driven development. *Information and Software Technology*, *46* (5), 337–342. doi: 10.1016/j.infsof.2003.09.011

Janzen, D. & Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer*, *38* (9), 43–50. doi: 10.1109/MC.2005.314

Karanam, R. (2019). *Best java unit testing frameworks: Unit testing is an important*

skill for programmers. Zugriff am 14.06.2020 auf https://dzone.com/articles/best-java-unit-testing-frameworks

Karlesky, M., Williams, G., Bereza, W. & Fletcher, M. (April 2017). *Mocking the embedded world: Test-driven development, continuous integration, and design patterns.* Embedded Systems Conference Silicon Valley (San Jose, California). Zugriff am 12.06.2020 auf https://www.researchgate.net/profile/Michael_Karlesky/publication/228711578_Mocking_the_embedded_world_Test-driven_development_continuous_integration_and_design_patterns/links/5512b1c80cf20bfdad51c6b4.pdf

Kerthyayana Manuaba, I. B. (2019). Combination of test-driven development and behavior-driven development for improving backend testing performance. *Procedia Computer Science*, *157*, 79–86. doi: 10.1016/j.procs.2019.08.144

Perriault, N. (o. J.). *Casperjs: Navigation scripting & testing for phantomjs and slimerjs.* Zugriff am 30.07.2020 auf https://www.casperjs.org/

Postman. (o. J.). *The collaboration platform for api development.* Zugriff auf https://www.postman.com/

Protractor. (o. J.). *Protractor - end to end testing for angular.* Zugriff am 06.08.2020 auf https://www.protractortest.org

Python Software Foundation. (13.06.2020). *The python standard library: unittest - unit testing framework.* Zugriff am 14.06.2020 auf https://docs.python.org/3/library/unittest.html

SAP SE. (2020). *Installing and upgrading sap commerce: Installing a local instance.* Zugriff am 30.07.2020 auf https://help.sap.com/viewer/a74589c3a81a4a95bf51d87258c0ab15/2005/en-US/8c71300f866910149b40c88dfc0de431.html

Selenium HQ. (o. J.). *The selenium browser automation project: Selenium automates browsers. that's it!* Zugriff am 30.07.2020 auf https://www.selenium.dev

SmartBear Software. (2020). *Cucumber open: Tool to support behaviour-driven development(bdd).* Zugriff am 30.07.2020 auf https://cucumber.io/

Sommerville, I. (2016). *Software engineering* (Tenth edition, global edition Aufl.). Boston and Columbus and Indianapolis and New York and San Francisco and Hoboken and Amsterdam and Cape Town and Dubai and London and Madrid and Milan and Munich and Paris and Montreal and Toronto and Delhi and Mexico City and São Paulo and Sydney and Hong Kong and Seoul and Singapore and Taipei and Tokyo: Pearson.

Spillner, A. & Linz, T. (2012). *Basiswissen softwaretest: Aus- und weiterbildung zum certified tester ; foundation level nach istqb-standard* (5. überarb. und aktualisierte Aufl. Aufl.). Heidelberg: dpunkt Verl. Zugriff auf https://ebookcentral.proquest

.com/lib/subhh/detail.action?docID=1049740

Suryawanshi, B. (Mai 2019). *What is test driven development (tdd) in javascript?* Zugriff am 14.06.2020 auf https://medium.com/@bharat.suryawanshi2010/what-is-test-driven-development-tdd-9b9b18cb6bc7

Watir. (2018). *Watir.* Zugriff am 06.08.2020 auf http://watir.com/guides/overview/

Wirdemann, R. & Mainusch, J. (2017). *Scrum mit user stories* (3., erweiterte Auflage Aufl.). München: Hanser.

Wynne, M. & Hellesøy, A. (2012). *The cucumber book: Behaviour-driven development for testers and developers*. Dallas, Texas: Pragmatic Bookshelf.

You, E. (2014). *Vuejs: The progressive javascript framework.* Zugriff am 16.08.2020 auf https://vuejs.org/