 Hochschule

Fachbereich Wirtschaft

Bachelorarbeit

**Cucumber –  
 Vorstellung und Demonstration des Frameworks zur Überprüfung der Akzeptanzkriterien von User Stories einer RESTful API**

Erstprüfer: Prof. Dr. rer. oec. Bernd Blümel

Zweitprüfer: Dipl. Inf. Stephan Müller

Vincent Becker

Matrikelnummer: 015207561

Herner Straße 40

44787 Bochum

Tel.: 0152/33901468

E-Mail: vincent.becker@openknowledge.de

Studiengang:

Wirtschaftsinformatik

8. Fachsemester, Wintersemester 2018/19

Abgabedatum 21. Mai 2013

Inhaltsverzeichnis[Abkürzungsverzeichnis IV](#_Toc14296575)

[Abbildungsverzeichnis VI](#_Toc14296576)

[Tabellenverzeichnis VIII](#_Toc14296577)

[1 Einführung 10](#_Toc14296578)

[1.1 Motivation 10](#_Toc14296579)

[1.2 Zielsetzung und Struktur der Arbeit 11](#_Toc14296580)

[1.3 Open Knowledge GmbH 12](#_Toc14296581)

[2 Behavior Driven Development 14](#_Toc14296582)

[2.1 Zweck 14](#_Toc14296583)

[2.2 Vorgehen 15](#_Toc14296584)

[2.2.1 Example Mapping 17](#_Toc14296585)

[2.2.2 Abgrenzung BDD zu TDD 18](#_Toc14296586)

[3 Cucumber 19](#_Toc14296587)

[3.1 Grundlagen 19](#_Toc14296588)

[3.1.1 Zweck 19](#_Toc14296589)

[3.1.2 Architektur 20](#_Toc14296590)

[3.1.3 Gherkin 21](#_Toc14296591)

[3.2 Exemplarische Darstellung wichtiger Cucumber-Elemente 23](#_Toc14296592)

[3.2.1 Szenario 23](#_Toc14296593)

[3.2.2 Step-Definitionen 24](#_Toc14296594)

[3.2.3 Scenario Outline 25](#_Toc14296595)

[3.2.4 Parameter übergeben 27](#_Toc14296596)

[3.2.5 Doc Strings 29](#_Toc14296597)

[3.2.6 Data Tables 29](#_Toc14296598)

[3.2.7 Background 30](#_Toc14296599)

[3.2.8 Type Registry 31](#_Toc14296600)

[3.2.9 Hooks 33](#_Toc14296601)

[3.3 Arbeitsweise 34](#_Toc14296602)

[3.3.1 Aufruf eines Cucumber Tests 34](#_Toc14296603)

[3.3.2 Konfiguration der Testausführung 34](#_Toc14296604)

[3.3.3 Testvorbereitung 35](#_Toc14296606)

[3.3.4 Testausführung 36](#_Toc14296607)

[3.4 Best Practices 37](#_Toc14296608)

[3.4.1 Vorgehen 37](#_Toc14296609)

[3.4.2 Regressionsfähige (unabhängige) Szenarien 37](#_Toc14296610)

[3.4.3 Szenarien wie Geschichten beschreiben 38](#_Toc14296611)

[3.4.4 Genaues Wording bei den Steps 38](#_Toc14296612)

[3.4.5 Step-Definitionen Klassen übersichtlich halten 38](#_Toc14296613)

[3.4.6 Szenarien übersichtlich halten 39](#_Toc14296614)

[3.4.7 Verwendung von Hooks 39](#_Toc14296615)

[3.4.8 Zustände teilen 39](#_Toc14296616)

[4 Exemplarische RESTful Beispielanwendung 41](#_Toc14296617)

[4.1 API und UseCases 41](#_Toc14296618)

[4.2 Technischer Hintergrund 43](#_Toc14296619)

[4.3 Tests ausführen 43](#_Toc14296620)

[4.4 Vorstellung der Features 43](#_Toc14296621)

[4.4.1 Package Aufbau 44](#_Toc14296622)

[4.4.2 Feature-Datei 44](#_Toc14296623)

[4.4.3 Technische Begriffe 44](#_Toc14296624)

[4.4.4 Cucumber-Testklasse 45](#_Toc14296625)

[4.4.5 Testvorbereitung 46](#_Toc14296626)

[4.4.6 Überprüfung der Schnittstelle 50](#_Toc14296627)

[4.4.7 Zustände teilen 54](#_Toc14296628)

[4.4.8 Authentication Token 55](#_Toc14296629)

[4.4.9 Arten der JSON Beschreibung 55](#_Toc14296630)

[4.4.10 Wiederholende Szenarien 56](#_Toc14296631)

[4.4.11 Verwendung des Scenario Outline 56](#_Toc14296632)

[5 Fazit 60](#_Toc14296633)

[Literaturverzeichnis LXII](#_Toc14296634)

Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.**Fehler! Linkreferenz ungültig.Fehler! Linkreferenz ungültig.

Abkürzungsverzeichnis

BDD Behavior Driven Development

TDD Test Driven Development

GUI graphical user interface

IDE Integrated development editor

CLI Command Line Interface

Abbildungsverzeichnis

[Abbildung 1: Unterschiedliche Verständnisse von Softwareanforderungen 11](#_Toc12891962)

[Abbildung 2: Struktur eines Cucumber Projektes 21](#_Toc12891963)

[Abbildung 3: Struktur einer Gherkin Feature-Datei 23](#_Toc12891964)

[Abbildung 4: Step-Definitions Beispiel 25](#_Toc12891965)

[Abbildung 5: Szenario Outline Beispiel 27](#_Toc12891966)

[Abbildung 6: Parameter Übergabe mit Captured Groups 28](#_Toc12891967)

[Abbildung 7: Flight Gherkin Step 29](#_Toc12891968)

[Abbildung 8: Flight Step-Definition mit regulärem Ausdruck 29](#_Toc12891969)

[Abbildung 9:Flight Step-Definition mit Cucumber Expression 29](#_Toc12891970)

[Abbildung 10: Doc String Beispiel 31](#_Toc12891971)

[Abbildung 11: Data Table Beispiel 31](#_Toc12891972)

[Abbildung 12: Background Beispiel 33](#_Toc12891973)

[Abbildung 13: Registrierung eines Output Parameters 34](#_Toc12891974)

[Abbildung 14: Flight Data Table 34](#_Toc12891975)

[Abbildung 15: Registrierung eines Data Table Typens 34](#_Toc12891976)

[Abbildung 16: Verwendung eines registrierten Data Table Typens 35](#_Toc12891977)

[Abbildung 17: Create tweet Szenario 49](#_Toc12891978)

[Abbildung 18: Background-Block im "Get users" Featrure 51](#_Toc12891979)

[Abbildung 19: Unterschiedliche Rollen fragen user Informationen ab 52](#_Toc12891980)

[Abbildung 20: Mehrere Example Tabellen in "Get users" Feature 53](#_Toc12891981)

[Abbildung 21: Cucumber Testklasse "Create tweet" Feature 55](#_Toc12891982)

[Abbildung 22: Versenden eines Requests mit RestAssured 56](#_Toc12891983)

[Abbildung 23: Validierung einer Response mit RestAssured 56](#_Toc12891984)

[Abbildung 24: Instanziierung der ShareDomain 57](#_Toc12891985)

[Abbildung 25: Before-Hook zum leeren der Datenbank 58](#_Toc12891986)

[Abbildung 26: Szenario "Cancel a tweet " 58](#_Toc12891987)

[Abbildung 27: Step-Definition zu Step von "Moderator cancel tweet" 59](#_Toc12891988)

Tabellenverzeichnis

[Tabelle 1: Mögliche Ergebnisse einer Step-Definition 24](#_Toc12892023)

[Tabelle 2: Cucumber Annotationen zu den Gherkin Key Words 25](#_Toc12892024)

[Tabelle 3: Built-in Output Parameter 29](#_Toc12892025)

1. Einführung
   1. Motivation

Wenn die Idee für eine Anwendung entsteht, entsteht damit auch die Herausforderung diese in die Realität umzusetzen. Da es sich beim Ideengeber (Stakeholder/ Auftraggeber) und dem Software-Entwickler i.d.R. nicht um dieselbe Person handelt, wird ab diesem Moment viel Kommunikation erforderlich. Stakeholder die sich mit der Fachlichkeit auskennen, die von der Software erfüllt werden soll, werden „*Domänen-Experten“* genannt. Sie und die Software-Entwickler müssen genau klären, was von der zu erstellenden Anwendung erwartet wird und welche Anforderungen an diese gestellt werden. Nichtsdestotrotz kommt es immer wieder zu Missverständnissen auf beiden Seiten – eine Problematik, die auch nach vielen Jahrzehnten der Softwareentwicklung unverändert besteht (vgl. Abb. 1).

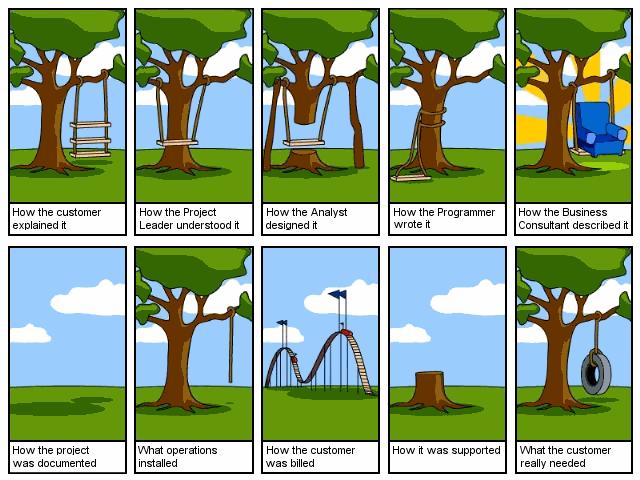


Abbildung 1: Unterschiedliche Verständnisse von Softwareanforderungen

Quelle: Online im Internet: <http://www.doktor-meffert.de/softwareentwicklung-loesungen-fuer-ihre-anforderungen/>, abgerufen am 01.04.2019

Um die Kommunikationsprobleme zwischen Domänen-Experten und Software-Entwickler zu lösen, wurde Behavior-Driven-Development („BDD“) entwickelt.

BDD stellt eine neuere Vorgehensweise der Softwareentwicklung dar, bei der anhand konkreter Beispiele des erwarteten Verhaltens bereits vor Beginn der Implementierung ein gemeinsames Verständnis über die zu erstellende Anwendung erlangt werden soll. Die Beispiele definieren Zustände, die nach Eintreten eines bestimmten Ereignisses und unter einem zuvor definierten Ausgangszustand entstanden sein sollten. Neben der Erklärung des Verhaltens der Anwendung, stellen die Beispiele gleichzeitig fachliche Tests dar, mit deren Hilfe Softwaretests erstellt werden können, die überprüfen, ob die Anwendung den Anforderungen entspricht.

Die Herausforderung bei der Implementierung der Softwartetests zu den fachlichen Tests besteht darin, die Übersicht zu erhalten. Der Überblick, zu welchem fachlichen Test ein Softwaretest gehört und zu welchem noch keine Implementierung existiert, kann schnell verloren gehen. Zu diesem Zweck wurde das Framework „Cucumber“ entwickelt, mit dem die fachlichen Tests mit den Softwaretests verbunden werden können.

* 1. Zielsetzung und Struktur der Arbeit

Im Rahmen dieser Arbeit soll das Framework „Cucumber“, zur vereinfachten Durchführung von Softwareprojekten im BDD-Verfahren vorgestellt werden. Die Arbeit beschränkt sich dabei ausschließlich auf „*Cucumber-JVM*“ – die Java Version von Cucumber.

Anhand kurzer exemplarischer Beispiele werden die wichtigsten Funktionalitäten von Cucumber aufgezeigt. Im Anschluss wird anhand einer Beispielanwendung die Einsatzmöglichkeiten von Cucumber bei der Entwicklung einer RESTful API demonstriert.

Zunächst wird BDD, die theoretische Grundlage des Cucumber Testframeworks, in Kapitel 2 vorgestellt. In Kapitel 3 folgt die Erläuterung der Architektur und Arbeitsweise von Cucumber, sowie die Vorstellung der wichtigsten Features an verständlichen Beispielen. Darüber hinaus wird die Beziehung des Frameworks zur Syntaxsprache „*Gherkin*“ aufgezeigt, in der die fachlichen Tests formuliert werden. Kapitel 4 befasst sich mit der Beispielanwendung, bei dessen Realisierung Cucumber zum Einsatz gekommen ist. Den Schluss bildet Kapitel 5 mit einem Fazit, indem auf die Möglichkeiten, aber auch auf die bestehenden Probleme von Cucumber eingegangen wird.

* 1. Open Knowledge GmbH

Die open knowledge GmbH ist ein brancheneutrales mittelständisches Dienstleistungs-Unternehmen mit dem Ziel bei der Analyse, Planung und Durchführung von Softwareprojekten zu unterstützen. Das Unternehmen wurde im Jahr 2000 in Oldenburg, dem Hauptsitzt des Unternehmens, gegründet und beschäftigt heute ca. 70. Mitte 2017 wurde der zweite Standort in Essen eröffnet an dem aktuell 10 Mitarbeiter angestellt sind.

Die Mitarbeiter von open knowledge GmbH übernehmen in Kundenprojekten Aufgaben bei der Analyse über die Projektziele und der aktuellen Ausgangssituationen, der Konzeption der geplanten Software, sowie der anschließenden Implementierung. Die erstellten Softwarelösungen stellen Individuallösungen dar und werden den Bedürfnissen der einzelnen Kunden entsprechend konzipiert und implementiert. Technisch liegt die Spezialisierung bei der Mobile-und bei der Java Enterprise Entwicklung, bei der stets moderne Technologien und Konzepte verwendet werden. Aufgrund der großen Expertise in den Bereichen Technologien und Konzepte sind sowohl die Geschäftsführer als auch diverse Mitarbeiter der open knowledge GmbH als Speaker auf Fachmessen wie der Javaland oder Autoren in Fachzeitschriften wie dem Java Magazin vertreten. Auch aufgrund der medialen Präsenz konnte die open knowledge GmbH in den letzten Jahre zahlreiche Projekte für namenhafte Kunden, wie z.B. Vodafone, die HUK-Coburg, die Daimler AG und die DB Schenker AG übernehmen.

Durch das breite Dienstleistungsangebot und die Branchenneutralität übernimmt die open knowledge GmbH Projekte aus der Automobilindustrie, der Luft-und Raumfahrt, dem Bankwesen und dem Versicherungswesen innerhalb des nationalen und europäischen Raums.

Entsprechend der Philosophie „Offenkundig Gut“ und dem Namen des Unternehmens wird stets versucht das benötigte Wissen zur Erstellung und Wartung der Software mit dem Kunden zu teilen, sodass die Kunden nach Abschluss der Projekte in der Lage sind, die Software selbstständig zu Pflegen und die Projektarbeit von OK damit abgeschlossen ist.

1. Behavior Driven Development
   1. Zweck

Behavior-Driven-Development (BDD) ist eine Technik des Software-Engineerings die insbesondere in Softwareprojekten mit agiler Vorgehensweise eingesetzt wird.

Bei BDD steht vor allem eine verbesserte Zusammenarbeit aller Beteiligten von Softwareprojekten im Vordergrund. Das übergeordnete Ziel ist es, frühzeitig ein detailliertes gemeinsames Verständnis über die zu erstellende Anwendung zu schaffen und allen Teammitgliedern, unabhängig von ihrem technischen Vorwissen, zu ermöglichen, das gewünschte Verhalten zu verstehen.[[1]](#footnote-1) Hierzu wird das Verhalten auf unterschiedliche Ereignisse in einem zuvor definierten Kontext in Form von konkreten Beispielen in einer einheitlichen und nicht-technischen Sprache (i.d.R. Englisch oder Deutsch) beschrieben.

Die konkreten Beispiele stellen die fachlichen Tests dar, anhand derer die technischen Tests erstellt werden. Da die Softwaretests die fachlich definierten Tests ausführen, können Domänen-Experten auch ohne technische Kenntnisse nachvollziehen, welches Verhalten durch die Tests überprüft wird.

Die Beschreibung des gewünschten Verhaltens anhand konkreter Beispiele ermöglicht es den Beteiligten zu überprüfen, ob ein gemeinsames Verständnis von der Software existiert. Domänen-Experten können durch das Aufstellen der Beispiele das gewünschte Verhalten der Software erklären und Software-Entwickler durch formulieren eigener Beispiele ein Feedback an die Domänen-Experten geben, wie sie das Verhalten verstanden haben, noch bevor sie mit dessen Implementierung beginnen.[[2]](#footnote-2) Durch die besondere Art des Austauschs und der Zusammenarbeit soll sichergestellt werden, dass **das richtige**, entsprechend der Anforderungen, implementiert wird und kein Code entsteht, der zu einem späteren Zeitpunkt wieder geändert werden muss, oder auf dem sogar bereits weiterer Code aufgebaut wird.[[3]](#footnote-3)

* 1. Vorgehen

Da ein gemeinsames Verständnis im Vordergrund steht, ist eine enge Zusammenarbeit dreier Rollen – den Three Amigos[[4]](#footnote-4) – erforderlich. Hierzu gehört die Rolle des Domänen-Experten, der sich mit der fachlichen, von der Software abzubildenden Fachlichkeit auskennt, des Software-Entwicklers und des Testers der Anwendung.[[5]](#footnote-5)

Idealerweise treffen sich Vertreter der drei Rollen in regelmäßigen Abständen zu „Three-Amigos-Meetings“ (auch „Discovery-Workshop“[[6]](#footnote-6) oder „Specification-Workshop“ genannt).[[7]](#footnote-7) Sinn und Zweck dieser Termine ist die Diskussion der vom Auftraggeber gewünschten User Stories, die die einzelnen Funktionen der Anwendung darstellen.

Soll ein Taschenrechner entwickelt werden, mit dem u.a. die Rechenoperation ‘Addition‘ durchgeführt werden kann, ist folgende Formulierung einer User Story denkbar:

*„Als Nutzer möchte ich zwei Zahlen miteinander addieren können, damit ich leicht die Summer dieser erhalten kann.“*.

Im Three-Amigo-Meeting werden zu User Stories abstrakt formulierte Kriterien aufgestellt, die erfüllt werden müssen, damit die Anwendung für den Auftraggeber „akzeptabel“ ist. Diese werden daher auch „Akzeptanzkriterien“ genannt.

Wünscht sich der Auftraggeber in dem Beispiel, das nur natürliche Zahlen miteinander addiert werden können, wäre dies ein konkretes Akzeptanzkriterium und das folgendermaßen formuliert werden kann:

*„Gibt ein Nutzer eine nicht natürliche Zahl ein, wird eine Fehlermeldung zurückgegeben.“.*

Zu jedem Akzeptanzkriterium werden konkrete Beispiele zum genaueren Verständnis des Verhaltens und zur Dokumentation dessen erstellt. Da die Beispiele gleichzeitig als fachliche Tests und Vorlage für Softwaretests dienen, werden sie „Akzeptanztest“ genannt.

Der Akzeptanztests zu dem beschriebenen Akzeptanzkriterium kann wie folgt formuliert werden:

*„Möchte ein Nutzer die Zahlen 3 und 6,5 miteinander addieren, wird ihm eine Fehlermeldung zurückgegeben, die besagt, dass er nur natürliche Zahlen miteinander addieren kann.“*

Bei der Formulierung von User Stories, Akzeptanzkriterien und Akzeptanztests sowie in jeder anderen Form von Kommunikation zwischen den Projektbeteiligten soll eine „Ubiquitous Language“ (auch „ubiquitäre“ oder „allgegenwärtige“ Sprache genannt) verwendet werden. Dies bedeutet, die einheitliche und eindeutige Verwendung von Fachbegriffen, die spätestens zu Projektbeginn definiert wurden. Fachbegriffe können oftmals uneindeutig oder unklar für die Projektbeteiligten sein. Damit soll einem wesentlichen (Kommunikations-) Problem entgegengewirkt werden. Zum Beispiel hat der Begriff „Prozess“ für Informatiker eine andere Bedeutung, als für Juristen, was bei der Kommunikation zwischen den beiden Gruppen schnell zu Missverständnissen führen kann. Damit Missverständnisse und Unklarheiten dieser Art nicht auftreten, sollen Fachbegriffe, ihre Definition sowie korrekte Verwendung bereits zu Beginn des Projektes allen Beteiligten verinnerlicht werden.

Bei einem Vorgehen nach BDD werden die im Rahmen des Three-Amigo-Meetings erstellten Akzeptanztest nun als automatisch ausführbare Softwaretest realisiert.[[8]](#footnote-8)

Derartige Tests können beliebig oft und ohne Zutun eines Testers ausgeführt werden. Dies ermöglicht Entwicklern neuen Code hinzuzufügen oder ein Refactoring des bereits bestehenden Codes durchzuführen, ohne Gefahr zu laufen unbewusst fachliche Fehler einzubauen. Damit werden vor allem kostspielige und zeitintensive Korrekturen in der Testphase sowie im späteren Betrieb vermieden.

Die Umsetzung erfolgt bevorzugt nach dem „*outside-in*“ Vorgehen. Dabei wird der Test vor dem eigentlichen Code entwickelt, so dass die Ausführung des Tests zunächst einmal fehlschlägt.[[9]](#footnote-10) In der Folge wird die Anwendung soweit entwickelt, bis der Test erfolgreich und damit sichergestellt ist, dass das gewünschte Verhalten ausgeführt wird.

* + 1. Example Mapping

Example Mapping ist eine Methode zur effektiven Gestaltung der Three-Amigo-Meetings, mit der Akzeptanztests zu den Akzeptanzkriterien der User Stories erstellt werden können. Example Maaping verfolgt dabei ein visuelles und intuitives Vorgehen, das lediglich Post-Ist mit vier verschiedenen Farben erfordert.

User Story, Akzeptanzkriterium, Akzeptanztest sowie mögliche Fragen werden jeweils einer eigenen Farbe zugeordnet. Jeder User Story werden ein bis n Akzeptanzkriterien zugeordnet und zu jedem Akzeptanzkriterium jeweils ein bis n Akzeptanztests[[10]](#footnote-12). Ergibt sich bei den Formulierungen eine Frage, die nicht direkt geklärt werden kann, wird für diese ebenfalls ein Post-It der entsprechenden Farbe an die Stelle gehangen, an der sie aufgetreten ist.

Beim Erstellen der Akzeptanztests sollte darauf geachtet werden, dass eine „kombinatorische Explosion“ vermieden wird. Damit ist gemeint, dass zu jedem inhaltlich unterschiedlichen Fall nur ein Akzeptanztest geschrieben wird. Für das Beispiel des Taschenrechners sollten somit nicht mehrere Tests existieren, die überprüfen, ob das richtige Ergebnis bei zwei natürlichen positiven Zahlen berechnet wird. Jedoch sollte das Verhalten aller Grenzfälle („Edge Cases“ bzw. „Key Examples“) berücksichtigt und überprüft werden, wie z.B. die Eingabe der Zahl 0.

Das Ergebnis des Example Mapping ist eine visuelle Übersicht, anhand der mögliche Missstände schnell auffallen können. Hängen z.B. übermäßig viele Post-Its der Farbe der Akzeptanzkriterien an einer User Story, so könnte dies ein Indiz dafür sein, dass die User Story zu groß ist und in mehrere unterteilt werden sollte. Hängen viele Post-Its in der Farbe der Fragen, so kann es sein, dass die Domänen-Experten sich bewusst werden müssen, welches genaue Verhalten er zu der entsprechenden User Story erwartet.

Example Mapping fördert die Kreativität beim Erstellen der Beispiele, wodurch das Verhalten sehr umfangreich und detailliert beschrieben wird und so möglichst alle Unklarheiten beseitigt werden.

Um langen ermüdenden Meetings entgegenzuwirken, sollte das Example Mapping schnell und regelmäßig erfolgen und sich wenn möglich auf die Diskussion einer einzelnen User Story für einen Termin konzentrieren.

* + 1. Abgrenzung BDD zu TDD

BDD hat starke Ähnlichkeiten zu der bekannteren Technik Test-Driven-Development („TDD“), weswegen häufig die Frage gestellt wird, worin der Unterschied besteht.[[11]](#footnote-13) Die Unklarheit entsteht aufgrund derselben Grundidee beider Techniken. Sowohl bei BDD als auch bei TDD gilt es, vor der Implementierung das Verhalten der umzusetzenden Anwendung zu definieren, und dieses mittels Softwaretests zu überprüfen.[[12]](#footnote-14)

Während bei TDD Tests in einer Programmiersprache implementiert werden, die für einen Domänen-Experten ohne technischen Hintergrund nur schwer oder gar nicht verständlich sind, versucht BDD die Zusammenarbeit zwischen Domänen-Experte und Software-Entwickler möglichst effektiv zu gestalten und Testfälle in einer natürlichen Sprache zu beschreiben. Der Fokus auf as gemeinsame Verständnis aller Projektbeteiligter äußert sich dies auch in der Art der Tests. Diese überprüfen die gesamte Anwendung bzw. ein vollständiges Modul und werden Integration- bzw. Ende-zu-Ende-Tests genannt. TDD hingegen ist besonders häufig bei der Realisierung von Unit-Tests vorzufinden.

1. Cucumber
   1. Grundlagen
      1. Zweck

Cucumber ist ein BDD-Framework zur textuellen Spezifikation von Akzeptanzkriterien und automatisierten Überprüfung dieser Beschreibung auf ihre korrekte Implementierung hin.

Ursprünglich wurde Cucumber in der Programmiersprache Ruby als Ergänzung zum BDD-Framework RSpec geschrieben und für das Testen von Ruby-Anwendungen genutzt. Mittlerweile unterstützt Cucumber neben Java und weiteren JVM Sprachen auch C++ und JavaScript. Darüber hinaus gibt es mit Cuke4php, Cuke4Lua und SpecFlow eine Reihe Open-Source Projekte, die das Framework für weitere Programmiersprachen wie z.B. PHP, Lua und C# bereitstellen.

Auf der offiziellen Webseite wirbt das Framework mit dem Slogan „*Our open source tool tests business-readable specifications against your code on any modern development stack*“.[[13]](#footnote-15) Damit wird auf die Möglichkeit verwiesen, Akzeptanztests in einer natürlichen Sprache unter Verwendung der Syntaxsprache Gherkin (vgl. Kapitel 3.1.3) zu formulieren und diese mit Softwaretests zu verbinden.

Durch die Verbindungen entstehen ausführbare fachliche Definitionen der Anwendung - eine „lebende Dokumentation“. Mittels Testautomatisierung können Softwaretests leicht durchgeführt und so sichergestellt werden, dass die Dokumentation aktuell und korrekt ist. Sie stellt somit eine „*source of truth*“ dar, die für sich alleine ausreichen soll, um die ‚beschriebene‘ Anwendung zu verstehen.

Wird ein Akzeptanzkriterium geändert oder ein neues wird hinzugefügt, entstehen zu diesem auch neue Akzeptanztests. Eine Abweichung des beschriebenen Verhaltens zu dem von der Anwendung ausgeführten fällt beim Fehlschlagen der automatisierten Tests sofort auf.

* + 1. Architektur

Cucumber Tests bestehen aus verschiedenen Elementen, die entweder der Erklärung der Fachlichkeit oder der Ausführung der Tests dienen.

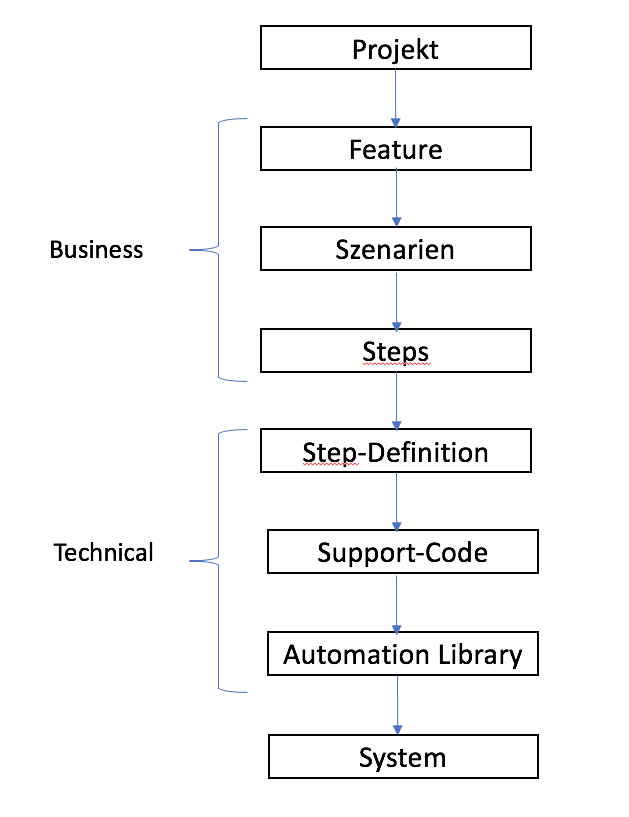


Abbildung 2: Struktur eines Cucumber Projektes

Quelle: Vgl. Rose, Wynne, Hellesoy (2015), S. 7

Abbildung 2 zeigt die Elemente eines Cucumber Tests auf und ordnet diese dem business-bzw. technischen Bereichen zu.

Anwendungen stellen i.d.R. verschiedene Funktionen bereit, die im Rahmen von Softwareprojekten als User Stories formuliert werden. Zu jeder User Story wird eine Feature-Datei mit der Endung „.feature“ erstellt, in der die zur User Story gehörenden Akzeptanzkriterien als Akzeptanztests, in Gherkin „*Szenarien*“, genannt, formuliert werden.

Ein Szenario besteht aus drei Phasen mit jeweils beliebig vielen „Steps“.

1. Die Definition der Ausgangssituation,
2. die Bestimmung des Ereignisses
3. und der Zustand, der nach dem Ereignis eingetreten sein soll.

Feature-Dateien, Szenarien und Steps dienen als fachliche Erklärungen der gewünschten Anwendung und werden daher den business-relevanten Elementen zugeordnet.

Zu jedem Step muss eine zuordenbare Step-Definition implementiert sein. Der Step ist die textuelle Beschreibung eines Zustandes oder eines eintretenden Events. Die Step-Definition ist dessen Implementierung, die den Zustand erstellt bzw. überprüft, oder das Event auslöst. I.d.R. entsteht bei der Implementierung der Softwaretests Code, der nicht direkt mit der Ausführung des Steps zu tun hat, sondern z.B. Konvertierungen vornimmt, oder Zustände unter den Step-Definitionen teilt. Dieser Code wird Support-Code genannt. Damit ein Cucumber-Test automatisiert ausgeführt werden kann, wird oftmals ein Test-Framework wie z.B. JUnit oder TestNG (in Abb. 2 als „Automation Library“ definiert) verwendet.

Step-Definition, Support-Code und Automation Library führen die Cucumber-Tests aus und werden daher den technischen Elementen zugeordnet.

* + 1. Gherkin

Die Syntaxsprache Gherkin ist ein zentraler Bestandteil von BDD, deren Syntaxregeln die Beschreibung von Akzeptanztests in einer natürlichen Sprache ermöglichen. Zu diesem Zweck bietet Gherkin eine einfache Grammatik, um die Tests zu strukturieren. Die Bestandteile der Grammatik werden in Gherkin „*Key Words*“ genannt, deren Verwendung in einer bestimmten Reihenfolge erfolgen muss.[[14]](#footnote-16)

Damit Tests in verschiedenen natürlichen Sprachen geschrieben werden können und trotzdem gut lesbar bleiben, stehen die Key Words nicht nur in Englisch, sondern auch in über 70 weiteren Sprachen wie Deutsch oder Spanisch zur Verfügung. Werden die Szenarien anstelle von Englisch in einer anderen Sprache formuliert, muss dies in jeder Feature-Datei mit dem Header „#language:“ unter Angabe des entsprechenden 2-stelligen ISO-Codes definiert werden.

Ein Bild, das Text, Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 3: Struktur einer Gherkin Feature-Datei

Der Aufbau einer Feature-Datei mit einem Szenario in der Gherkin-Syntax kann in Abbildung 3 anhand des Taschenrechners aus Kapitel 2.2 nachvollzogen werden. Die lila markierten Wörter sind die Gherkin Key Words. Es fällt dabei ins Auge, dass die Key Words stets am Anfang einer neuen Zeile stehen müssen.

In Zeile 1 wird zunächst die Sprache mit dem optionalen „#language:“-Header und dem Kürzel „en“ bestimmt. In der vorliegende Feature-Datei wird somit die englische Version der Gherkin Key Words verwendet.

Die nächste Zeile (hier Zeile 3) muss mit dem Key Word „*Feature*:“ gefolgt von dem Titel der zu definierenden Funktion beginnen. Dieser Zeile können weitere Zeilen ohne beginnendem Key Word folgen (Zeile 4 - 6), die das Feature näher beschreiben.

Das Key Word „*Scenario*:“ (Zeile 8), gefolgt von dem Titel des Szenarios, markiert den Beginn eines neuen Akzeptanztests. Analog zum Feature kann ein Szenario in den nachfolgenden Zeilen näher beschrieben werden.

In den Zeilen 11 bis 14 finden sich die einzelnen Steps die die drei Phasen eines Szenarios abbilden.

Im ersten Step (Zeile 11) wird, mit dem Key Word „*Given*“ ein Step der ersten Phase eingeleitet, in dem der Ausgangszustand beschrieben wird. In der nachfolgenden Zeile (hier 12) fügt das Key Word „*And*“ einen zusätzlichen Step der Phase hinzu, um den vorhergehenden Given-Step weiter zu konkretisieren.

Der dritte Step, beginnend mit dem Key Word „*When*“, gehört der zweiten Phase an und beschreibt das eintretende Ereignis. Mit dem Key Word „*Then*“ zu Beginn von Zeile 14 wird ein Step der dritten Phase eingeleitet, der den erwarteten Endzustand beschreibt.

Wie bereits erwähnt, wird in dem Beispiel das Key Word „*And*“ verwendet, um eine Phase zu erweitern. Dieses und das Key Word „*But*“ wurden in Gherkin eingeführt, um die Lesbarkeit der Szenarien aufrechtzuerhalten. Beide Key Words leiten einen Step ein, der der gleichen Phase entspricht, wie der vorherigen Step.

Mit den in diesem Kapitel vorgestellten Key Words ist es möglich einfache Features und Szenarien zu beschreiben. Gherkin kennt darüber hinaus eine Reihe weiterer Key Words, die in dem Kapiteln 3 erläutert werden. Eine Übersicht über die gesamte Grammatik findet sich im Anhang/in der Quelle xxx.

* 1. Exemplarische Darstellung wichtiger Cucumber-Elemente

Damit redundante Steps und unnötige Wiederholungen vermieden werden, bietet Cucumber eine Reihe an Möglichkeiten, um das Beschreiben von Szenarien und Testen der Anwendung möglichst effizient zu gestalten. Hier sind insbesondere die Key Words „*Background*“ und „*Scenario Outline*“ sowie die Parametrisierung von Step-Definitionen zu nennen. Diese werden nachfolgend vorgestellt.

* + 1. Szenario

Ein Szenario beschreibt das konkrete Verhalten der Anwendung auf ein bestimmtes Ereignis unter einem zuvor definierten Kontext.[[15]](#footnote-17)

Die Ausführung eines Szenarios impliziert die Ausführung aller Step-Definitionen und die Bewertung derer Ergebnisse mit einem Status (vgl. Tabelle 1). Werden alle Step-Definitionen erfolgreich ausgeführt, wird auch das Szenario als erfolgreich (Status PASSED, Priorisierung 0) bewertet. Schlägt jedoch eine einzelne Step-Definition fehl, wird die Ausführung abgebrochen und das Szenario gilt automatisch als fehlgeschlagen (Status FAILED, Priorisierung 5).

Um den Ausgang eines Szenarios zu bewerten wird die ordinal()- Funktion des Enums, in dem die möglichen Status enthalten sind, für jeden Statuswert aufgerufen. Die Funktion überprüft die Rangordnung und somit die Priorisierung der enthaltenen Status. Das Szenario wird mit dem höchsten Status (d.h. dem schlechteste Ergebnis) aller Step-Implementierung bewertet.

|  |  |  |
| --- | --- | --- |
| **Status** | **Priorisierung** | **Bedeutung** |
| PASSED | 0 | Eine zugeordnete Step-Definition konnte ohne Exception ausgeführt werden |
| SKIPPED | 1 | Step wurde bei der Ausführung übersprungen |
| PENDING | 2 | Eine einem Step zuordenbare Step-Definition wurde nicht ausimplementiert |
| UNDEFINED | 3 | Es existiert noch keine zuordenbare Step-Definition zu dem Step |
| AMBIGUOUS | 4 | Es wurden mehr als eine zuordenbare Step-Definition gefunden |
| FAILED | 5 | Bei der Ausführung der Step-Definition wurde eine Exception geworfen |

Tabelle 1: Mögliche Ergebnisse einer Step-Definition

Erhält bei der Ausführung eines Szenarios eine Step-Definition einen anderen Zustand als „PASSED“ werden die nachfolgenden Step-Definitionen des Szenarios übersprungen (d.h. nicht mehr ausgeführt) und als SKIPPED markiert.

* + 1. Step-Definitionen

Eine Step-Definition ist die Implementierung eines Steps und somit die konkrete Ausführung dieses. Jede Step-Definition wird in Java als eigene Methode abgebildet (vgl. Abb 4).

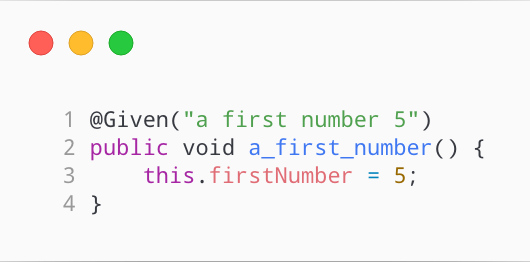


Abbildung 4: Step-Definitions Beispiel

Mithilfe entsprechender Java Annotationen werden die Methoden für Cucumber als Step-Definition erkennbar gemacht. Cucumber liefert zu jedem Gherkin Step Key Word eine passende Annotation (vgl. Tabelle 2).

|  |  |
| --- | --- |
| **Gherkin Key Word** | **Cucumber Annotation** |
| Given | @Given |
| When | @When |
| Then | @Then |
| And | @And |
| But | @But |

Tabelle 2: Cucumber Annotationen zu den Gherkin Key Words

Um eine Step-Definition einem Step eindeutig zuordnen zu können, muss die Step-Definition Annotation zusätzlich einen String als Wert übergeben bekommen. Dieser wird als Cucumber Expression (vgl. Kapitel xxx) bezeichnet und per Pattern Matching mit dem Step auf Übereinstimmung überprüft.

* + 1. Scenario Outline

Oftmals haben Feature-Dateien eine Reihe an Szenarien, die inhaltlich dasselbe beschreiben, dabei jedoch unterschiedliche Werte verwenden. Grund dafür ist häufig, dass das Verhalten der Anwendung bei unterschiedlichen Grenzwerten beschrieben werden muss.

In dem Beispiel aus Kapitel 2.2 soll die Addition von natürlichen Zahlen möglich sein und bei einer falschen Eingabe eine Fehlermeldung zurückgeliefert werden.

Eine solche wäre somit, dass eine oder beide Zahlen negative oder reelle Zahlen sind. Unabhängig von der Eingabe ist der Ablauf der gleiche. Es erfolgt eine nicht valide User Eingabe, die zu einer Fehlermeldung führen soll. Damit nicht zwei Szenarien geschrieben werden müssen, kann anstelle des bereits bekannten Szenarios das Gherkin Key Word Scenario Outline (vgl. Abb. 5) verwendet werden.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 5: Szenario Outline Beispiel

Die Abbildung zeigt, wie mit einer Scenario Outline ein inhaltlich gleiches Szenario für unterschiedliche Werte erstellt werden kann. Der Aufbau ist fast identisch zu dem eines normalen Szenarios. Die Steps können jedoch Platzhalter (werden durch spitze Klammern kenntlich gemacht, vgl. Zeile 6 und 7) verwenden, die mit Werten aus einer „Examples“-Tabelle ersetzt werden.

Die Examples-Tabelle muss unterhalb des letzten Steps definiert werden und für jeden Platzhalter eine Spalte mit entsprechender Überschrift beinhalten. Bei der Ausführung wird für jede Zeile der Tabelle ein eigenes Szenario erstellt.

Um Gruppierungen verschiedener Wertkombinationen vorzunehmen, können die unterschiedlichen Ausprägungen in verschiedenen Example-Tabellen logisch zusammengefasst werden – für das Beispiel aus Kapitel 2.2 hätte eine zusätzliche Tabelle für Wertkombinationen mit negativen Zahlen und eine mit reellen Zahlen erstellt werden können. Die Ausführung ändert sich dadurch nicht, da auch in diesem Fall für jede Zeile aller Tabellen ein eigenes Szenario erstellt wird. Die Scenario Outline kann so jedoch nachvollziehbarer und übersichtlicher aufgebaut werden.

* + 1. Parameter übergeben

Da in den Szenarien oftmals gleiche Steps mit unterschiedlichen Werten vorkommen können diese Werte aus dem Step extrahiert werden und als Parameter an die Step-Definition übergeben werden. Dies geschieht per Pattern Matching.

Ein Parameter wird in einer Step-Definition als Pattern (z.B. einer Captured Group) definiert.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 6: Parameter Übergabe mit Captured Groups

Abbildung 6 zeigt ein Beispiel, in dem die Step-Definitions Annotation einen regulären Ausdruck mit einer Captured Group übergeben bekommt. Die Captured Group ([\\d](file://d)+) definiert, dass im Step eine n-stellige ganze zahl enthalten sein soll. Ist dies der Fall kann Cucumber den Step der im Beispiel gezeigten Step-Definition zuordnen, liest die Zahl aus und übergibt sie als Parameter an die Methode. Ist stattdessen eine alphanumerische Zeichenkette oder eine Fließkommazahl enthalten, kann Cucumber den Step nicht zuordnen und liefert einen Fehler.

Da reguläre Ausdrücke die Lesbarkeit eher erschweren, wurden in der Version 3.0.0 Cucumber Expressions eingeführt, um erstere zu ersetzen. Cucumber Expressions können anstelle der Captured Groups „Output Parametern“ (durch geschweifte Klammern kenntlich gemacht) enthalten. Dies wird am Beispiel eines Systems zum Speichern von durchgeführten Flügen deutlich.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 7: Flight Gherkin Step

Das Feature aus Abbildung 7 enthält ein Szenario mit einem Step, der einen durchgeführten Flug beschreibt. Im Step sind eindeutige Kürzel für den Start- und Zielflughafen, das Flugdatum sowie die Anzahl an Passagiere enthalten.

Ein Bild, das Screenshot enthält.

Automatisch generierte BeschreibungAbbildung 8: Flight Step-Definition mit regulärem Ausdruck

Abbildung 8 zeigt die Step-Definition unter Verwendung regulärer Ausdrücke. Um die relevanten Daten aus dem Step als Parameter an die Step-Definition zu übergeben, werden mehrere teils sehr komplexe Captured Groups definiert.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 9:Flight Step-Definition mit Cucumber Expression

Abbildung 9 zeigt die Step-Definition unter Verwendung einer Cucumber-Expression. Diese ermöglicht ein einfacheres und fachlicheres Lesen im Vergleich zu den regulären Ausdrücken. Als Parameter können der Step-Definition nicht nur einzelne Strings, sondern auch eigene Datentypen z.B. vom Typ „Flight“ und „Month“ übergeben werden. Um dies zu ermöglichen müssen die entsprechenden Typen in der Type Registry (vgl. Kapitel 3.2.8) registriert werden. Diese kennt bereits eine ganze Reihe von vordefinierten Output Parametern für die gängigsten Datentypen (vgl. Tabelle 3).

|  |  |
| --- | --- |
| **Output-Parameter** | **Typ** |
| {int} | Integer |
| {string} | String |
| {word} | Ein Wort |
| {float} | Float |
| {byte} | Byte |
| {short} | Short |
| {double} | Double |
| {long} | Long |
| {bigdecimal} | BigDecimal |
| {biginteger} | BigInteger |

Tabelle 3: Built-in Output Parameter

* + 1. Doc Strings

Doc Strings bieten die Möglichkeit Texte, die zu lang für eine einzelne Zeile sind, über mehrere zu beschreiben.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 10: Doc String Beispiel

Sie werden durch drei aufeinanderfolgende Anführungszeichen, direkt unterhalb eines Steps, geöffnet und auch wieder geschlossen.[[16]](#footnote-18) Längere Texte, wie z.B. der Inhalt einer E-Mail, oder einer JSON- Datei können so beschrieben werden, ohne den Step unübersichtlich zu machen. Der enthaltene Text wird als String Parameter an die zugehörige Step-Definition übergeben.

Das Beispiel in Abbildung 10 zeigt einen Doc String, der den Inhalt einer E-Mail beschreibt. Der Text in den Zeile 8 und 9 wird hier als ein String Parameter an eine zuordenbare Step-Definition übergeben.

* + 1. Data Tables

Neben Doc Strings bieten Data Tables eine weitere Option einen Step über mehrere Zeilen auszudehnen und so größere Datenmengen in Form einer Tabelle zu beschreiben.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 11: Data Table Beispiel

Während der Aufbau analog zu einer Examples-Tabelle ist[[17]](#footnote-19), erhält die entsprechende Step-Definition einen Parameter vom Typ „DataTable“ in dem die beschriebenen Daten enthalten sind.

Die Data Table Klasse bietet eine Reihe an Methoden, um die enthaltenen Daten zu manipulieren und auszulesen.[[18]](#footnote-20) Über die bereits in Kapitel 3.2.4 erwähnte Type Registry besteht die Möglichkeit DataTable-Typen zu registrieren, sodass die Daten der Tabelle zu einem Objekt transformiert werden, welches an die Step-Definition als Parameter übergeben wird (vgl. Kapitel 3.2.8).

* + 1. Background

Da einzelne oder auch mehrere Szenarien innerhalb einer Feature-Datei häufig die gleichen Steps beinhalten, können diese in einen Background-Block verlegt werden.[[19]](#footnote-21) Damit bleiben sowohl der einheitliche Ablauf sowie die wesentlichen Unterschiede der einzelnen Szenarien ersichtlich.

Ein Background-Block wird durch das gleichnamige Gherkin Key Word, gefolgt von einem erklärenden Titel, eingeleitet und kann beliebig viele Steps enthalten, die am Anfang eines jeden Szenarios ausgeführt werden.[[20]](#footnote-22)

Ein Bild, das Screenshot, Text enthält.

Automatisch generierte Beschreibung

Abbildung 12: Background Beispiel

Im vorliegenden Beispiel (vgl. Abb. 12) wird bei der Ausführung beider Szenarien zunächst der Background Step „*Given a first integer 5*“ (Zeile 4) ausgeführt. Je Feature Datei kann nur ein Background-Block enthalten sein. Dieser wird für alle enthaltenen Szenarien ausgeführt.

Mit der sechsten Version von Gherkin wurde das Key Word „Rule“ eingeführt, über das Szenarien gruppiert werden können, die ein Akzeptanzkriterium beschreiben sollen. Jeder Gruppe kann ein eigener Background-Block zugeordnet werden, womit ein Background-Block nicht mehr für alle Szenarion einer Feature-Datei gelten.[[21]](#footnote-23) Das Key Word „*Rule*“ ist derzeit noch nicht für Cucumber-JVM[[22]](#footnote-24) implementiert, soll aber in künftigen Versionen bereitgestellt werden.

* + 1. Type Registry

Die Type Registry wurde im Rahmen der Einführung von Cucumber Expressions bereitgestellt. Mit ihr lassen sich Output Parametern für eigene Datentypen definieren, was insbesondere die Testimplementierung erleichtern soll.

In dem Beispiel aus Abbildung 9 (vgl. Kapitel 3.2.4) wird der Cucumber Annotation eine Cucumber Expression mit zwei selbsterstellten Output-Parametern (vom Typ „Flight“ und „Month“) übergeben. Diese müssen zuvor in der Type Registry registriert werden.

Ein Bild, das Screenshot enthält.

Automatisch generierte BeschreibungAbbildung 13: Registrierung eines Output Parameters

In Abbildung 13 wird am Beispiel des Output Parameter „flight“ gezeigt, wie die Klasse *Flight* in der Type Registry registriert werden kann. Hierfür muss das Interface „TypeRegistryConfigurer“ aus dem Package „*cucumber.api*“ implementiert werden.

Bei der Konfiguration des Cucumber-Tests (vgl. Kapitel 3.3.2) muss das Package der implementierenden Klasse Cucumber mitgeteilt werden. Bei der Testdurchführung durchsucht Cucumber das Package nach einer Klasse, die dieses Interface implementiert und verwendet diese anstelle der Default-Implementierung um die Type Registry zu konfigurieren.

Die Methode „configureTypeRegistry“ des Interfaces bekommt die Type Registry als Parameter übergeben. Diese enthält wiederum weitere Methoden, mit der die Parameter-Typen registriert werden können (vgl. Abbildung 13 und 15).

Um die Daten einer Data Table als Typen an die Step-Definition zu übergeben, können ebenfalls entsprechende Data-Table- Typen registriert werden (vgl. Abbildung 15).

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 14: Flight Data Table

Ein Bild, das Screenshot enthält.

Automatisch generierte BeschreibungAbbildung 15: Registrierung eines Data Table Typens

Ein Bild, das Screenshot enthält.

Automatisch generierte BeschreibungAbbildung 16: Verwendung eines registrierten Data Table Typens

Bei der Ausführung des Szenarios aus Abbildung 14 würde für jede Zeile, mit Ausnahme der Überschriften-Zeile, aus der Data Table die Lambda Funktion aus Abbildung 14 aufgerufen werden, sodass die Step-Definition aus Abbildung 16 eine Liste an Flügen als Parameter übergeben bekommt.

* + 1. Hooks

Hooks sind Methoden, die vor- bzw. nach jedem Szenario ausgeführt werden. Dazu können Methoden mit den von Cucumber mitgelieferten Annotationen „@Before“, bzw. „@After“ annotiert werden[[23]](#footnote-25) – nicht zu verwechseln mit den von JUnit gelieferten Annotation @Before und @After.

Mit Cucumber v3.0.0 wurden zusätzlich die Annotationen „@BeforeStep“ bzw. „@AfterStep“ eingeführt. Damit ist es möglich Code vor- bzw. nach der Durchführung eines jeden Steps auszuführen.[[24]](#footnote-26)

Ergänzend dazu können den Cucumber Hook Annotationen Tags als String-Array übergeben werden, um den Aufruf des Hooks einzuschränken. Um ein Szenario zu taggen wird in der Zeile oberhalb des „Szenario“- Key Words ein „@“-Zeichen mit einem beliebigen Begriff geschrieben. Enthält eine Hook-Annotation ein solches Tag wird die annotierte Methode nur bei Szenarien ausgeführt, die entsprechend getaggt sind.[[25]](#footnote-27)

* 1. Arbeitsweise
     1. Aufruf eines Cucumber Tests

Cucumber Tests können auf verschiedene Weisen angestoßen werden. Als Standalone-Variante steht im Package „cucumber.api.cli“ ein Command Line Interface („*CLI*“) bereit, um Tests auf der Kommandozeile ausführen zu können.

In der Regel wird Cucumber-JVM jedoch in Verbindung mit Test-Frameworks wie JUnit oder TestNG verwendet, um Tests automatisiert ausführen zu können. Für beide Frameworks stellt Cucumber-JVM eigene Module bereit.

Um einen Cucumber-Test über JUnit auszuführen, enthält Cucumber-JVM einen eigenen JUnit-Runner. Dieser wird über die von JUnit-Annotation @RunWith an der Testklasse registriert. Für TestNG steht die abstrakte Oberklasse AbstractTestNGCucumberTests bereit, von der eine Testklasse erben muss.

* + 1. Konfiguration der Testausführung

Cucumber stellt eine Reihe von Optionen bereit (vgl. Anhang xxx), um die Testausführung zu konfigurieren. Wird ein Test über die CLI ausgeführt, kann die Konfiguration per Command Line Arguments vorgenommen werden. Die übergebenen Parameter werden geparst und in einer Instanz der Klasse „RuntimeOptions“ bereitgestellt. Erfolgt die Testausführung hingegen mit JUnit oder TestNG, wird der Test über die Annotation „@CucumberOptions“ konfiguriert.

Die wichtigsten Konfigurations-Parameter für einen Cucumber-Test sind *feature*, *glue*, *name*, *tags* und *plugin*:

* **feature**: Über den Parameter wird Cucumber-JVM mitgeteilt, in welchen Verzeichnissen sich die Feature-Dateien befinden. Die Angabe erfolgt über ein String-Array.
* **glue**: Mit dem „Parameter, können Java-Packages als String-Array angegeben werden, die die für die Testausführung relevanten Klassen (nachfolgend „glue-Klassen“ genannt) enthalten. Zu diesen zählen z.B. Klassen mit enthaltenen Step-Definitionen und die implementierende Klasse des „TypeRegistryConfigurer“ Interfaces.
* **name/tags**: Über die Parameter „name“ und „tags“ kann nach bestimmten Szenarien gefiltert werden, um bei Bedarf nur eine Teilmenge auszuführen. Der „name“-Parameter enthält die Titel von Szenarien, der „tags“-Parameter nimmt die Namen von Tags entgegen. Beide Parameter können einen oder mehrere Werte entgegennehmen, nach denen die Szenarien gefiltert werden.
* **plugins**: Die Ausgabe der Test-Ergebnisse kann mithilfe von Plugins beeinflusst werden. Hierzu stellt Cucumber eine Reihe built-in Plugins bereit. Um ein Plugin bei der Ausführung eines Cucumber-Tests anzumelden, werden dem Parameter die Namen als String-Array übergeben.
  + 1. Testvorbereitung

Bei der Ausführung eines Cucumber-Tests wird zuerst die Konfigurationen für die Testausführung ausgelesen. Dies geschieht durch Erstellen einer Instanz von „RuntimeOptions“. Im Konstruktor wird eine Liste an Strings geparst und in entsprechenden Attributen gespeichert. Beim Nutzen der CLI werden die Command Line Arguments an den Konstruktor übergeben.

Bei der Testausführung mit JUnit bzw. TestNG wird die Konfiguration ( @CucumberOptions) über die RuntimeOptionsFactory ausgelesen. Als Ergebnis wird eine Instanz der Klasse RuntimeOptions zurückgeben, welche für die Instanziierung der Klasse „*Runtime*“ - die zentrale Klasse eines jeden Cucumber-Tests - genutzt wird.

Mit dem Aufruf des Konstruktors wird die Testdurchführung vorbereitet. Nach der Konfiguration der Type Registry wird eine „Runner“ Instanz erstellt, die in den glue-Klassen nach Step-Definitionen und Hook-Methoden sucht.

* + 1. Testausführung

Ist die Testvorbereitung abgeschlossen, wird die Ausführung der Runtime-Instanz angestoßen. Zunächst werden alle angegebenen Feature-Dateien geparst und für jede Datei eine Instanz der Klasse „CucumberFeature“ erstellt. Im Anschluss werden die für die Ausgaben zuständigen Plugins geladen. Die Ausgaben erfolgen als Reaktion auf Events, die Cucumber bei der Testausführung zu unterschiedlichen Zeitpunkten, z.B vor und nach der Ausführung eines Szenarios, auslöst. Über einen Event-Handler (eine Instanz der Klasse „EventPublisher“) werden die Plugins über für sie relevante Events informiert und können eine entsprechende Ausgabe durchführen.

Sind alle Plugins geladen, wird zu jedem CucumberFeature ein TestCase erstellt. In diesem Schritt werden die glue-Klassen instanziiert und Step zu Step-Definition zugeordnet.

Die Ausführung eines Cucumber-Tests erfolgt in der beschriebenen Reihenfolge:

1. Before-Hooks
2. Matches: Jeder Match enthält wiederum Listen an auszuführenden Methoden:
   1. BeforeStep-Hooks
   2. Step-Definitionen
   3. AfterStep-Hooks
3. After-Hooks

Wird für einen Step keine passende Step-Definition gefunden, gibt Cucumber eine zu dem Step passende Step-Definition aus. Diese wirft zunächst eine PendingException, um darauf hinzuweisen, dass sie noch ausimplementiert werden muss.

Alle ausgeführten Methoden, auch die Hooks, erhalten ein Ergebnis (vgl. Kap. 3.2.1 Tabelle 1). Wird ein Step anders als mit PASSED bewertet, werden alle weiteren Steps des Szenarios als SKIPPED bewertet und übersprungen. Eine Ausnahme bilden die optionalen „AfterStep“- und „After“-Hook, die in jedem Fall ausgeführt werden.

Nachdem alle TestSteps ausgeführt wurden wird das Szenario an sich bewertet (vgl. Kap. 3.2.1) und alle Instanzen der glue-Klassen dem Garbage Collector zugeführt. Dieses Vorgehen geschieht iterativ, bis alle Szenarien aller Feature-Dateien ausgeführt wurden.

* 1. Best Practices
     1. Vorgehen

Cucumber unterstützt das Vorgehen nach BDD (vgl. Kapitel 2.2). Wurden im Rahmen eines Three-Amigo Workshops bereits Szenarien erstellt, empfiehlt es sich diese in Feature-Dateien abzulegen. Anschließend kann Cucumber genutzt werden, um die noch nichtexistierenden Step-Definitionen als „leere“ (wirft beim Aufruf lediglich eine PendigException) Methoden erstellen zu lassen (vgl. Kapitel 3.3.4). Diese werden in einem Terminal ausgegeben und können in eine beliebige Klasse kopiert und dort ausimplementiert werden.

* + 1. Regressionsfähige (unabhängige) Szenarien

Jedes Szenario definiert seinen eigenen Anfangszustand, das auszuführende Event und den gewünschten Endzustand. Dadurch ist sichergestellt, dass Szenarien unabhängig voneinander sind und nicht von der Ausführung anderer Szenarien beeinflusst werden.

Ein Szenario ist regressionsfähig, wenn es unabhängig von der Anzahl der eigenen Ausführungen sowie unabhängig von der Änderung, Löschung, Ausführung bzw. nicht Ausführung anderer Szenarien, immer dasselbe Ergebnis liefert.

Die Zustände, auf die die Step-Defintionen zugreifen, sollten somit in jedem Szenario neu gesetzt, bzw. initialisiert werden.

Cucumber unterstützt dies indem es für jedes Szenario neue Instanzen der glue- Klassen erstellt und so keine Zustände innerhalb der glue-Klassen erhalten bleiben – dies gilt jedoch nicht für static Argumente und der Anwendung an sich.

Damit die Szenarien wirklich unabhängig voneinander sind, müssen auch die Zustände der zu prüfenden Anwendung gesetzt werden. Die Steps der Given- Phase müssen dafür den gesamten relevanten Kontext beschreiben. Gerade bei Tests, die Daten aus einer Datenbank überprüfen ist es notwendig im Vorfeld diese korrekt zu setzen.

* + 1. Szenarien wie Geschichten beschreiben

Die Szenarien der Features stellen konkrete Beispiele dar, die das Verhalten des Systems beschreiben. Damit diese leicht zu verstehen und nachzuvollziehen sind, sollten sie wie kleine Geschichten aufgebaut sein, die eine Funktion oder einen Teil einer Funktion beschreiben.[[26]](#footnote-28) Technische Formulierungen sollten für das gemeinsame Verständnis vermieden werden.

* + 1. Genaues Wording bei den Steps

Bei größeren Anwendungen kann es schnell dazu kommen, dass in mehreren Szenarien ähnlich formulierte Steps vorkommen, mit ihnen jedoch nicht das gleiche gemeint ist.

Existieren auf einer Webseite verschiedene Buttons, wäre die Formulierung „When the client clicks the button“ nicht eindeutig. Der Step könnte an verschiedenen Stellen vorkommen, sich jedoch auf unterschiedliche Buttons beziehen. Da jedoch immer dieselbe Step-Definition ausgeführt wird, würde sich der Test anderes als ursprünglich vorgesehen verhalten. Um Unklarheiten und Verwirrungen von vornherein zu verhindern, sollte daher eine exakte und eindeutige Formulierung für jeden Step gewählt werden – z.B. „When the client clicks the Log In Button“.

* + 1. Step-Definitionen Klassen übersichtlich halten

Selbst kleinere Anwendungen stellen i.d.R. mehrere Funktionen breit. Um die Step-Definition einfach finden zu können, bietet es sich an, Step-Definitionen logisch auf verschiedene Klassen aufzuteilen. Damit sollen übergroße und schlecht verständliche („Messy“) Klassen vermieden werden. Stattdessen sollte das „Single responsibility principle“ eingehalten werden und nur Step-Definition zu einer konkreten Funktion in einer Klasse gebündelt werden.[[27]](#footnote-29) [[28]](#footnote-30)

* + 1. Szenarien übersichtlich halten

Szenarien dienen dem gemeinsamen Verständnis von Domänen-Experten, Entwicklern und Testern. Um ein gemeinsames Verständnis leicht erlangen zu können und zu fördern, sollen Szenarien möglichst kurz und übersichtlich sein. Cucumber selbst empfiehlt nicht mehr als 3-5 Steps pro Szenario zu verwenden.[[29]](#footnote-31)

Benötigt die Beschreibung eines Akzeptanzkriteriums zu viele Steps, um z.B. den Anfangszustand zu beschreiben oder mehrere verschiedene Zustände abzufragen, sollte versucht werden, dieses Szenario in kleinere Szenarien aufzuteilen.

Grundsätzlich sollte jedes Szenario nur einer Erklärung dienen. Werden verschiedene Aspekte mit einem Szenario erklärt, kann es schnell unübersichtlich und unnötig komplex werden.

* + 1. Verwendung von Hooks

Die Verwendung von Hooks sollte nach Möglichkeiten vermieden werden, da der ausgeführte Code nicht in den Szenarien ersichtlich ist. Das Ziel von BDD ist es, die Zusammenarbeit zwischen Domänen- Experten und Entwicklern zu verbessern und Missverständnisse zu vermeiden. Stattdessen sollte versucht werden Steps die bei jedem Szenario eines Features ausgeführt werden sollen, über das Background Key-Word einer Feature-Datei abzubilden.[[30]](#footnote-32)

Hilfereich sind Hooks um Voraussetzungen zu schaffen, die für die Testausführung erforderlich sind, jedoch unerheblich für das Verständnis des Szenarios sind. Dazu gehört z.B. das Starten eines Servers oder den Aufbau einer Datenbankverbindung.

* + 1. Zustände teilen

Der zu testende Zustand ist für alle Step-Definitionen relevant. Sie bauen den Anfangszustand auf, manipulieren Daten durch das Auslösen des Events, oder überprüfen, ob der gewünschte Endzustand eingetreten ist. Da die Step-Definitionen jedoch in unterschiedlichen Klassen liegen können, reicht es i.d.R. nicht, die Zustände in Klassen-Variablen zu definieren. Nötig ist es den Zustand über mehrere Klassen hinaus zu teilen.

Ein static Singleton- Objekt eignet sich nicht zum Halten der Zustände. Es hätten zwar alle Step-Definitionen Klassen Zugriff auf ein gemeinsames Zustandshaltendes Objekt, jedoch würde dieses beim Ausführen eines neuen Szenarios nicht neu instanziiert werden.

Mit Dependency- Injection (DI) Frameworks können mehrere Objekte auf die gleiche Instanz einer Klasse Zugriff haben.[[31]](#footnote-33) Es bietet sich somit an, das Zustandshaltende Objekt mit einem solchen Framework den Step- Definitions-Klassen zugänglich zu machen. Im Rahmen von Cucumber stößt man dabei oft auf das Framework „PicoContainer“. Es erfolgt eine Injection eines Singleton Objektes als Konstruktor- Parameter der Step-Definitionen Klassen.[[32]](#footnote-34) Bei der Konfiguration des Cucumber-Tests muss der package- Name der Zustandshaltenden Klasse dem glue- Parameter mitgeteilt werden. So wird auch dieses Objekt bei der Ausführung eines neuen Szenarios neu instanziiert.

1. Exemplarische RESTful Beispielanwendung

In dem folgenden Kapitel wird das Beispielprojekt „twttr-service“ vorgestellt. Dabei handelt es sich um einen Klon des Kurznachrichtendienstes Twitters, mit dem über eine REST-Schnittstelle kommuniziert werden kann. Um die korrekte Umsetzung der Funktionen zu prüfen, wurden im Vorfeld der Implementierung User Stories und Akzeptanzkriterien definiert und mit Hilfe von Cucumber als Akzeptanztests umgesetzt.

Eine vollständige Dokumentation der REST API, sowie der Architektur des Systems finden sich in Form der Open API Definition sowie eines Klassendiagramms im Anhang. Der Quellcode steht auf GitHub in dem öffentlichen Repository xxx zur Verfügung.

* 1. API und UseCases

Der Twitter-Klon Twttr stellt analog zum Original eine grundlegende Funktion zum Veröffentlichen von Kurznachrichten sowie zur Verbindung mit anderen Benutzern bereit. Damit kein Nutzer, im Namen eines anderen, Nachrichten versenden kann, sind sämtliche Funktionen über token-basierte Authentifizierung abgesichert. Als Technologien kommen an dieser Stelle der Single-Sign-On Server JBoss Keycloak, das Authentifizierungsverfahren OpenID\_connect sowie JSON Web Token zum Einsatz.

Der Twttr-Klon ist über die URL „[http://localhost:8080/twttr-service](http://localhost:8080/twttr-service/api)“ erreichbar. Um die bereitgestellte Schnittstelle einfach verstehen und testen zu können, stellt der Twttr-Klon die Webanwendung Swagger-UI unter der URL xxx bereit, die die API Dokumentation optisch ansprechend aufbereitet.

Nachfolgend werden die implementierten Funktionen exemplarisch vorgestellt, anhand derer die Verwendung von Cucumber demonstriert wird.

|  |  |  |  |
| --- | --- | --- | --- |
| **Titel** | **URL** | **HTTP-Methode** | **Beschreibung** |
| Neuen Tweet erstellen | /tweets | POST | Der übergebene Text wird als neuer Tweet des anfragenden Users veröffentlicht |
| Neuste Tweets erhalten | /tweets | GET | Liefert die neusten Tweets von Usern, denen der anfragende User folgt |
| Tweet Details erhalten | /tweets/{tweetId} | GET | Liefert zusätzliche Informationen zu dem gewählten Tweet |
| Tweet canceln | /tweets/{tweetId} | DELETE | Der gewählten Tweet wird in den Status „CANCELED“ versetzt |
| Tweet liken | /tweets/{tweetId}/like | POST | Der anfragende User wird als „Liker“ des gewählten Tweets vermerkt. |
| Liker ansehen | /tweets/{tweetId}/like | GET | Liefert Liste mit Usern, die den gewählten Tweet geliked haben zurück |
| Unlike Tweet | /tweets/{tweetId}/like | DELETE | Der anfragende User wird nicht mehr als „Liker“ des gewählten Tweets vermerkt |
| Retweet Tweet | /tweets/{tweetId}/retweets | POST | Für den anfragenden User wird ein Retweet des gewählten Tweets erstellt |
| Get Retweeter | /tweets/{tweetId}/retweets/authors | GET | Liefert eine Liste mit Usern, die den gewählten Tweet retweetet haben zurück |
| Get Users | /users | GET | Eine Liste von Usern wird zurückgegeben |
| Follow User | /users/{userId}/follower | POST | Der anfragende User wird als Follower des gewählten Users vermerkt |
| Get Follower | /users/{userId}/follower | GET | Liefert eine Liste mit Usern, die als Follower des gewählten Users vermerkt sind zurück |
| Unfollow User | /users/{userId}/follower | DELETE | Der anfragende User wird nicht mehr als Follower des gewählten Users vermerkt |
| Get Tweets | /users/{userId}/tweets | GET | Liefert eine Liste mit veröffentlichten Tweets des gewählten Users zurück |

Tabelle 4: Implementierte Funktionen des twttr-service

* 1. Technischer Hintergrund

Die Java Webanwendung twttr-service wurde mit dem Microservice-Framework Thorntail in der Version 2.3 realisiert und stellt eine RESTful API auf Basis des Java Enterprise Standards JAX-RS bereit. Als Datenbank wird die In-Memory Datenbank H2 in der Version xxx verwendet. Für das Buildmanagement kommt Apache Maven in Version 3.5.2 in Verwendung.

Im Unterschied zu klassischen Webanwendungen, die als WAR bereitgestellt werden, nutzt Thorntail das Konzept des uberjars (auch Fat-JAR genannt) und stellt selbstständig ausführbare Java Anwendungen bereit. Das JAR enthält neben der eigentlichen Anwendung und allen nötigen Abhängigkeiten auch eine Laufzeitumgebung (JBoss Undertow), die beim Start der Anwendung gestartet wird und in diese sich die Anwendung selbst deployed.

Der Keycloak-Server wurde als Abhängigkeit eingebunden und ist damit ebenfalls Teil des uberjar. Keycloak wird beim Start der Laufzeitumgebung noch vor der Anwendung deployed.

Für die Cucumber-Tests werden neben Cucumber-JVM und JUnit die Testframeworks REST Assured zum Initiieren und Prüfen von HTTP-Anfragen gegen eine datenzentrierte Schnittstelle sowie DbUnit zur Initialisierung und Verifikation der Datenbank genutzt.

* 1. Tests ausführen

Um die erstellen Tests ausführen zu können, muss sowohl die Datenbank als auch das uberjar gestartet werden. Die .jar der Datenbank befindet sich im Ordner „twttr-service“. Das uberjar wird bei Ausführung von Maven in der Phase *package* erstellt und standardmäßig im *target*-Verzeichnis abgelegt.

* 1. Vorstellung der Features

Die im Rahmen der Beispielanwendung erstellten Feature-Dateien enthalten Szenarien zum Testen der Schnittstelle des twttr-Service.

In diesem Kapitel werden der Aufbau und etwaige Besonderheiten erläutert. Die Feature-Dateien sind dem Ordner „src/test/resources/features“ der Beispielanwendung zu entnehmen.

* + 1. Package Aufbau

Die Cucumber-Testklassen sowie die zugehörigen Glue-Klassen befinden sich parallel zu den Schnittstellen des twttr-service im Package „de.openknowledge.playground.api.rest“. Die Klassen sind je nach Zugehörigkeit auf Sub-Packages runner, stepDefinitions und supportCode verteilt. In dem Sub-Package „runner“ sind die Cucumber-Testklassen enthalten. Zu jeder Feature-Datei existiert eine Testklasse, sowie eine zusätzliche, die sämtliche Feature-Dateien überprüft.

Das Package „stepDefinitions“ enthält sämtliche Step-Definitionen, die den Steps der Szenarien zugeordnet werden können. Im Fall der Beispielanwendung wurde für jedes Feature eine eigene Klasse mit Step-Definitionen implementiert. Step-Definition die in mehr als einer Feature-Datei verwendet werden, sind in der Klasse „GeneralSteps“ gebündelt.

Im Package „supportCode“ befinden sich, dem Namen entsprechend, Hilfs-Klassen wie die Klasse „Converter“ (implementiert das Interface „TypeRegistryConfigurer“) sowie die zustandshaltende Klasse „SharedDomain“.

* + 1. Feature-Datei

Am Anfang jeder Feature-Datei werden die zugrundeliegenden Akzeptanzkriterien als Kommentar aufgeführt, die in der entsprechenden Datei durch die Szenarien erklärt und überprüft werden. Dies ist keine gängige Konvention, dient aber dennoch der Vollständigkeit der Dokumentation und dem Verständnis des Lesers.

Mit zukünftiger Erweiterung um das Gherkin Key Word „Rule“ wird diese Art der Erläuterung überflüssig, da Szenarien dann auch nach Akzeptanzkriterien gruppiert werden können.

* + 1. Technische Begriffe

Die Beispielanwendung stellt eine RESTful API bereit, mit der Client-Anwendungen über das HTTP-Protokoll kommunizieren können. Als Datenaustauschformat kommt JSON zum Einsatz.

Da die RESTful API eine datenzentrierte Schnittstelle bereitstellt, werden in den Szenarien HTTP Methoden, URIs und Statuscodes sowie Payloads verwendet, um Anfragen (Requests) und Antworten (Responses) gegen einzelne API Operationen zu beschreiben. Dies hat zur Folge, dass die Szenarien komplexer werden und sich nicht mehr wie Geschichten lesen lassen.

Um in diesem speziellen Kontext ein gemeinsames Verständnis der Domänen-Experten und Entwickler zu erreichen, muss auch auf Seiten der Domänen-Experten Grundwissen zum HTTP-Protokoll vorhanden sein.

Um das Verständnis zu erhöhen sollten die technischen Aspekte, die ein Szenario für den Domänen-Experten nur schwer verständlich machen, soweit wie möglich in Kommentare ausgelagert (vgl. Abbildung xxx) und in einer zusätzlichen API Dokumentation bereitstellt werden.

* + 1. Cucumber-Testklasse

Die Cucumber-Testklassen haben einen einheitlichen Aufbau (Naming, Struktur), der anhand der Testklasse „CreateTweetIT“ erläutert wird.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 29: Cucumber Testklasse "Create tweet" Feature

Cucumber-Tests die gegen eine laufende Anwendung ausgeführt werden, zählen zur Klasse der Integrations-Test. In Java-Anwendungen wird aufgrund der unterschiedlichen Ausführungszeiten (wenige Millisekunden, mehrere Sekunden) bei der Testentwicklung insbesondere zwischen Unit-Tests und Integration-Tests unterschieden. Da Unit-Tests in der Entwicklung immer wieder ausgeführt werden (sollen), muss die Ausführungszeit so kurz wie möglich sein. Ein Integration-Test hingegen benötigt eine laufende Anwendung, die vor der Testausführung gestartet werden muss – was schon mal einige Sekunden (oder länger) in Anspruch nimmt. Da die Ausführungszeit eines Integration-Tests im Vergleich zum Unit-Tests mindestens um den Faktor 10 länger dauert, werden Unit-Tests und Integrations-Tests zu unterschiedlichen Zeitpunkten im Maven Build-Lifecycle ausgeführt. Die Ausführung der Unit-Tests erfolgt in der Maven Phase Test direkt nach der Kompilierung der Java-Klassen über das Surefire-Plugin. Integration-Tests können erst nach dem Zusammenbau der Anwendung (Maven Phase Package) in der Maven Phase Verify durch das Failsafe-Plugin ausgeführt werden, da vorher die nötige Voraussetzung – eine lauffähige Anwendung – noch nicht existiert.

Damit die beiden Plugins die auszuführenden Tests erkennen können, wird mit unterschiedlichen Suffixen bei der Benennung der Testklassen gearbeitet. Das Surefire-Plugin führt alle Tests mit dem für Unit-Tests übliche Suffix „Test“ aus. Das Failsafe-Plugin hingegen führt standardmäßig Testklassen mit dem Suffix „IT“ aus.

Da die Testklassen mit dem Testframework JUnit ausgeführt werden, wird in Zeile 1 der Abbildung 29 die Runner Klasse „Cucumber“ aus dem Cucumber-JUnit Modul registriert.

In den Zeilen 2 – 4 erfolgt die Konfiguration des Tests mit Hilfe der Parameter plugin, feature und glue.

* **plugin**: Für eine umfangreichere und gut lesbare Ausgabe der Testergebnisse auf der Konsole und in der Logdatei wird das Plugin „pretty“ definiert.
* **feature:** Über den Parameter wird angegeben, welche Feature-Datei von der Testklasse überprüft werden soll.
* **glue:** Der Parameter gibt an, in welchen Packages (inkl. Sub-Packages) sich die für die Testdurchführung benötigten glue-Klassen befinden. Dies umfassen die Step-Definitionen und zusätzliche Hilfs-Klassen.
  + 1. Testvorbereitung

Um Integration-Tests unabhängig voneinander ausführen zu können, ist für bestimmte Funktionen die Bereitstellung eines bestimmten Zustands in der Datenbank erforderlich. Da die Ausführung aller Anfragen bis zum Erreichen des gewünschten Zustands in der Regel eher unpraktisch ist, wird die Datenbank mit Hilfe von DbUnit für den Test vorbereitet. Hierfür kommen Data-Sets zum Einsatz, i.d.R. eine CSV-, XLS- oder XML-Dateien, in denen für eine oder mehrere Datenbank-Tabellen, die zu importierenden Daten angegeben werden.

**Before-Hook**

Damit sichergestellt ist, dass nur der erwartete Anfangszustand und keine Daten aus anderen Testfällen in der Datenbank enthalten sind, werden alle Datenbanktabellen – mit Ausnahme der schreibgeschützten Tabelle TAB\_ACCOUNT – vor jedem Szenario geleert. Hierbei handelt es sich um eine technische Voraussetzung zur Realisierung regressionsfähiger Tests, die sich in den Szenarien nicht wiederspielt und daher über einen Before-Hook (vgl. Abbildung 25) realisiert wird.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 32: Before-Hook zum leeren der Datenbank

Der Zugriff auf die Datenbank erfolgt über die Hilfsklasse DBConnection. Neben der Verbindung zur Datenbank stellt diese Methoden zum Ändern und Prüfen der gespeicherten Daten bereit. Z.B. führt die Methode clearTables auf den Tabellen TAB\_TWEET, TAB\_LIKER und TAB\_FOLLOWER ein DELETE\_ALL aus, sodass sämtliche vorhandenen Daten gelöscht werden.

**Testdatenbank initialisieren**

Im Rahmen eines Cucumber-Tests wird ein Data-Set vor bzw. zu Beginn der Ausführung eines Szenarios in die Datenbank eingespielt. Ein Problem ergibt sich jedoch aus der Verwendung eines Data-Sets in Form einer externen Datei. Die Feature-Datei verliert ihren Status als „Source of truth“, da sie von weiteren Dateien abhängig sind. Damit leidet auch das Verständnis, da nicht mehr sämtliche Daten auf einen Blick zu erfassen sind. Eine mögliche Lösung für diese Problem ist in der nachfolgenden Abbildung zu sehen.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 17: Background "getTweetsFromUser.feature"

In der Abbildung 17 wird ein Background-Block verwendet, um Tweets zu beschreiben, die in den Szenarien Teil des Ausgangszustandes sind.

Wird die in dem Background-Block enthaltene Beschreibung des Ausgangszustandes zur Initialisierung der Datenbank genutzt, wird sichergestellt, dass die Feature-Dateien als Dokumentation ausreichen sowie stets aktuell und korrekt sind.

In der zugehörigen Step-Definition wird anhand der definierten Tweets (Zeile 5-16) ein Data-Set erzeugt und in die Datenbank eingespielt. Dabei werden nur Pflichtfelder (Tweet-Id, Author-Id , Content, State, Publishing Date) sowie zusätzliche für den Test notwendige Daten berücksichtigt.

In manchen Steps, wie auch in Abbildung 17 zu erkennen, sind jedoch nicht alle Informationen zu einer Tweet Entität beschrieben. Eine automatische Transformierung durch die Type Registry in TweetEntity- Instanzen kann somit nicht erfolgen.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 18: Registrierung des Typs "TweetEntity.Builder" als DataTable Typen

Abbildung 18 zeigt ein Beispiel für die Registrierung eines eigenen Datentyps (TweetEntity.Builder) in der Type Registry.

„Builder“ ist eine innere Klasse von „TweetEntity“, die das Builder-Pattern des Domain-Driven-Design realisiert. Da die Klasse TweetEntity einen private Konstruktor besitzt können von ihr ausschließlich über die Builder Klasse Instanzen erzeugt werden. Vor der Erzeugung wird sichergestellt, dass alle nötigen Informationen einer Tweet Entität vorhanden sind. Die für die Transformation registrierte Methode liest somit zunächst alle Informationen einer Tweet Entität aus – auch wenn sie nicht vorhanden sind – und fügt sie anschließend dem Builder hinzu.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 19: Persistierung von Tweets

Die Step-Definition aus Abbildung 19 erhält die in der Data Table beschriebenen Informationen in Form von TweetEntity.Builder Instanzen. Die fehlenden Informationen über die Veröffentlichungsdaten und die Autoren werden den Builder-Instanzen hinzugefügt. Anschließend wird für jeden Builder die build()- Methode aufgerufen und die zurückgegebene TweetEntity Instanz einer Liste hinzugefügt.

Die fehlenden Informationen können nicht manuell innerhalb der transformierenden Methode der Type Registry hinzugefügt werden, da zu jedem Typen nur eine Methode registriert werden kann. In unterschiedlichen Steps können jedoch unterschiedliche Informationen zu einer Tweet Entität beschrieben werden. Die Data Table aus der Feature-Datei „getTweets.feature“ beschreibt z.B. zusätzlich den Autor der Tweets. Würde der Autor in der Methode der Type Registry fix als User „John“ gesetzt werden würden trotz der Beschreibung der Autoren jeder Tweet mit User „John“ als Autor abgespeichert werden.

Nachdem die Tweet Entitäten erstellt wurden werden diese an die Methode insertTweets der Klasse DBConnection übergeben. Diese erzeugt über den DBUnit DataSetBuilder ein entsprechendes Data-Set erstellt und fügt die Daten der Tabelle „TAB\_TWEET“ hinzu.

* + 1. Testausführung

Beim twttr-service handelt es sich wie beschrieben um eine Webanwendung mit einer datenzentrierten Schnittstelle. Mit Hilfe der Cucumber-Test sollen das Verhalten der bereitgestellten API Operationen geprüft werden.

Grundsätzlich sind alle Szenarien nach demselben Muster aufgebaut: Nach einer Authentifizierung gegen den Single-Sign-On Server (Keycloak Server) wird mit den Berechtigungen des authentifizierten Users ein HTTP-Request an die Schnittstelle geschickt und das erwartete Verhalten geprüft. Dies lässt sich am nachfolgenden Beispiel für das Szenario „*Create a new tweet*“ (vgl. Abbildung 20 und folgende) nachvollziehen.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 20: *createTweet.feature*

Das Szenario beginn mit dem Step „Given the user *max* is authenticated“ (Zeile 6). Die zugehörige Step-Definition *Authorize Account* wird in Abbildung 21 dargestellt.

Mit Hilfe des Keycloak AuthzClient wird eine Authentifizierungs-Anfrage mit dem Usernamen und dem zugehörigen Passwort an den KeyCloak Server gesendet. Während der Username als Output Parameter der Step-Definition übergeben wird, muss das Password aus der Instanz der Klasse SharedDomain – diese hält alle Usernamen und zugehörigen Passwörter in einer HashMap vor - geladen werden. Als Antwort auf die erfolgreiche Authentifizierung wird ein JSON Web Token zurückgeliefert, der für den folgenden Aufruf gegen den twttr-service benötigt wird. Damit die nächste Step-Definition auf den Token zugreifen kann, wird dieser in gemeinsamen State abgelegt.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 21: Step-Definition Authorize Account (GeneralSteps.java)

Die gezeigte Step-Definition enthält eine Captured Group „(user|moderator)“ die die beiden verwendeten Benutzerrollen abbildet. Damit kann der Step nicht nur für Szenarien der Rolle User sondern auch für die Rolle Moderator genutzt werden.

Anschließend wird der Step „a client sends a POST … (vgl. Abbildung 20, Zeile 7) ausgeführt, die einen HTTP-POST Request für den User max ausführt um einen neuen Tweet zu erstellen.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 22: Versenden eines Requests

Abbildung 22 zeigt die zuordenbare Step-Definition. Diese enthält als Parameter die anzusprechende Ressource, den Usernamen sowie den zu erstellenden Tweet. Letzter wird als eigener Datentyp (Klasse NewTweet) übergeben und vorab in der Type Registry der Typ „NewTweet“ als Output Parameter registriert.

Der HTTP-POST-Request mit JSON als Austauschformat sowie die Validierung der erwarteten Antwort erfolgen mit Hilfe des Testframework REST Assured. Damit der Request mit den Berechtigungen des Users Max ausgeführt wird, wird der zuvor abgerufene JSON Web Token im Authorization-Header mitgesendet (vgl. Zeile 7).

Die Steps 3 und 4 (vgl. Abbildung 20, Zeile 10 & 11) des Szenarios beschreiben wie die Schnittstelle auf den POST-Request (Step 2) reagieren soll. Die zugehörigen Step-Definitionen sind in den Abbildungen 23 und 24 dargestellt.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 23: Überprüfung des HTTP Status-Codes

Die Step-Definition zum Step „*Then the http response status-code will be 201*“ (vgl. Abbildung 23) prüft ob die Schnittstelle mit dem erwarteten Statuscode auf den Request reagiert hat. Eine zusätzliche Prüfung ob die Daten tatsächlich in der Datenbank gespeichert wurden erfolgt an dieser Stelle nicht.

Über die Step-Definition zum Step „*And the http response body contains …“* wird die erwartete Payload mittels REST Assured geprüft.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 24: Überprüfen der Response

Die durch RestAssured verwendete Struktur zum Überprüfen der Response bietet einen sehr übersichtlichen und gut verständlichen Aufbau. Dabei fällt auf, dass die Attribute „pubDate“ und „tweetId“ lediglich auf NotNull geprüft werden.

Dies liegt daran, dass es sich um dynamische, vom System vergebene Daten handelt. Das pubDate wird abhängig von der Zeitquelle des Host-Systems vergeben und lässt sich somit nicht ohne weiteres vorhersehen. Hingegen kann die tweetId aufgrund der bekannten Anzahl der persistierten Tweets und dem Intervall, in dem sich die IDs erhöhen leicht vorhergesehen werden. Es ist jedoch irrelevant, welche ID vergeben wurde und kann somit bei der Überprüfung vernachlässig werden.

* + 1. Zustände teilen

Wie in Kapitel 3.4.5 erwähnt müssen die Step-Definitionen eines Szenarios die Möglichkeit besitzen Zustände untereinander zu teilen. Zu diesem Zweck empfiehlt sich ein Objekt zum Halten der Zustände, auf das die glue-Klassen Zugriff haben.

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 31: Instanziierung der SharedDomain

Die Klasse „SharedDomain“ wird als Zustandshaltende Klasse verwendet. Sie enthält Attribute für alle Zustände, die über verschiedenen Step-Definitionen geteilt werden müssen. In der Beispielanwendung müssen z.B. Responses auf Requests geteilt werden sowie valide Token der User. Damit die glue-Klassen die gleiche Referenz auf das Zustandshaltende Objekt bekommen wird das DI-Framework „PicoContainer“ verwendet. Abbildung 31 zeigt wie die Klasse GeneralSteps per Konstruktor-Injection die Referenz auf das Singleton Objekt erhält. Die Konstruktoren können dabei beliebig viele Parameter enthalten, um vom PicoContainer Singleton Objekte Instanziieren zu lassen. Dies ermöglicht eine Aufteilung der Zustände auf verschiedene Klassen. Da die zu speichernden Zustände in der vorliegenden Beispielanwendung jedoch sehr überschaubar sind, wurde hier keine weitere Aufteilung vorgenommen.

* + 1. Authentication Token

Die Authentifizierung erfolgt durch das validieren von JWTs. Das Erzeugen der Token ist dabei jedoch nicht Part der RESTful API und ist somit auch nicht das überprüfen, ob korrekte Token generiert wurden. Da diese jedoch notwendig sind um die Schnittstelle verwenden zu können erläutert die Feature-Datei „authentication.feature“ vollständigkeitshalber, wie ein valider JWT erzeugt werden kann.

* + 1. Arten der JSON Beschreibung

Das Szenario „Create a new Tweet“ (vgl. Abbildung 20) demonstriert zwei Varianten zum Umgang mit Daten. Der Step „Given the user …“ (Zeile 8) verwendet eine Data Table, um Attribut-Namen, Datentyp und Wert zu definiere und diese mit Hilfe der Type Registry als eigenen Datentyp an die Step-Definition zu übergeben. Der Step „And the http response …“ (Zeile 12) nutzt die Doc String Variante um die erwartete Antwort zu definieren.

In der Beispielanwendung wird in den meisten Fällen ein Doc String für die Beschreibung von Antworten verwendet, da insbesondere bei verschachtelten Datenstrukturen eine bessere Übersicht gegeben ist.

Sollen unterschiedliche Datensätze aufeinander referenzieren, ist eine Data Table nur bedingt geeignet. Werden jedoch Key-Value Paare oder andere Strukturen über eine Data-Table definiert bietet diese eine deutliche höhere Lesbarkeit und damit ein leichteteres Verständnis für alle Beteiligten.

* + 1. Wiederholende Szenarien

Bitte eine Schnittstelle mehrere Funktionen an, so ist es wahrscheinlich das sich diese in gewisser Art und Weise ähneln. Z.B. erwartet ein Client immer dasselbe Verhalten, wenn eine Anfrage ohne gültigen Token gestellt wird, ein Validierungsfehler auftritt oder ein angefragter Datensatz nicht existiert. Nichtsdestotrotz muss das erwartete Verhalten für jede Funktion beschrieben werden, wodurch eine ganze Reihe von Szenarien mit großer Ähnlichkeit entstehen. Dies hat zur Folge, dass einige Szenarien mit nur geringen Unterschieden in mehreren Feature-Dateien existieren.

Jedoch sollte man nicht versuchen dies zu vermeiden, da das Ziel eine vollständige Dokumentation des erwarteten Verhaltens ist und jedes Feature mit seiner gesamten Funktionsweise abgebildet werden soll.

* + 1. Verwendung des Scenario Outline

Eine Besonderheit bieten die Features „Get user …“ (getUsers.feature), xxx (getTweets.feature) und yyy (getTweetsFromUser.feature). Alle drei Features beschreiben das erwartete Verhalten beim Abfragen einer Liste von Tweets bzw. Usern. Da mehrere Szenarien mit denselben Steps arbeiten, kommt die Szenario Outline zum Einsatz, wie nachfolgend am Beispiel des Features „Get …“ beschrieben wird.

Verwendung des Background-Blocks

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 25: Background-Block in "getUsers.feature"

Das Kernelement ..

Der Background-Block initialisiert die Datenbank, indem alle existierenden User und Moderatoren beschrieben werden. Da in der Beispielanwendung keine Funktion zum Hinzufügen, Verändern oder Entfernen von Accounts existiert ist dies grundsätzlich ein überflüssiger Step. In den übrigen Feature-Dateien wird daher auch auf die Initialisierung der Accounts verzichtet. Jedoch ist es zum Verstehen unter welchen Bedingungen wessen User-Informationen zurückgesendet werden notwendig sie hier aufzuführen.

Da die Initialisierung der Datenbank durch einen Background-Block erfolgt wird diese auch in jedem Szenario entsprechend durchgeführt (vgl. Kapitel xxx). Das Szenario „Unauthorised request to get a list of users“ würde dies nicht benötigen, da in diesem Fall keine User-Informationen zurückgeliefert werden – die Initialisierung ist wiederum für die Ausführung des Szenarios auch nicht störend. Das Verlegen des Steps in den Background-Block erhöht dafür deutlich die Lesbarkeit der übrigen, ohnehin komplexen, Szenarien.

Initialisierung der Datenbank

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 26: Persist Accounts

Die Accounts werden zugunsten des Leser-Verständnisses aufgeführt. Da die Beispielanwendung jedoch keine Funktion zum Verändern der Account-Daten bietet und diese sich somit nicht verändern würde eine leere Step-Defintion zu dem gleichen Ergebnis führen. Damit die Dokumentation konsistent bleibt sollte die beschriebene Logik trotzdem ausgeführt werden. Im Fall einer später hinzugefügten Funktion zum Verändern der Account-Daten könnte sonst schnell zu Verwirrungen führen.

Inhaltlich unterschiedliche Szenarien

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 27: Unterschiedliche Rollen fragen user Informationen ab

Abbildung 19 zeigt zwei sich ähnelnde Szenarien der Feature-Datei. In beiden wird ein valider Request, ohne Setzen von Query Parametern, durchgeführt. Der einzige Unterschied liegt darin, dass im ersten Szenario der Request für einen User und im zweiter für einen Moderator versendet wird. Aufgrund der inhaltlich sehr ähnlichen Vorgehensweisen hätten ggf. die beiden Szenarien in einem Szenario Outline zusammengefasst werden können. Da jedoch eine inhaltliche Unterscheidung vorliegt, wurden an dieser Stelle zwei explizit unterschiedliche Szenarien verwendet.

Verschiedene Example-Tabellen

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 28: Mehrere Example Tabellen in "getUsers.feature"

Im Gegensatz zu der Erklärung, wie sich die Ressource auf Requests verschiedener Rollen verhält, soll an dieser Stelle erklärt werden, wessen User-Informationen unter Verwendung verschiedener Werte-Kombinationen der Query Parameter zurückgesendet werden. Aufgrund des inhaltlich gleichen Szenarios mit unterschiedlichen Ausprägungen der Query Parameter wurde an dierser Stelle ein Szenario Outline verwendet. Da die Kombination aller Query Parameter zusammen einen Einfluss darauf haben, wessen User Informationen zurückgesendet werden existieren recht viele Grenzfälle, die abgebildet werden müssen (vgl. Kapitel 3.2.3).

Damit die vielen verschiedenen Fälle übersichtlich bleiben und schnell ersichtlich wird, was mit den Beispielen erläutert und getestet werden soll sind die verschiedenen Kombinationen in unterschiedlichen betitelten Example-Tabellen gruppiert. Zu jedem Grenzfall existiert ein Szenario, welches beschreibt, wessen User-Informationen zurückgesendet werden.

1. Fazit

Cucumber verknüpft fachliche Akzeptanzkriterien mit Softwaretests. Daraus ergeben sich einige maßgebliche Vorteile:

1. Domänen-Experten und andere Projektbeteiligte ohne technischen Hintergrund können anhand der in natürlicher Sprache formulierten Szenarien nachvollziehen, welche Anforderungen durch die Softwaretests überprüft werden und ob die Software die Akzeptanzkriterien auch erfüllt.
2. Software-Entwickler erhalten durch die Szenarien bereits mit der Anforderung eine genaue Vorgabe wie sie ihre Implementierung auf fachliche Korrektheit hin überprüfen können. Durch die enge Zusammenarbeit und den intensiven Austausch zwischen Entwicklern und Domänen-Experten können sich beide Gruppen bereits vor dem Beginn der Entwicklung Feedback geben.

Maßgeblich zum Erfolg von Cucumber trägt die Verknüpfung mit Test-Frameworks wie JUnit und TestNG bei. Bei der Testentwicklung können die Entwickler die Vorteile und Stärken beider Seiten zusammenführen und Cucumber ohne jede Schwierigkeit mit Testframeworks zum Testen von Schnittstellen, Oberflächen und Datenbanken kombinieren. Darüber hinaus bekommt man die vollständige Automatisierung von Akzeptanztest sowie die Integration in Continuous Integration Pipelines quasi geschenkt.

Seine volle Stärke spielt Cucumber in Verbindung mit grafischen Benutzerschnittstellen aus, deren erwartetes Verhalten in einfacher natürlicher Sprache definiert werden kann. Etwas anders sieht es bei datenzentrierten und damit technischen Schnittstellen aus. Auch wenn eine Schnittstelle eine fachliche Anforderung abbildet, ist es in der Regel nicht möglich technische Aspekte wie technische Fachbegriffe, Statuscodes oder verschachtelte Datenstrukturen in der Formulierung der Akzeptanztests vollkommen außen vor zu lassen. Dies erschwert jedoch Lesbarkeit wie Verständnis und ist letztendlich für die Zusammenarbeit mit Domänen-Experten nicht so nachhaltig.

Unter dem Strich bietet Cucumber in jedem Fall einen signifikanten Mehrwert bei der Qualitätssicherung von Software-Projekten und hilft Domänen-Experten, Entwicklern und Testern dabei bessere, verständlichere und weniger fehlerbelastete Anwendungen zu entwickeln, die vor allem die Anforderungen erfüllen die an sie gestellt werden.

Positiv:

* Fachlich sehr genau geklärt, was von Software erwartet wird
* Mehr Mut zu Code Änderungen 🡪 Fehler fallen schnell auf
* Für jeder Mann verständlich
* Lebende Dokumentation
* Single Source of truth
* Nur programmieren, was auch wirklich gewünscht ist

Negativ:

* Unit-Tests 🡪 Keine Referenzierungen von DataTabel in DataTable möglich (schwierig Datenzustände zu simulieren)
* Dynamische Daten nur schwer möglich
* Nur Black-Box 🡪 heißt nicht, dass es effizient ist
* Kommunikations-Gap zwischen Domänen-Experten und Software-Entwickler beim Erstellen einer RESTful API nicht mehr so groß
* Szenarien aufgrund der Datenbeschreibung nicht mehr wie Geschichten lesbar/ formulierbar

Literaturverzeichnis

Rose S., Wynne M., Hellesoy A. (2015): The Cucumber for Java Book, Dallas.

<https://cucumber.io/blog/2017/05/15/intro-to-bdd-and-tdd>, abgerufen am 06.12.2019

<https://docs.cucumber.io/bdd/overview/>, abgerufen am 01.04.2019

<https://cucumber.io/blog/example-mapping-introduction/>, abgerufen am 06.02.2019

[https://docs.cucumber.io/guides/bdd-tutorial/#discovery-workshops](https://docs.cucumber.io/guides/bdd-tutorial/" \l "discovery-workshops), abgerufen am 06.02.2019

<https://cucumber.io/>, abgerufen am 21.02.2019

<https://dzone.com/articles/guide-to-behavior-driven-development-in-java?edition=423370&utm_source=Zone%20Newsletter&utm_medium=email&utm_campaign=java%202019-01-29>, abgerufen am 20.03.2019

<https://docs.cucumber.io/cucumber/step-definitions/>, abgerufen am 20.03.2019

<https://docs.cucumber.io/gherkin/reference/>, abgerufen am 10.03.2019

<https://jkotests.wordpress.com/2013/08/22/specify-execution-order-of-cucumber-features/>, abgerufen am 10.03.2019

[https://docs.cucumber.io/cucumber/state/#dependency-injection](https://docs.cucumber.io/cucumber/state/" \l "dependency-injection), abgerufen am 11.03.2019

<https://cucumber.io/blog/2017/07/26/announcing-cucumber-expressions>, abgerufen am 11.03.2019

<https://docs.cucumber.io/cucumber/cucumber-expressions/>, abgerufen am 11. 03.2019

https://mincong-h.github.io/2018/10/04/maven-failsafe-plugin-understanding/, abgerufen am 02.04.2019

[https://docs.cucumber.io/bdd/who-does-what/#writing-features](https://docs.cucumber.io/bdd/who-does-what/" \l "writing-features), abgerufen am 03.04.2019

Wirdemann R., Mainusch J. (2017): Scrum mit User Stories, 3. Auflage, München

EIDESSTATTLICHE ERKLÄRUNG

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt und mich anderer als der in den beigefügten Verzeichnissen angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Der Durchführung einer elektronischen Plagiatsprüfung stimme ich hiermit zu. Die eingereichte elektronische Fassung der Arbeit entspricht der eingereichten schriftlichen Fassung exakt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen hat.

Ort, den Vorname, Name

1. Vgl. O.V. (o.J.) https://cucumber.io/blog/intro-to-bdd-and-tdd/ [↑](#footnote-ref-1)
2. Vgl. Rose, Wynne, Hellesoy (2015), S. 4 [↑](#footnote-ref-2)
3. Vgl. Rose, Wynne, Hellesoy (2015), S. 4f [↑](#footnote-ref-3)
4. Vgl. https://cucumber.io/blog/example-mapping-introduction/ [↑](#footnote-ref-4)
5. Vgl. https://cucumber.io/blog/intro-to-bdd-and-tdd/ [↑](#footnote-ref-5)
6. Vgl. https://docs.cucumber.io/guides/bdd-tutorial/#discovery-workshops [↑](#footnote-ref-6)
7. Vgl. https://cucumber.io/blog/example-mapping-introduction/ [↑](#footnote-ref-7)
8. Vgl. https://docs.cucumber.io/bdd/overview/ [↑](#footnote-ref-8)
9. Vgl. Rose, Wynne, Hellesoy (2015), S. 4 [↑](#footnote-ref-10)
10. Vgl. Wirdemann, Mainusch (2017), S. 195 [↑](#footnote-ref-12)
11. Vgl. O.V. (o.J.) https://cucumber.io/blog/intro-to-bdd-and-tdd/ [↑](#footnote-ref-13)
12. Vgl. O.V. (o.J.) https://cucumber.io/blog/intro-to-bdd-and-tdd/ [↑](#footnote-ref-14)
13. https://cucumber.io/ [↑](#footnote-ref-15)
14. Vgl. https://docs.cucumber.io/gherkin/reference/ [↑](#footnote-ref-16)
15. Vgl. Rose, Wynne, Hellesoy (2015), S. 36 [↑](#footnote-ref-17)
16. Buch S. 86 [↑](#footnote-ref-18)
17. Buch S. 73 [↑](#footnote-ref-19)
18. Buch S. 75 [↑](#footnote-ref-20)
19. [↑](#footnote-ref-21)
20. [↑](#footnote-ref-22)
21. https://docs.cucumber.io/gherkin/reference/#rule [↑](#footnote-ref-23)
22. In dieser Arbeit wird v3.0.2 genutzt [↑](#footnote-ref-24)
23. Vgl. O.V. (o.J.) https://docs.cucumber.io/cucumber/api/#hooks [↑](#footnote-ref-25)
24. https://cucumber.io/blog/announcing-cucumber-jvm-3-0-0/ [↑](#footnote-ref-26)
25. https://docs.cucumber.io/cucumber/api/#hooks [↑](#footnote-ref-27)
26. Vgl. Rose, Wynne, Hellesoy (2015), S. 37 [↑](#footnote-ref-28)
27. Vgl. http://www.thinkcode.se/blog/2017/08/16/sharing-state-between-steps-in-cucumberjvm-using-guice [↑](#footnote-ref-29)
28. Vgl. Rose, Wynne, Hellesoy (2015), S. 47 [↑](#footnote-ref-30)
29. https://docs.cucumber.io/gherkin/reference/#example [↑](#footnote-ref-31)
30. https://docs.cucumber.io/cucumber/api/#hooks [↑](#footnote-ref-32)
31. Vgl. https://dzone.com/articles/guide-to-behavior-driven-development-in-java?edition=423370&utm\_source=Zone%20Newsletter&utm\_medium=email&utm\_campaign=java%202019-01-29 [↑](#footnote-ref-33)
32. https://github.com/cucumber/cucumber-jvm/tree/master/picocontainer [↑](#footnote-ref-34)