Fundamentals of BDD with Cucumber (BDD-Cuke)
AU - Grundlagen von BDD mit Cucumber
Lehrplan

Herausgegeben auf Basis der englischen Version 1.0

Urheberrechtshinweis

Vielen Dank an die Autoren

- Pascal Moll

Vielen Dank an die Berater

- Boris Wrubel und Rahul Verma

Vielen Dank an das Reviewkomitee

- Kyle Alexander Siemens, Emilie Potin-Suau

Aus Gründen der Lesbarkeit wurde im vorliegenden Text die männliche Form gewählt, nichtsdestoweniger beziehen sich die Angaben auf alle Geschlechter.

Revisionshistorie

Version	Datum	Bemerkungen
0.01	November 2024	Erstversion
0.1	Juli 2025	Version für Alpha-Review
0.2	November 2025	Version für Beta-Review
0.3	Januar 2025	Anpassungen basierend auf Reviews
1.0	Februar 2026	Erste Veröffentlichung auf Deutsch
1.0	März 2026	Offizielle öffentliche Veröffentlichung

Inhaltsverzeichnis

Geschäftsergebnisse (Business Outcomes)

Business Objects (BOs) sind eine kurze Zusammenfassung dessen, was Sie nach der Schulung gelernt haben sollten.

BO-1	Lernen Sie allgemeine Informationen über Behaviour Driven Development und dessen Unterschiede zu anderen Methoden kennen.
BO-2	Verstehen Sie die Prinzipien von Agile und Scrum in Bezug auf Tests und BDD.
BO-3	Verstehen Sie, was die Ziele von BDD sind und wie Sie diese erreichen können.
BO-4	Lernen Sie etwas über Abstraktion und wie Sie BDD-Szenarien zu Ihrem Vorteil nutzen können.
BO-5	Lernen und verstehen Sie die verschiedenen Testtechniken in BDD und deren Zyklus.
BO-6	Verstehen Sie, wie Sie Ihre Testumgebung einrichten.
BO-7	Lernen Sie, wie Sie TestNG verwenden.
BO-8	Lernen Sie die verschiedenen Funktionen von Cucumber und Gherkin kennen.
BO-9	Lernen Sie, wie Sie BDD in Ihren Softwareentwicklungslebenszyklus integrieren können.
BO-10	Verstehen Sie, wie API und Mocks funktionieren
BO-11	Erkennen Sie die Unterschiede zwischen verschiedenen Übertragungstechnologien
BO-12	Lernen Sie Karate kennen und erfahren Sie, wie Sie es mit der Gherkin-Syntax verwenden
BO-13	Verwenden und verstehen Sie datengesteuertes Testen
BO-14	Lernen Sie die verschiedenen Mocking-Funktionen kennen
BO-15	Lernen Sie die Grundlagen von Continuous Integration und Continuous Development (CI/CD) kennen
BO-16	Lernen Sie den Buildserver Jenkins kennen
BO-17	Erstellen Sie Testberichte und bauen Sie Ihre Tests automatisch auf
BO-18	Best Practices in allen Bereichen

Lernziele

Lernziele (LOs) sind kurze Aussagen, die beschreiben, was Sie nach dem Studium jedes Kapitels wissen sollten. Die LOs werden auf der Grundlage der modifizierten Taxonomie von Bloom wie folgt definiert:

Definitionen	**K1 Erinnern**	**K2 Verstehen**	**K3 Anwenden**
Blooms Definition	Erinerung an zuvor gelerntes Material, indem Sie sich an Fakten, Begriffe, Grundkonzepte und Antworten erinnern.	Demonstrieren Sie Ihr Verständnis von Fakten und Ideen, indem Sie diese organisieren, vergleichen, übersetzen, interpretieren, beschreiben und die Hauptgedanken darlegen.	Lösen Sie Probleme in neuen Situationen, indem Sie erworbenes Wissen, Fakten, Techniken und Regeln auf andere Weise anwenden.
Verben (Beispiele)	Erinnern Wiedergeben Auswählen Definieren Finden Zuordnen In Beziehung setzen Auswählen	Zusammenfassen Verallgemeinern Klassifizieren Vergleichen Gegenüberstellen Demonstrieren Interpretieren Umformulieren	Umsetzen Ausführen Verwenden Anwenden Planen Auswählen

Weitere Einzelheiten zur Taxonomie von Bloom finden Sie unter **[BT1]** und **[BT2]** in den Referenzen.

Praktische Ziele (Hands-on Objectives)

Praktische Ziele (HOs) sind kurze Aussagen, die beschreiben, was Sie tun oder ausführen müssen, um den praktischen Aspekt des Lernens zu verstehen. Die HOs sind wie folgt definiert:

- HO-0: Live-Ansicht einer Übung oder eines aufgezeichneten Videos.
- HO-1: Geführte Übung. Die Auszubildenden folgen der Abfolge der vom Ausbilder durchgeführten Schritte.
- HO-2: Übung mit Hinweisen. Übung, die vom Teilnehmer unter Verwendung der vom Trainer gegebenen Hinweise gelöst werden muss.
- HO-3: Ungeleitete Übungen ohne Hinweise.

Voraussetzungen

Obligatorisch

- Keine

Empfohlen

- Ein Agile- oder Scrum-Zertifikat wie PSM, CSM oder ASF oder zumindest die Lektüre des Scrum-Leitfadens.
- Grundkenntnisse (ISTQB-CTFL oder ISTQB-CFTL-Agile Tester) im Bereich Testen im Allgemeinen.
- Mindestens 1 Jahr Berufserfahrung im Bereich Agile und Testen.
- Lesen Sie ein Buch über Agile Testing wie „Agile Testing“ **[JL1]** und „More Agile Testing” **[JL1].**

Kapitel 1 - Grundlagen von BDD

Schlüsselbegriffe

BDD, Behaviour Driven Development, Data Driven Testing, Keyword Driven Testing

LO-1.1	K1	Verstehen, was Behavior Driven Development ist und wie es sich von anderen Testmethoden unterscheidet
LO-1.2	K2	Wichtige Meilensteine in der Entwicklung von BDD benennen können
LO-1.3	K3	Die Auswirkungen von BDD auf die Qualität der Softwareentwicklung beschreiben
LO-1.4	K2	Grundlegende agile Prinzipien verstehen und deren Zusammenhang mit BDD erkennen. Die Rolle von Feedback-Schleifen in agilen Methoden im Kontext von BDD erläutern
LO-1.5	K2	Erklären, wie BDD in Scrum implementiert wird
LO-1.6	K2	Die spezifischen Verantwortlichkeiten jedes Stakeholders (Product Owner, Entwickler, Tester) im BDD-Prozess beschreiben.
LO-1.7	K2	Techniken zur Abstraktion von Anforderungen beschreiben.
LO-1.7.1	K1	Den Prozess der Anforderungsabstraktion verstehen
LO-1.7.2	K2	Beispiele für die Anwendung der Gherkin-Syntax in Testszenarien erstellen
LO-1.7.3	K1	Den Begriff der Trennung von Geschäftslogik und Implementierung definieren
LO-1.7.4	K2	Beispiele für wiederverwendbare Schritte in Testszenarien identifizieren
LO-1.7.5	K1	Die Rolle der Kommunikation zwischen Stakeholdern im BDD-Prozess erkennen
LO-1-8	K3	Beispiele für Testszenarien erkenne und die richtige auswählen können, die auf den Prinzipien von BDD basieren.
LO-1.9	K2	Die Vorteile von DDT im Kontext von BDD erläutern, einschließlich der Effizienzsteigerung bei Tests
LO-1.10	K2	Die Unterschiede zwischen KDT und DDT erklären und deren jeweilige Anwendungsfälle diskutieren.
LO-1.11	K1	Eine Übersicht über andere gängige Tools und Frameworks für BDD bereitstellen (z.B. Cucumber, FitNesse).
LO-1.12	K2	Die häufigsten Herausforderungen und Nachteile bei der Implementierung von BDD erläutern.
LO-1.13	K1	Die Hauptgründe für die Beliebtheit von BDD im Softwareentwicklungsbereich erkennen.

1.1 Was ist Behavior-Driven Development?

Behavior-Driven Development oder kurz BDD ist eine agile Softwareentwicklungsmethode, die sich auf die Zusammenarbeit zwischen der Entwicklungsabteilung, Qualitätssicherung und Fachpersonal fokussiert. Dabei wird das Verhalten eines Features in für jede Partei verständlichen Szenarien genauer beschrieben. Aus dieser Beschreibung lassen sich ebenso Tests ableiten. Beschreibung, Tests und weitere notwendige Informationen werden dabei an einer zentralen Stelle abgelegt und bilden so eine gemeinsame Quelle der Wahrheit.

Die grundlegenden Konzepte von BDD:

- **Verhaltensspezifikation**

BDD fördert die Spezifikation des Verhaltens einer Anwendung in einer für alle Beteiligten verständlichen Sprache. Dies geschieht oft in Form von "Given-When-Then"-Szenarien, die eine Vorbedingung (Given), eine Aktion (When) und das erwartete Ergebnis (Then) beschreiben.

- **Zusammenarbeit**

BDD fördert die Zusammenarbeit zwischen verschiedenen Stakeholdern, einschließlich Entwicklern, Testern und Fachexperten. Diese Zusammenarbeit hilft dabei, ein gemeinsames Verständnis der Anforderungen, dessen Umsetzung und zugehörigem Tests zu schaffen.

- **Testen als Teil des Entwicklungsprozesses**

In BDD wird das Testen nicht als separater Schritt betrachtet wie bei anderen Modellen, sondern als integraler Bestandteil des Entwicklungsprozesses. Tests werden so früh wie möglich erstellt und dienen auch als Dokumentation der Anforderungen.

- **Fokus auf Stakeholderbedürfnisse**

BDD legt großen Wert darauf, dass die entwickelten Funktionen den tatsächlichen Bedürfnissen der Anwender und Stakeholder entsprechen. Die Szenarien sollten auch aus der Perspektive des Endbenutzers beschrieben werden.

- **Iterative Entwicklung**

Wie bei anderen agilen Methoden wird auch bei BDD in kurzen Iterationen gearbeitet, um schnell Feedback zu erhalten und Anpassungen frühzeitig vorzunehmen. Die iterative Arbeitsweise ist dabei ein wichtiger Bestandteil von BDD, da die Szenarien kontinuierlich verfeinert und erweitert werden, um eine bessere Verständlichkeit und Testabdeckung zu gewährleisten.

Durch diese Konzepte zielt BDD darauf ab, Missverständnisse zu reduzieren und sicherzustellen, dass die entwickelte Software den Erwartungen der Stakeholder entspricht. In der Praxis wird BDD häufig zusammen mit Scrum eingesetzt und stellt dabei kein eigenständiges Framework für Zusammenarbeit dar. Vielmehr ist BDD eine Methodik, die genutzt wird.

1.2 Geschichte

- **Der Ursprung in TDDD**

Test Driven Development, abgekürzt mit TDD, wurde in den späten 1990er Jahren beliebt. Besonders die Arbeit von Kent Beck und anderen Mitgliedern der Extreme Programming (XP)-Bewegung trugen hierzu bei. Der Fokus von TDD liegt auf dem Schreiben von Tests, bevor der eigentliche Code entwickelt wird. Dies fördert eine testgetriebene Denkweise und hilft dabei, Fehler frühzeitig zu erkennen.

- **2003**

Der Begriff "Behavior Driven Development" wurde von Dan North geprägt. Er stellte fest, dass vor der Implementierung zuerst die Verhaltensbeschreibung einer Software erfolgen sollte.

- **2008**

Die Veröffentlichung von "Cucumber", einem weiteren wichtigen BDD-Tool, das es ermöglicht, Tests in einer leicht verständlichen Sprache (Gherkin) zu schreiben. Cucumber förderte die Akzeptanz von BDD über die Ruby-Community hinaus.

- **2010er Jahre**

BDD gewann an Beliebtheit in zahlreichen Programmiersprachen und Frameworks. Viele Tools und Bibliotheken wurden entwickelt, um BDD-Praktiken zu unterstützen (z.B. SpecFlow für .NET).

- **2015 und darüber hinaus**

BDD wird zunehmend als Teil agiler Entwicklungsmethoden angesehen und findet Anwendung in DevOps-Umgebungen sowie bei der Entwicklung von Microservices.

1.3 Ziele von BDD

- **Verbesserte Kommunikation**

BDD fördert die Zusammenarbeit zwischen Entwicklern, Testern und nicht-technischen Stakeholdern (z. B. Produktmanagern oder Kunden). Durch die Verwendung einer gemeinsamen Sprache (oft in Form von Gherkin-Syntax) wird sichergestellt, dass alle Beteiligten ein gemeinsames Verständnis der Anforderungen haben.

- **Frühe Validierung von Anforderungen**

Durch das Schreiben von Tests vor der Implementierung (Test-First-Ansatz) können Missverständnisse über Anforderungen frühzeitig erkannt und behoben werden. Dies reduziert das Risiko von Fehlern und kostspieligen Änderungen in späteren Phasen der Entwicklung.

- **Lebende Dokumentation**

Die in BDD verwendeten Spezifikationen dienen auch als lebendige Dokumentation des Systems. Sie sind leicht verständlich und können sowohl von technischen als auch von nicht-technischen Teammitgliedern gelesen werden. Dabei sind sie immer aktuell und Änderungen stets eingepflegt, da sie auch als Testausführung dienen.

- **Verhalten abtesten**

BDD legt den Schwerpunkt auf das gewünschte Verhalten der Software aus Sicht seiner Nutzer. Dies hilft dabei, sicherzustellen, dass die entwickelten Funktionen tatsächlich den Bedürfnissen entsprechen. Eine intuitive Bedienung durch seine Anwender wird dadurch gefördert.

- **Ermöglichung schnelle Reaktion der Teams**

BDD unterstützt agile Entwicklungsmethoden. Teams wird ermöglicht, schnell auf Änderungen in den Anforderungen zu reagieren und iterative Verbesserungen vorzunehmen.

1.4 Agile Fundamentals - Einbindung in BDD

- **Kundenorientierung**

BDD fördert die enge Zusammenarbeit mit den Stakeholdern. Durch den Einsatz von BDD lassen sich deren Anforderungen und Erwartungen leichter verstehen. Dies steht im Einklang mit dem agilen Wert der Zusammenarbeit mit dem Kunden über Vertragsverhandlungen.

- **Iterative Entwicklung**

Iterative Entwicklungszyklen werden durch BDD gefördert. Features lassen sich in kleinen Inkrementen definieren und implementieren. Die Testszenarien erfolgen in Form von formulierten "Given-When-Then"-Szenarien, was eine klare Struktur für die iterative Entwicklung ermöglicht.

- **Transparenz**

Durch die Verwendung einer klaren und verständlichen Sprache in BDD-Szenarien wird Transparenz für alle Beteiligten geschaffen. Stakeholder und Teammitglieder können Anforderungen und Tests verstehen, was die Kommunikation auch untereinander verbessert und Missverständnisse reduziert.

- **Selbstorganisierte Teams**

BDD ermutigt Teams dazu, gemeinsam an der Definition von Anforderungen und Tests zu arbeiten. Dies fördert ein Gefühl der Eigenverantwortung und Teamarbeit. Genau dieser Sachverhalt ist ein zentraler Aspekt agiler Methoden.

- **Kontinuierliche Verbesserung**

Die regelmäßige Überprüfung von BDD-Szenarien auch während Retrospektiven oder Sprint-Reviews ermöglicht es Teams, ihre Prozesse kontinuierlich zu verbessern. Dabei erfolgt auch eine Prüfung und Sicherstellung, dass sie den Bedürfnissen der Adressanten entsprechen.

- **Flexibilität und Anpassungsfähigkeit**

Da BDD eng mit den Anforderungen des Kunden verbunden ist, können Teams schnell auf Änderungen reagieren und neue Szenarien hinzufügen oder bestehende anpassen. Somit sind sich ändernde Situationen stets adressiert und berücksichtigt.

- **Technische Exzellenz**

BDD fördert gute Praktiken in der Softwareentwicklung durch das Schreiben von Tests von allen Teammitgliedern. Devs testen ihre Auslieferung vor der Übergabe und werden ermutigt, dies in einer allgemein verständlichen Sprache zu tun. Auch der Test Driven Development Ansatz wird ermutigt. Dies trägt zur technischen Exzellenz bei und stellt sicher, dass die entwickelte Software qualitativ hochwertig ist und hohe Codequalität eingehalten wird.

1.5 Anwendung von BDD in agilen Methoden

1.5.1 Scrum

1.5.1.1 Integration von BDD in Scrum

1. Rollen

- Product Owner:
  Definiert die Anforderungen in Form von BDD-Szenarien.
  Stellt sicher, dass die Szenarien klar und verständlich sind, um Missverständnisse zu vermeiden.
- Scrum Master:

Unterstützt das Team bei der Implementierung von BDD-Praktiken.
Fördert eine Kultur der Zusammenarbeit und des kontinuierlichen Lernens.

- Entwicklungsteam und QA:
  Arbeitet gemeinsam an der Umsetzung der BDD-Szenarien in funktionierenden Code.
  Kann eigene BDD Szenarien bei Bedarf ergänzen.
  Involviert Tester frühzeitig im Prozess, um sicherzustellen, dass alle Perspektiven berücksichtigt werden.

2. Sprint Planning

- Bereits erstellte BDD Szenarien einplanen:

Während des Sprint Planning werden neue Features oder User Stories in Form von BDD-Szenarien in den Sprint eingeplant. Die Szenarien können dabei als noch auszuformulieren als Aufgabe eingeplant werden aus als bereits fertig definiertes Szenario.

- Priorisierung:

Die Szenarien werden priorisiert und in das Sprint Backlog aufgenommen

3. Sprint Review

Präsentation der implementierten Features anhand der BDD-Szenarien.
Stakeholder geben Feedback zu den umgesetzten Funktionen und deren Übereinstimmung mit den Erwartungen.

1.5.1.2 Kontinuierliche Verbesserung

- Anpassung der Praktiken:
  Basierend auf den Erkenntnissen aus der Retrospektive passt das Team seine Vorgehensweise an, um die Effizienz und Effektivität bei der Anwendung von BDD zu steigern.
- Schulung und Weiterbildung:
  Gegebenenfalls werden Schulungen oder Workshops angeboten, um das Verständnis für BDD im Team zu vertiefen und Best Practices zu teilen.

4. Sprint Retrospective

Reflection on the sprint process:

- **Reflexion über den Sprint-Prozess**

Das Team diskutiert, wie gut die BDD-Praktiken während des Sprints umgesetzt wurden.
Identifikation von Stärken und Schwächen im Umgang mit BDD-Szenarien.

- **Verbesserungsmöglichkeiten werden diskutiert**

Diskussion darüber, welche Aspekte der BDD-Integration gut funktioniert haben und welche nicht.
Entwicklung von Aktionspunkten zur Verbesserung der Zusammenarbeit zwischen Entwicklern, Testern und dem Product Owner in zukünftigen Sprints.

5. Daily Stand-up
   - Teammitglieder berichten auch über den Fortschritt bei der Umsetzung der BDD-Szenarien.
   - Diskussion über Herausforderungen oder Hindernisse, die bei der Implementierung auftreten.

6. Entwicklungsmöglichkeiten

- **Test-First-Ansatz:**

Die Entwicklung beginnt mit dem Schreiben von Tests basierend auf den BDD-Szenarien. Diese Tests dienen als Spezifikation für die zu entwickelnde Funktionalität.

- **Kollaboration:**

Entwickler und Tester arbeiten eng zusammen, um sicherzustellen, dass die implementierten Funktionen den definierten Szenarien entsprechen.

1.5.1.3 Vorteile der Integration von BDD in Scrum

- **Verbessertes Verständnis**

Durch die Verwendung einer gemeinsamen Sprache wird das Verständnis zwischen technischen und nicht-technischen Stakeholdern verbessert.

- **Frühe Fehlererkennung**

Tests werden frühzeitig geschrieben, was dazu beiträgt, Fehler bereits in der Entwicklungsphase zu identifizieren.

- **Höhere Qualität**

Die enge Zusammenarbeit zwischen Entwicklern und Testern führt zu einer höheren Softwarequalität und besserem Feedback.

- **Kundenorientierung**

Die Fokussierung auf das Verhalten aus Sicht des Endbenutzers sorgt dafür, dass die entwickelten Funktionen den tatsächlichen Bedürfnissen entsprechen.

1.5.1.4 Nachteile und Herausforderungen von BDD in Scrum

1. Einarbeitungszeit period:

- **Einarbeitung**

Teams, die neu in BDD sind, benötigen Zeit, um sich mit den Konzepten und der Syntax (z. B. Gherkin) vertraut zu machen. Dies kann anfangs zu Verzögerungen führen.

- **Schulung**

Möglicherweise sind Schulungen erforderlich, um sicherzustellen, dass alle Teammitglieder die Prinzipien von BDD verstehen und anwenden können.

2. Zusätzlicher Aufwand:

- **Testentwicklung**

Das Schreiben von BDD-Szenarien erfordert zusätzlichen Aufwand im Vergleich zu traditionellen Testmethoden. Dies kann den Entwicklungsprozess gerade zu Beginn verlängern, insbesondere wenn das Team nicht erfahren ist.

- **Wartung der Szenarien**

Die Pflege und Aktualisierung der BDD-Szenarien kann zeitaufwendig sein, insbesondere wenn Anforderungen sich häufig ändern.

3. Missverständnisse bei der Implementierung:

- **Unklare Szenarien**

Wenn die BDD-Szenarien nicht klar formuliert sind oder Missverständnisse zwischen den Stakeholdern bestehen, kann dies zu falschen Implementierungen führen.

- **Übermäßige Komplexität**

Teams können zu viele Details in die Szenarien aufnehmen, was sie unübersichtlich und schwer verständlich macht.

4. Abhängigkeit von Zusammenarbeit:

- **Teamzusammensetzung**

Der Erfolg von BDD hängt stark von der Zusammenarbeit zwischen Entwicklern, Testern und Fachleuten ab. Wenn diese Zusammenarbeit nicht harmoniert, kann dies die Effektivität von BDD beeinträchtigen. Ein hohes Maß an Absprachen ist stets erforderlich. Geg. Müssen alte Gewohnheiten verändert und durch neue ersetzt werden. Hier wird auch oft von der Kultur der Veränderung gesprochen.

- **Kunden- & Stakeholder-Engagement**

Ein aktives Engagement der Kunden und Stakeholder ist entscheidend für den Erfolg von BDD. Wenn Stakeholder nicht regelmäßig involviert sind, können wichtige Anforderungen übersehen oder falsch umgesetzt werden.

1.5.1.5 BDD und Kanban

- Kanban Prinzipien
  - Visualisierung: Der Arbeitsfluss wird visualisiert, um Engpässe und Fortschritte sichtbar zu machen.
  - Limitierung der Work-in-Progress (WIP) Limits: Eine definierte Maximalgrenze von angefangenen Tasks hilft, Überlastung zu vermeiden und den Fokus auf die prevent overload and encourage task completion.

1.5.1.6 Integration von BDD into Kanban

- **User Stories als Ausgangspunkt**

User Stories können im Kanban-Board als Aufgaben dargestellt werden. Jede Story sollte klare Akzeptanzkriterien enthalten, die im BDD-Stil formuliert sind.

- **Kollaboration**

Entwickler, Tester und Business-Anwender arbeiten zusammen an der Definition der Akzeptanzkriterien. Dies fördert ein gemeinsames Verständnis der Anforderungen.

- **Testautomatisierung**

Die Akzeptanzkriterien können als Grundlage für automatisierte Tests dienen, was die Qualitätssicherung während des gesamten Entwicklungsprozesses unterstützt.

1.6 Vergleich zwischen Kanban und Scrum

Aspekt	Kanban	Scrum
Framework	Flexibles Framework ohne feste Rollen	Striktes Framework mit definierten Rollen
Rollen	Keine festen Rollen; Teammitglieder übernehmen verschiedene Rollen	Feste Rollen: Product Owner, Scrum Master, Entwicklungsteam
Planung	Kontinuierliche Planung; keine Sprints	Sprint-basierte Planung (typischerweise 2-4 Wochen)

Aspekt	Kanban	Scrum
Work-in-Progress	WIP-Limits zur Vermeidung von Überlastung	Keine spezifischen WIP-Limits; Fokus auf Sprint-Ziele
Änderungen	Änderungen können jederzeit vorgenommen werden	Änderungen sind während eines Sprints eingeschränkt
Meetings	Regelmäßige Meetings nach Bedarf (z.B. Stand-ups)	Feste Meetings: Sprint Planning, Daily Scrum, Sprint Review, Sprint Retrospective
Zielsetzung	Kontinuierlicher Fluss und Lieferung	Lieferung am Ende jedes Sprints

1.7 Vor und Nachteile gegenüber Scrum mit dem Fokus BDD

1.7.1 Vorteile

- **Effizienter Workflow durch Visualisierung des Arbeitsflusses und WIP-Limits**

Das Kanban-Board hilft dabei, den Fortschritt sichtbar zu machen und Engpässe im Workflow zu identifizieren.
Die Begrenzung der Work-in-Progress (WIP) sorgt dafür, dass das Team sich auf die Fertigstellung von Aufgaben konzentriert, bevor neue begonnen werden.

- **Bessere Nachverfolgbarkeit durch transparente Anforderungen und Testdokumentation**

Die Verwendung von BDD macht es einfacher, Anforderungen nachvollziehbar zu gestalten und eine Erfüllung sicherzustellen.
Die Tests selbst dienen als lebendige Dokumentation der Anforderungen und deren Beachtung.

- **Flexibilität**

Keine festen Iterationen oder sprints daher sehr schnelle Reaktion der Teams auf Änderungen in den Anforderungen. Prioritäten können dynamisch verschoben werden.

1.7.2 Nachteile

- Weniger Struktur und fehlende feste Rollen

Kanban hat keine definierten Rollen wie beispielsweise Scrum Master oder Product Owner. Dies kann zu Unklarheiten in der Verantwortlichkeit führen.

- Schwierigkeiten bei der Priorisierung

Das Fehlen von Sprint-Planung kann es erschweren, klare Prioritäten zu setzen. Die Sicherstellung und Priorisierung der wichtigsten Aufgaben für das Team ist somit schwieriger.

- Herausforderungen bei der Teamdynamik durch fehlende Rituale

Die fehlenden regelmäßigen Meetings in Kanban können zu einem mangelnden Austausch zwischen den Teammitgliedern führen. Die Zusammenarbeit sollte daher anders gestärkt werden.

1.8 Klare Testszenarien durch Abstraktion

1.8.1. Hig Level Requirements

- **Hohe Ebene der Abstraktion**

Anforderungen werden high Level formuliert. Sowohl das Verhalten als auch die Benutzererfahrung stehen im Fokus. Technische Details verbergen sich hinter den ausformulierten BDD Schritten.

- **Szenarien als Abstraktion**

Szenarien beschreiben allgemeine Verhaltensmuster, die für möglichst verschiedene Anwendungsfälle gelten. Dadurch sind Schritte wiederverwendbar und spezifische Implementierungsdetails werden ausgeblendet.

- **Kontextualisierung**

Anforderungen werden im Kontext des gesamten Systems betrachtet, um sicherzustellen, dass sie auch in verschiedenen Situationen relevant sind.

1.8.2 Given-When-Then

- Szenarien so allgemein wie möglich, aber spezifisch wie nötig

Szenarien werden so formuliert, dass sie allgemeine Verhaltensmuster darstellen, anstatt spezifische Implementierungsdetails zu beschreiben.

- Wiederverwendbarkeit

Die Struktur ermöglicht es, ähnliche Szenarien zu abstrahieren und wiederzuverwenden. Nur die relevantesten Teile sind anzupassen und gegebenenfalls durch Variablen oder Datentabellen allgemeingültig.

- Kontextualisierung durch Given

Der "Given"-Teil kann mehrere Bedingungen enthalten, um verschiedene Kontexte abzudecken. Dadurch entsteht eine abstrakte Sicht auf verschiedene Ausgangssituationen.

- Flexibilität durch When

Der "When"-Teil kann verschiedene Aktionen und Logiken umfassen. Somit fließen unterschiedliche Interaktionen in den Test mit ein.

- Erwartungen im Then

Der "Then"-Teil definiert die Erwartungen an das Systemverhalten und das erwartete Ergebnis.

1.8.3 Trennung von Logik und Implementierung

Bei dieser Trennung ist es das Ziel, die Geschäftslogik (was die Software tun soll) von der technischen Implementierung (wie es umgesetzt wird) zu trennen.

Dies fördert eine klare Kommunikation zwischen Entwicklern, Testern und Fachleuten.

1.8.4 Wiederverwendbare Testfälle

Wie bereits erwähnt, ermöglicht es BDD, die Szenarien so zu gestalten, dass sie wiederverwendet werden können. Sowohl in anderen Testfällen als auch in anderen Projekten. Dies führt zu einer verkürzten Entwicklungszeit und erleichterten Wartung. Bei Änderungen profitieren direkt alle abhängigen Tests und Szenarien davon.

1.8.5 Zielgerichtete Kommunikation

BDD fördert eine klare und zielgerichtete Kommunikation zwischen Entwicklern, Testern, Fachleuten und anderen Stakeholdern. Dabei werden auch alle Projektbeteiligten stets eingebunden und können Sachverhalte nachvollziehen. Durch die verständliche Sprache sind alle Beteiligten in der Lage, Anforderungen in der verständlicher Form zu erfassen und zu formulieren. Dabei unterstützt auch die Gherkin-Syntax (Given, When, Then). Somit werden auch Missverständnisse reduziert.

1.9 Testarten

1.9.1 Akzeptanztests

Akzeptanztests überprüfen, ob das System die definierten Anforderungen und Akzeptanzkriterien erfüllt. Diese Tests werden oft direkt aus den Benutzerstories und deren Akzeptanzkriterien abgeleitet und sind ein zentrales Element von BDD. Sie helfen dabei, das Verhalten des Systems aus der Sicht des Endbenutzers zu validieren.

1.9.2 Integrationstests

Integrationstests überprüfen, ob verschiedene Module oder Komponenten eines Systems korrekt miteinander zusammenarbeiten. In BDD ist durch die Szenarien leicht erfasst, welche Module und Komponenten miteinander interagieren. Im Testbericht ist dabei leicht ersichtlich, ob dieser Prüfung erfolgreich war.

1.9.3 Systemtests

Systemtests testen das System als Ganzes, um sicherzustellen, dass es den spezifizierten Anforderungen entspricht. Diese Tests können auf Basis der in BDD formulierten Szenarien erstellt werden und helfen dabei, das Gesamtverhalten des Systems zu validieren. Hier kommen auch die Vorteile der Wiederverwendung vorhandener Szenarien-Schritte zu tragen. Vorhandene Integrationstests können erweitert werden, um ganze Systeme abzubilden.

1.9.4 End-to-end tests

End-to-End-Tests simulieren reale Benutzerinteraktionen mit dem System von Anfang bis Ende, also der Oberfläche bis zur Persistenzschicht. Diese Tests sind besonders wichtig in BDD, da sie das gesamte Benutzererlebnis abdecken und sicherstellen, dass alle Teile des Systems nahtlos zusammenarbeiten. Das Userfeedback kann dabei ebenfalls als Anforderung abgeprüft und in Szenarien abgebildet werden.

1.9.5. Unit-tests

Unit-tests überprüfen einzelne Komponenten oder Funktionen eines Systems isoliert. Obwohl Unit-Tests nicht direkt Teil von BDD sind, können sie dennoch hilfreich sein, um sicherzustellen, dass die kleinsten Bausteine des Codes korrekt funktionieren. In vielen Fällen können Unit-Tests auch aus den Verhaltensspezifikationen abgeleitet werden und innerhalb weniger Schritte ein Szenario bilden.

1.9.6. Regressionstests

Regressionstests stellen sicher, dass neue Änderungen im Code keine bestehenden Funktionen oder Verhaltensweisen beeinträchtigen. Bestehenden BDD-Szenarien eignen sich sehr gut, um bei regelmäßiger Ausführung in eine Regressionstestsuit aufzunehmen. Sie helfen dabei, sicherzustellen, dass das System nach Änderungen weiterhin wie erwartet funktioniert und veranschaulichen die Feedback-Loop dabei.

1.9.7. Performance-Tests

Performance-Tests messen, wie gut ein System unter bestimmten Lastbedingungen funktioniert. Während BDD sich hauptsächlich auf das Verhalten konzentriert, ist auch die Leistung des Systems ein entscheidender Aspekt. Diese Art von Test lässt sich ebenfalls in BDD Szenarien integrieren und lässt sich daher mit abtesten.

1.9.8. Sicherheitstests

Sicherheitstests überprüfen ein System auf Schwachstellen und evaluieren den Bedrohungsfaktor. Auch wenn Sicherheit nicht immer im Mittelpunkt von BDD steht, können Sicherheitsaspekte in die Akzeptanzkriterien aufgenommen werden. Beispielsweise könnte ein Szenario beschreiben, wie ein Benutzer auf sensible Daten zugreifen kann und welche Berechtigungen notwendig sind. In einer Datentabelle könnten verschiedene Rollen für diesen Test übergeben werden.

1.9.9. Usability-Tests

Usability-Tests bewerten die Benutzerfreundlichkeit und das Benutzererlebnis eines Systems (User-Experience). Da BDD stark benutzerzentriert ist, können Usability-Aspekte in die Szenarien integriert werden. Diese Tests helfen dabei zu überprüfen, ob das System intuitiv und einfach zu bedienen ist.

1.10 Data-Driven Testing (DDT)

Datadriven Testing ist eine Teststrategie, bei der die gleichen Testfälle mit verschiedenen Eingabewerten und erwarteten Ergebnissen ausgeführt werden. Es ermöglicht die Überprüfung der Funktionalität unter verschiedenen Bedingungen, ohne dass für jede Kombination ein neuer Testfall geschrieben werden muss. Nur durch die Vorgabe anderer Datensätze ändert sich dabei das Testverhalten.

In BDD lassen sich datadriven Tests durch Parametrisierung von "Given-When-Then"-Szenarien realisieren, wodurch Szenarien flexibler und wiederverwendbarer werden. (Nutzung von Parametern und Datentabellen, Siehe Kapitel 2)

Die Testdaten werden auch als Datenquellen bezeichnet und können aus verschiedenen Quellen stammen, wie z.B. CSV-Dateien, Datenbanken oder Excel-Tabellen. Durch die Trennung von Testlogik und Testdaten wird die Wartung erleichtert und die Wiederverwendbarkeit erhöht. Neue Testdaten lassen sich einfach hinzufügen und ermöglichen somit weitere Testszenarien ohne Codeanpassungen. Erweiterungen und Veränderungen sind schnell durch einfaches Ergänzen oder Entfernen von Datenzeilen möglich.

1.11 Keyword-Driven Testing (KDT)

Beim Keyword-driven Testing erfolgt eine Definition von Testfällen durch Schlüsselwörter (Keywords). Diese repräsentieren spezifische Aktionen oder Funktionen. Es ermöglicht eine abstrahierte und benutzerfreundliche Möglichkeit, Tests zu erstellen, ohne die tiefgehende Nutzung von Programmierkenntnissen.

In BDD können Keywords verwendet werden, um die Schritte in "Given-When-Then"-Szenarien zu definieren, was die Lesbarkeit und Verständlichkeit erhöht. Keywords repräsentieren oft wiederverwendbare Testaktionen. Auch dabei erfolgt wieder eine Trennung von Testlogik und Testdaten. Durch neue Keywords können zusätzliche Funktionalitäten oder Szenarien abgedeckt werden. Auch bestehende Tests sind dadurch leicht mit neuen Keywords zu erweitern. Die Wiederverwendung und Flexibilität ist hier ebenfalls ein weiterer positiver Aspekt.

1.12 Tools und Frameworks

Weitere Tools und Frameworks im BDD Umfeld sind: FitNesse, Robot Framework, Jbehave, Sikuli, Jspec.

1.13 Vor- Nachteile von BDD

- Vorteile:

BDD fördert durch die gemeinsame Sprache und die gemeinsame Quelle der Wahrheit die Zusammenarbeit. Da die Anforderungen auch gleichzeitig als Basis für die Tests herangezogen werden, entsteht eine lebende Dokumentation, die nicht veralten kann. Außerdem wird dadurch auch sichergestellt, das Richtige zu entwickeln und Missverständnisse aus dem Weg zu räumen. Alle beteiligten Parteien können stets mitreden und ihren Input liefern. Zudem sind durch die für jeden ersichtliche Dokumentation kurze Antwortzyklen und Verbesserungen möglich.

- Nachteile:

Durch die Einführung von BDD entstehen zunächst hohe Initialkosten. Teams sind unter Umständen nicht an das Vorgehen gewöhnt und benötigen eine gewisse Zeit. Zudem muss unter den Teams eine Grundharmonie herrschen, denn sonst wird BDD dort keinen Erfolg haben. Hohe Absprachen sind notwendig und somit ist diese Methode auch nicht für alle geeignet. Um das richtige richtig zu entwickeln, ist auch stets der Austausch mit den Stakeholdern notwendig. Sind diese schlecht verfügbar, könnte BDD

auch problematisch sein.

1.14 Popularity of BDD

Nachfolgend einige Aspekte, die Behaviour Driven Development als Vorgehensmethode so beliebt macht. Diese wurden bereits im vergangenen Abschnitt vorgestellt und dienen hier nur noch mal zur Wiederholung.

1.14.1 Lesbarkeit

Die Verwendung von natürlicher Sprache in "Given-When-Then"-Szenarien macht Tests für alle Beteiligten verständlich, auch für Nicht-Techniker.

1.14.2 Agilität

BDD unterstützt agile Methoden, indem es schnelle Iterationen und einfache Anpassungen an sich ändernde Anforderungen ermöglicht.

1.14.3 Wiederverwendbarkeit

Testszenarien können leicht wiederverwendet und erweitert werden, was die Wartung erleichtert und die Übersichtlichkeit fördert.

1.14.4 Zusammenarbeit

BDD fördert die Zusammenarbeit zwischen Entwicklern, Testern und Fachabteilungen, was zu besserem Verständnis und weniger Missverständnissen führt.

1.14.5 Dokumentation

Die Szenarien dienen gleichzeitig als Dokumentation der Anforderungen und des Verhaltens der Software. Wie bereits angesprochen ist diese Dokumentation stets aktuell und durch das einhalten eines gemeinsamen Ablageortes ersteht eine gemeinsame Quelle der Wahrheit.

Kapitel 2 – Testautomatisierung mit Cucumber

Referenz

[MS1] [CC1]

Schlüsselbegriffe

Gherkin, Datatables, TestNG, Feature-Dateien, Step-Dateien

LO-2.1	K1	Die Teilnehmer sind in der Lage, eine Testumgebung für Cucumber und TestNG einzurichten.
LO-2.2	K2	Zeigt die Vorteile von TestNG für das Testmanagement und die Ausführung von Testszenarien auf.
LO-2.3.1	K3	Ermöglicht es den Teilnehmern, eigene Feature Files zu erstellen und diese in einem praktischen Beispiel anzuwenden.
LO-2.3.2	K3	Teilnehmer können eigene Step Definitions schreiben und diese mit ihren Feature Files verknüpfen
LO-2.3.3	K2	Erläutert, wie beide Testarten in Cucumber implementiert werden können und was dabei zu beachten ist.
LO-2.3.4	K1	Beschreibt die Rolle von Hooks in Cucumber und wie sie zur Bestimmung von Reihenfolgen in Testszenarien verwendet werden.
LO-2.3.5	K2	Behandelt verschiedene Konfigurationen für Runner Classes, einschließlich der Auswahl spezifischer Features oder Tags
LO-2.3.6	K3	Teilnehmer sind in der Lage parametrisierten Testszenarien in Cucumber zu erstellen. Sie lernen, wie man Parameter in Gherkin-Szenarien definiert und diese in Step Definitions verwendet, um flexible und wiederverwendbare Tests zu entwickeln.
LO-2.3.7	K3	Zeigt den Teilnehmenden wie Data Tables in ihren Feature Files implementiert werden. Sie üben das Erstellen von Testszenarien mit Data Tables und lernen, wie man strukturierte Daten effektiv in Cucumber verwendet.
LO-2.3.8	K3	Fördert das Verständnis und die Anwendung komplexer Data Tables. Die Teilnehmer erstellen Szenarien mit verschachtelten oder mehrdimensionalen Datenstrukturen und lernen, wie diese in Cucumber eingesetzt werden.

2.1 Setting up the Test Environment software

Die Testumgebung ist die Grundlage für automatisierte Tests. Sie umfasst alle notwendigen Komponenten, um Tests zu entwickeln und ausführen zu können. In dieser Testumgebung liegen benötigte Tools zur Automatisierung und die notwendigen Komponenten, um auf das System under Test (SuT) zugreifen zu können.

Die Testumgebung besteht für diesen Syllabus aus Java, TestNG, Maven und Cucumber. Die erfolgreiche Installation von Java und einer IDE wird dabei als gegeben betrachtet.

TestNG ist ein beliebtes Test-Framework für Java. Es unterstützt das Schreiben von Unit-Tests und Integrationstests. Auch eine Ausführung ist damit möglich. Für die gängigsten Entwicklungsumgebungen wie z. B. IntelliJ oder Eclipse bietet TestNG zugehörige Plugins. Mit diesen lassen sich Test-Konfigurationen leicht erstellen und verändern. Über die zugehörigen Marketplaces sind diese komfortabel zugänglich.

Maven ist ein Build-Management Werkzeug. Es vereinfacht die Verwaltung von Abhängigkeiten und externen Bibliotheken. Über eine zentrale Konfiguration, die als POM.xml bezeichnet wird, lassen sich Build-Prozesse konfigurieren und externe Komponenten integrieren.

Cucumber vervollständigt diese Umgebung. Die Verwendung der Gherkin Syntax wird durch ein externes Plugin unterstützt. Auch dieses Plugin ist für die gängigsten IDEs über den Marketplace erhältlich und bietet neben Syntaxhighlighting auch die Möglichkeit, Featurefiles als Test direkt auszuführen. Die Externen Dependencies für Cucumber sind über das Maven Central Repository erhältlich.

2.2 TestNG

[TE1]

TestNG ist ein umfangreiches Test-Framework für Java. Es bietet eine Vielzahl von Annotationen und Funktionen, um das Testen von Anwendungen zu erleichtern. Nachfolgend einige der wichtigsten Annotationen und Konzepte in TestNG:

2.2.1 @BeforeClass

Die Annotation @BeforeClass wird verwendet, um eine Methode einmalig vor dem Beginn aller Tests in einer bestimmten Klasse auszuführen. Dies ist vorteilhaft, um z. B. Komponenten vor einem Test einmalig hochzufahren. Ein gängiges Beispiel ist das Starten eines Browsers oder der Verbindungsaufbau zu einer Datenbank.

2.2.2 @BeforeMethod

Die Annotation @BeforeMethod kennzeichnet eine Methode, die vor jeder einzelnen Testmethode innerhalb der Klasse ausgeführt wird. Dies ist ideal für Aufgaben wie das Zurücksetzen von Variablen oder das Erstellen neuer Instanzen von Objekten, die für jeden Test benötigt werden.

2.2.3 @AfterClass

Die Annotation @AfterClass wird zum Kennzeichnen einer Methode verwendet, die einmal nach der Ausführung aller Tests in einer Klasse aufgerufen wird. Diese Methode eignet sich hervorragend zum Aufräumen von Ressourcen oder zum Schließen von Verbindungen, Beispielweise einer Testdatenbank.

2.2.4 @AfterMethod

Die @AfterMethod Annotation kennzeichnet eine Methode, die nach jeder einzelnen Testmethode ausgeführt wird. Dies stellt sicher, dass nach jedem Test bestimmte Zustände zurückgesetzt werden. Beispielsweise beim Zurücksetzen eines Browserstatus.

2.2.5 Testgruppen (Groups)

TestNG ermöglicht es, Tests in Gruppen einzuteilen. Dadurch lassen sich eingruppierte Tests entweder ausführen oder direkt ausschließen. Dies ist besonders nützlich für große Projekte mit einer Vielzahl von Tests. So können gezielt nur bestimmte Tests ausgeführt werden. Ein Gängiger Anwendungsfall hierfür ist die Eingruppierung von Tests in eine Smoke-Test-Gruppe.

Um eine Gruppe zu definieren, wird die groups-Eigenschaft in der @Test-Annotation verwendet.

Diese Gruppierungen lassen sich bei der Testausführung angeben.

2.2.6 Parameters

TestNG unterstützt auch Parameter, um Werte an Testmethoden zu übergeben. Insbesondere für die Durchführung von Testmethoden mit unterschiedlichen Eingabewerten ist dies nützlich.

Diese Parameter können in der XML-Konfigurationsdatei definiert werden.

In der Testklasse können auf diese Parameter mit der Annotation @Parameters zugegriffen werden:

Insgesamt liefert TestNG eine strukturierte und effiziente Möglichkeit, Tests zu organisieren und durchzuführen. Dadurch steigt sowohl die Qualität als auch die Zuverlässigkeit der

Softwareanwendungen.

2.2.7 Assert-Methoden in TestNG

TestNG bietet eine Reihe von Assert-Methoden um Testergebnisse zu prüfen. Diese Methoden vergleichen dabei erwartete Resultate mit den tatsächlichen Ergebnissen. Im nachfolgenden wird eine Auswahl der mit am häufigsten verwendeten behandelt.

AssertEquals

Die Methode assertEquals wird zur Gleichheitsprüfung zweier Werte angewendet. Stimmen diese Werte nicht überein, schlägt der Test fehl.

assertNotEquals

Die Methode assertNotEquals ist das Gegenteil von assertEquals und prüft auf Ungleichheit zweier Werte.

assertTrue

Mit der Methode assertTrue wird ein Boolischer Ausdruck auf true geprüft. Wenn der Ausdruck false ist, schlägt der Test fehl.

In diesem Beispiel wird überprüft, ob die Bedingung (5 > 3) true ist. Dies ist der Fall und der Test somit erfolgreich.

assertFalse

Die Methode assertFalse wird verwendet, um einen Ausdruck auf false zu prüfen. Ist Bedingung true, schlägt der Test fehl.

2.3 Cucumber

Wie bereits erwähnt ist Cucumber eines der bekanntesten BDD Tools. Es verwendet die Gherkin Syntax, um Testszenarien zu beschreiben. Diese Syntax wurde bereits in Kapitel 1 beschrieben und ist sowohl für Entwicklungspersonal als auch Fachleuten ohne technischen Hintergrund einfach zu lesen. Mit Cucumber sind dieses Testfälle ausführbar. Die Szenarien werden in Feature-Files abgelegt.

2.3.1 Feature-Files

Feature-Files sind einfach Textdateien mit der Endung .feature. Sie werden in der Gherkin-Syntax geschrieben und beinhalten sowohl Anforderungen als auch zugehörige Testdaten. Ein Szenario spiegelt dabei einen Testfall wieder. Sie dienen dazu, die Funktionalitäten aus der Sicht des Endbenutzers zu beschreiben. Diese Schritte sind mit sogenannten Steps verbunden und ermöglichen somit eine Zugehörigkeit zu ausführbarem Programmcode. Mit den gängigsten Cucumber Plugins für IDEs werden bei einer Ausführung noch nicht implementierte step Definitionen angelegt.

2.3.2 Step Definitions

Step Definitions oder auch „Glue Code“ sind ausführbarer Code, der die Schritte in den Feature Files mit der tatsächlichen Implementierung verbindet. Jeder Schritt in einem Feature File wird durch eine Step Definition wiedergegeben. Sie beschreibt, was bei der Ausführung dieses Schrittes genau geschehen soll. Innerhalb eines jeden Steps befindet sich zugehöriger Programmcode mit Aktionen und möglicher Überprüfung. Somit können auch Frameworks und Bibliotheken von anderen Anbietern einfach in die Tests eingebunden werden.

2.3.3 UI- & Fachliche Tests

Auch UI-Testing Bibliotheken wie Selenium oder Playwright lassen sich leicht in Step Definitionen integrieren. Die UI-Tests konzentrieren sich dabei auf die Interaktion des Benutzers mit der Anwendung,

während die fachlichen Tests sicherstellen, dass die Geschäftslogik korrekt funktioniert. Die Vorgangsbeschreibung bei diesen Tests kann für den Fachbereich deutlich komfortabel aufzeigen, welche Aktionen genau durchgeführt werden und wie ein Workflow aussieht.

2.3.4 Hooks

Hooks sind spezielle Funktionen in Cucumber um bestimmte Aktionen vor oder nach der Ausführung von Testszenarien durchzuführen. Vergleichbar sind diese Funktionen mit den Before- und After-Annotationen von TestNG. Sie helfen dabei, wiederkehrende Aufgaben wie das Laden von Testdaten oder das Bereinigen von Ressourcen zu automatisieren. Eine gängige Funktion ist das starten des Browsers oder das Öffnen einer häufig verwendeten URL vor dem eigentlichen Testbeginn. Im Gegensatz zu TestNG erfolgt eine Ausführung auf Szenarioebene. Wesentlich sind drei Before- Hooks und drei After-Hooks. Der BeforeAll Hook ist sinnvoll, um etwas einmalig vor einem Szenario auszuführen. Der Before Hook hingegen wird einmalig vor jedem ersten Schritt eines Szenarios umgesetzt, während before Step vor jedem einzelnen Schritt erfolgt. Äquivalent verhält es sich mit den After- Hooks. Zudem können noch Reihenfolgen mit dem order = Reihenfolgen Nummer, übergeben werden. Somit lassen sich mehrere Hooks in eine definierten Hierarchie übergeben.

2.3.5 Runner Classes

Runner Classes sind Java-Klassen, die für das Ausführen von Cucumber-Tests verantwortlich sind. Sie ermöglichen es den Entwicklern, spezifische Features oder Tags auszuwählen und verschiedene Konfigurationen für den Testlauf festzulegen. Durch diese Konfigurationen können Plugins integriert, Testreports erstellt oder bestimmte Testgruppen über Tags ausgeschlossen werden. Innerhalb dieser Runner Class sind die Pfade der Feature-Files und des Glue-Codes anzugeben. Über die Plugin- Option können auch diese angegeben werden.

2.3.6 Parametrisierung

Parametrisierung ermöglicht die Ausführung von Testszenarien mit verschiedenen Eingabewerten. Eine Duplizierung oder Anpassung des Codes ist somit nicht notwendig. Vergleichbar ist dies mit der Nutzung von Variablen in einer Programmiersprache. Mit “Inhalt einer Variable“ wird die der Inhalt im Feature-File definiert und an den Glue Code übergeben. Im Stepfile wird an die zugehörige Stelle eine Variable eingefügt und kann im folgenden Programmcode der Methode verwendet werden.

Zusätzlich kann das "Background"-Keyword verwendet werden, um gemeinsame Schritte für alle Szenarien in demselben Feature-File festzulegen. Dieses Keyword sorgt dafür, dass die definierten Steps innerhalb eines jeden Szenarios vor dem ersten Szenario-Step ausgeführt werden. Somit einfällt eine Wiederholung dieser für folge Szenarien. Dadurch wird der Testcode übersichtlicher und wartungsfreundlicher, da alle Szenarien auf eine einheitliche Ausgangsbasis zurückgreifen können.

2.3.7 Data Tables

Data Tables in Cucumber sind eine Möglichkeit, Daten strukturiert in Testszenarien zu verwenden. Diese Daten können sowohl mehrere Eingabewerte als auch erwartete Ergebnisse sein. Diese Informationen werden in tabellarischer Form im Feature-File angegeben. Data Tables können unter einem jeweiligen Test Schritt stehen als auch am Ende des Szenarios. Auf jedes einzelne Tabellenelement kann auch im Programmcode zugegriffen werden und somit den Ablauf beeinflussen.

Der Mehrwert von Data Tables liegt darin, dass sie die Wiederverwendbarkeit und Wartbarkeit von Testszenarien verbessern, da nicht für jede Kombination von Eingaben ein separates Szenarien benötigt wird. Stattdessen können mehrere Datensätze in einer einzigen Tabelle zusammengefasst werden, was den Testaufwand reduziert und gleichzeitig die Abdeckung erhöht.

2.3.8 Komplexe Data Tables, Mischformen

Komplexere Data Tables benötigen das die Erweiterung „Scenario Outline“ und das „Examples“ Keword. Somit wird dem Scenario signalisiert, es soll mit der unter Examples angefügten Datentabelle verarbeitet werden. Die Überschriften der Tabelle signalisieren dabei die verwendete Spalte. Diese Überschrift kann mit spitzen Klammern in der Szenariobeschreibung verwendet werden. Durch diese Schreibweise wird die eingesetzte Überschrift bei der Ausführung durch den zugehörigen Tabelleninhalt ersetzt. Pro Zeile erfolgt dabei ein Szenariodurchlauf. Auch eine Kombination aus Data Tables und Example Table ist möglich. Damit kann unter einem Szenario Step eine Tabelle mit Variablen verwendet werden. Wird also ein weiterer Testdurchlauf mit anderen Daten benötigt, genügt es eine neue Zeile in der Tabelle zu ergänzen.

Kapitel 3 – Der BDD Zyklus - 100 Minuten

Schlüsselbegriffe
TDD (Test-Driven Development), ATDD (Acceptance Test-Driven Development)

LO-3.1	K2	Die Teilnehmer verstehen, wie BDD zur Verbesserung der Zusammenarbeit zwischen Entwicklern und Fachabteilungen beiträgt.
LO-3.2	K1	Die Teilnehmer können die Definition und die Hauptmerkmale von TDD wiedergeben.
LO-3.3	K3	Die Teilnehmer können Szenarien beschreiben, in denen jede Methode am besten geeignet ist.
LO-3.4	K3	Die Teilnehmer können ein einfaches Beispiel für jede Phase des BDD-Zyklus erstellen und erläutern, wie diese in einem Projekt umgesetzt werden.
LO-3.5	K2	Die Teilnehmer verstehen, warum bestimmte Vor- oder Nachteile für unterschiedliche Projekte relevant sein können.

3.1 Motivation für den Behaviour Driven Development Zyklus

BDD zielt darauf ab, die Zusammenarbeit zwischen Entwicklern, QA und Stakeholdern zu verbessern. Im Verlauf dieses Syllabus wurde bereits der Einsatz von Behaviour Driven Development vorgestellt. Bei diesem Einsatz lag der Fokus jedoch auf dem Erstellen der Spezifikation. Wenn die Hauptakteure nur beim Erstellen dieser Spezifikation miteinander agieren, jedoch bei Test und Entwicklung nicht mehr involviert sind, bleiben Chancen ungenutzt. Stakeholder sollten sowohl bei der Testfallerstellung, ihrer Ergebnisanalyse als auch bei der Featureentwicklung miteinbezogen werden. Dies bedeutet nicht, dass sie aktiv mit programmieren sollen, aber mit Informationen und ihren Sichtweisen unterstützen und potenzielle Missverständnisse auch hier frühzeitig aus dem Weg räumen.

Beispielsweise könnte ein Stakeholder einen Testbericht betrachten und die entsprechenden Testschritte überprüfen. Dieser Abgleich mit der Featurevision zeigt direkt auf, ob hier irgendwelchen Lücken bestehen.

Alle Beteiligten sollten in jedem Schritt mit agieren können. Eine solche Kooperation muss während des gesamten Entwicklungsprozess erfolgen. Auch die Prüfung, ob das Richtige richtig entwickelt wird, ist dadurch leichter möglich. Somit kann auch sicher gestellt werden, dass alle die Spezifikation verstanden haben.

3.2 Wiederholung Test Driven Development

TDD ist eine Methode, bei der passende Tests vor dem Schreiben des eigentlichen Codes erstellt werden. Der Prozess besteht aus drei Hauptschritten:

- **Test schreiben**: Zuerst wird ein Test geschrieben, der beschreibt, was der Code tun soll.
- **Code implementieren**: Dann wird der Code geschrieben, um diesen Test bestehen zu lassen.
- **Refactoring**: Schließlich erfolgt eine Codeoptimierung, ohne die Funktionalität zu verändern.

Diese Abfolge hilft dabei, Fehler frühzeitig zu erkennen und Anforderungen zielgerichtet umzusetzen.

Vorteile
Durch den Test-First-Ansatz erfolgt eine frühe Fehlererkennung, da passende Tests bereits bestehen. Die Wartbarkeit wird ebenfalls erleichtert, da bei jeder Codeänderungen passende Tests direkt durchgeführt werden können, um die korrekte Funktionalität sicherzustellen. Tests erfolgen im ersten Schritt der Methode und können daher auch nicht im Nachhinein vergessen werden.

Nachteile
Diese Methode ist für einige gewöhnungsbedürftig und bedarf einer gewissen Eingewöhnungszeit, daher ist sie gerade zu Beginn vergleichsweise zeitintensiv. Für kleine triviale Projekte könnte diese Art ungeeignet sein, da sie auch einen gewissen Overhead erzeugt. Des weiteren kann ein falsches Gefühl der Sicherheit entstehen, weil nicht zwangsläufig alle Teile des Codes abgedeckt sind und auch nicht alle relevanten Testfallkonstellationen in einen solchen Test fallen. Zu beachten gilt ebenfalls eine Lernkurve bei der Anwendung dieser Methodik.

3.3 Unterschiede zwischen BDD, TDD und ATDD

Acceptance Test Driven Development (kurz: ATDD) ist eine Vorgehensweise, bei der Akzeptanzkriterien und Akzeptanztests bereits vor der eigentlichen Implementierung erstellt werden. Dabei arbeiten Entwickler, Tester und der Fachbereich gemeinsam daran, die Anforderungen in Form von Tests zu formulieren. Diese Vorgaben sollen das gewünschte Verhalten der Software beschreiben. Ziel ist die Sicherstellung, dass alle vorgegebenen Anforderungen auch erfüllt sind.

3.3.1 ATDD Entwicklungsprozesses

Begonnen wird mit der Anforderungsanalyse. Hier arbeiten Entwickler, Tester und der Fachbereich zusammen, um sowohl Kriterien als auch Testszenarien für ein Feature zu definieren. Nach der Analyse erfolgt die Definition von Akzeptanztests. Diese Tests prüfen das gewünschte Verhalten des Features ab. Anschließend erfolgt die eigentliche Implementierung des Features. Zuletzt werden die Tests ausgeführt. Wurde diese Phase erfolgreich abgeschlossen, kann ein optionales Refactoring stattfinden oder die Funktionalität gilt als umgesetzt. Bei Fehlschlägen erfolgen die notwendigen Codekorrekturen.

3.3.2 Vor- und Nachteile

ATDD verbessert die Kommunikation und ermutigt die Einbeziehung aller Beteiligten. Die eigentlichen Features entsprechen eher den Anforderungen, da diese zuerst gesetzt werden und als Nachschlagewerk zur Verfügung stehen. Missverständnisse können früh erkannt und behoben werden.

Bei ATDD wird eine enge Zusammenarbeit benötigt, da die Kommunikation äußerst wichtig ist. Ohne diese Absprachen ist eine zielführende Umsetzung kaum möglich. Die präzisen und für alle verständlichen Formulierungen können schwer und zeitaufwendig sein.

3.3.3 Vergleich zwischen BDD, TDD und ATDD

Ziel:
TDD legt den Fokus auf die Funktion des Codes, BDD prüft hingegen, ob das gewünschte Verhalten erfüllt ist, während ATDD den Fokus auf die Anforderungen der Stakeholder legt.

Fokus:
TDD findet überwiegend Anwendung auf Unittest bzw. Modultest Ebene, während BDD das Verhalten des gesamten Systems im Fokus hält. ATDD hat dabei eher die Akzeptanztests und Benutzeranforderungen im Vordergrund.

Testbeschreibung:
Die Beschreibung von TDD Szenarien ist eher technisch und findet meist im Code statt, während BDD in natürlicher Sprache, häufig Gherkin beschrieben ist. Auch ATDD Szenarien werde eher in natürlicher Sprache beschrieben.

Beteiligte:
Während TDD klar Entwickler im Fokus hat, ist es bei BDD und ATDD Entwickler, Test, PO und bei ATDD zusätzlich die Stakeholder.

Kollaboration:
TDD hat wenig Kollaboration im Fokus, wenn eher auf Entwicklerseite. Bei BDD und ATDD ist diese eher hoch und bezieht das gesamte Team mit ein.

Anwendungsbereiche:

Bei den Anwendungsbereichen wird TDD am häufigsten für Komponenten, Funktionen und Module eingesetzt. BDD hingegeben ist allgegenwärtig von einzelnen Unittests über API Test oder UI Tests. ATDD hat den Fokus auf Anforderungsvalidierung und findet hier am häufigsten den Einzug.

3.4 Behavior-Driven Development Prozess

Der BDD Entwicklungsprozess setzt auf den zuvor beschriebenen Prozessen auf. Zuerst wird mithilfe der Give-When-Then Syntax das gewünschte Verhalten beschrieben. Durch diese Vorgaben lässt sich auch schon der Gluecode erzeugen. Wie in bereits im behandelten TDD Vorgehen vorgestellt, folgen die ersten Tests, die allerdings noch failen solange bis der zugehörige Produktionscode besteht. Nachdem die Testfälle grün sind, beginnt die Codeoptimierungs und Refactoring phase. Anschließend endet der Prozess oder beginnt von neuem mit der Verbesserung.

3.5 Vor- und Nachteile

Dieser Prozess legt wieder großen Wert auf die Zusammenarbeit zwischen den verschiedenen Bereichen, ohne diese wird eine Umsetzung schwer. Dieses Vorgehen bietet sehr schnelles Feedback, zeigt auf, ob das richtige richtig entwickelt wurde und unterstützt die lebende Dokumentation, die hinter BDD steckt. Entwickler erhalten eine gewisse Sicherheit, dass sie ein Feature korrekt und getestet umsetzen. Auch Refaktoring-Änderungen sind davon abgedeckt und können nahezu gefahrlos durchgeführt werden. Alle Parteien sind von Anfang bis zum ende eines Features stets eingebunden.

Wie bei BDD bekannt, bestehen auch hier zunächst hohe Initialkosten und eine entsprechende Lernkurve. Die notwendige Absprache ist sehr hoch und muss von allen Seiten akzeptiert werden. Ohne die Mitwirkung aller Beteiligten wird es sehr schwer, diesen Prozess umzusetzen.

Kapitel 4 – APIs, Mocking & BDD mit Karate

Schlüsselbegriffe

APIs, Mocking, JSON, SOAP, REST

LO-4.1	K2	Die Teilnehmer können die grundlegenden Konzepte von APIs und Mocks beschreiben und die Unterschiede zwischen REST und SOAP erläutern.
LO-4.2	K3	Die Teilnehmer können die Hauptmerkmale und die Architektur von Karate benennen sowie den Installations- und Einrichtungsprozess in einem Maven-Projekt erklären.
LO-4.3	K3	Die Teilnehmer verstehen das Konzept des datengetriebenen Testens, können einen solchen Test unter Verwendung verschiedener Daten erstellen und die Vorteile gegenüber statischen Tests analysieren.
LO-4.4	K1	Die Teilnehmer verstehen die Rolle von Hooks im Testprozess.
LO-4.5	K2	Die Teilnehmer verstehen die Bedeutung des Mockings im Kontext von API-Tests und können Mocking-Funktionen in Karate implementieren
LO-4.6	K1	Die Teilnehmer verstehen häufige Herausforderungen bei API-Tests, verstehen Best Practices, Allgemeines Know How und Vor- und Nachteile.

4.1 API & Mock Grundlagen

4.1.1 Was sind APIs?

[MS1] [AW1]

API steht für Application Programming Interface. Das ist eine Art Schnittstelle, die es verschiedenen Softwareprogrammen ermöglicht, miteinander zu kommunizieren. APIs ermöglichen es sowohl Daten als auch Funktionen zwischen verschiedenen Systemen auszutauschen. Zum Beispiel kann eine Wetter-App eine API nutzen, um aktuelle Wetterdaten von einem Server abzufragen.

4.1.2 Funktionsweise von APIs

Zunächst wird eine Anfrage (Request) an einen Server gesendet. Dieser Request wird verarbeitet und anschließend eine Antwort (Response) zurückgeschickt. Diese Übertragung findet häufig über das HTTP Protokoll statt und ist sowohl synchron als auch asynchron möglich.

4.1.3 API Architektur

Eine API besteht aus Endpunkten, über die verfügbare Funktionen angesprochen werden können. Diese Endpunkte sind URLs und können aufgerufen werden. Auch Parameter sind bei solchen Aufrufen nutzbar. Der Datenaustausch erfolgt über das JSON oder XML Format. Zudem bestehen häufig Authentifizierungsmechanismen und Berechtigungen. Die Datenübertragung kann über unterschiedliche HTTP Methoden erfolgen, diese werden in Abschnitt 4.1.5, behandelt.

4.1.4 Übertragungsstandards

4.1.4.1 REST

REST Kommunikation erfolgt über das HTTP Protokoll und unterstützt sowohl JSON als auch XML Formate. Diese Art Schnittstellen sind sehr flexibel und haben kaum Vorgaben. Ihre Übertragung erfolgt direkt.

4.1.4.1 SOAP

SOAP Schnittstellen sind sehr streng angelegt und halten sich an Standards, was sie vergleichsweise starr macht. Die Übertragung erfolgt mit einer Validierung gegenüber einer Schema Datei oder auch WSDL genannt. SOAP verwendet für den Austausch nur das XML Format.

4.1.5 HTTP Methoden

Es bestehen zahlreiche HTTP Methoden, für diesen Lehrplan wird der Fokus lediglich auf die vier wichtigsten gelegt: GET, POST, PUT und DELETE.

4.1.5.1 GET

GET Requests werden verwendet, um Daten von einem Server abzurufen. Sie lassen keine Datenmanipulation zu.

4.1.5.2 POST

Diese Art von Request sendet Informationen oder Befehle an einen Server. Dadurch können Aktionen ausgelöst oder eine Verarbeitung angesteuert werden.

4.1.5.3 PUT

PUT Requests ermöglichen das Verändern von Daten, vergleichbar mit dem Update einer Datenbank. Datenbestände können aktualisiert oder ersetzt werden.

4.1.5.4 DELETE

Mit DELETE Requests können Daten vom Server dauerhaft entfernt werden.

4.1.6 API validieren

Die Überprüfung der API-Antworten ist essenziell, um die zurückgegebenen Daten auf erwartete Standards und Formate abzugleichen. Im Folgenden sind einige Methoden aufgeführt, mit denen sich API-Antworten validieren lassen:

4.1.6.1 Schema validation

Verwendet JSON-Schema oder XML-Schema, um die Struktur der Antwort zu definieren. Diese Antworten werden mit z. B. vorhandenen XSDs (bei XML) verglichen.

4.1.6.2 Type checking

Prüft ob die die Datentypen der Felder in der API-Antwort den erwarteten Typen entsprechen (z. B. String, Integer, Boolean).

4.1.6.3 Value range check

Sicherstellung, ob Wertebereiche eingehalten wurden. Sind vorgegebene numerische Werte vorhanden, liegen Strings in vordefinierten Minimal- und Maximallängen.

4.1.6.4 Presence of required fields

Überprüft, ob alle erforderlichen Pflichtfelder in der Antwort vorhanden sind

4.1.6.5 Inhaltsvalidierung

Validiert den Inhalt bestimmter Felder z.B. ob eine E-Mail-Adresse im richtigen Format vorliegt oder ob ein Statuswert einen bestimmten Satz von zulässigen Werten enthält.

4.1.7 Mocking

Mocks sind Nachbildungen oder Platzhalter für echte APIs. Sie werden in der Entwicklung verwendet, um die Funktionalität zu testen, ohne auf die tatsächliche API zugreifen zu müssen. Das ist besonders nützlich, wenn die echte API noch nicht fertig ist oder ein Echtzugriff dieser API mit hohen Kosten verbunden ist. Somit können Tests auch isoliert durchgeführt werden, ohne andere Teile des Systems zu berühren. Mit Mocks ist ein vordefiniertes Ergebnis verlässlich erzeugbar, unabhängig von äußeren Einflüssen.

4.2 Karate

[KA1]

Karate ist ein BDD-Framework, das speziell für API-Tests und Webservice-Tests entwickelt wurde. Es verwendet die bereits beschriebene Gherkin-Syntax, um Tests in einer lesbaren, Form zu schreiben. HTTP-Anfragen werden direkt unterstützt und ermöglichen somit die leichte Testfallentwicklung von APIs. Karate bietet sowohl Funktionen zur Validierung von JSON als auch XML Antworten direkt, was eine leichte Datenverarbeitung ermöglicht. Durch das Setzen einer Konfigurationsvariable ist auch die parallele Testausführung sehr leicht, was zudem die Testdurchführungsgeschwindigkeit erhöht. Ebenfalls besitzt Karate integrierte Reporting-Funktionen zur Analyse der Testergebnisse. Für diese API- Tests ist kein Glue Code notwendig, diese funktionieren bereits out of the box über die Gherkin Syntax.

Auch weitere 3rd Party Systeme wie z. B. Playwright, Spring oder Xray lassen sich problemlos integrieren.

4.2.1 Karate Architektur

Karate selbst baut auf den Konzepten von Cucumber auf, benutzt aber seine völlig eigene Architektur. Wesentlich sind die Teile für Web Browser Automation, API Test Doubles und API Testing. Alle drei bieten selbst zahlreiche Schnittstellen und Adapter, um Tools wie Playwright oder Selenium auszuführen und über eine Runtime Umgebung zu steuern. Obwohl Karate hauptsächlich für API Tests konzipiert ist, können so die unterschiedlichsten Werkzeuge integriert werden. API Test Doubles ermöglichen dabei die Mock-Server Integration. Diese Teile zählen zu der Core Component. Ein optionaler Teil ermöglicht die Integration von Desktop-Testautomatisierungstools.

4.2.2 Vergleich zwischen Karate und Cucumber

Im Gegensatz zu Cucumber sind in Karate bereits gängige Frameworks wie Selenium integriert und können ohne weiteres Zutun verwendet werden. Die Gherkin Syntax wird durch weitere Eigenschaften ergänzt. Dabei sind besonders die HTTP Request und Response Features von Bedeutung. Dennoch ist die Gherkin Syntax gleichartig. Karate bietet außerdem ein integriertes Reporting und erstellt nach jedem Testlauf einen Bericht mit entsprechenden Fehlerbeschreibungen. In diesem Bericht ist direkt ersichtlich bei welchem Step ein Fehler aufgetreten ist. Für Karate ist kein separater Glue Code notwendig.

4.2.3 Karate Runner und Konfiguration

Karate selbst lässt sich über Maven integrieren. Für die Ausführung von Karate Tests kann JUnit verwendet werden. Hierzu ist es jedoch notwendig, den Classpath des zugehörigen Feature-Files anzugeben, was über Runner.Path erfolgt. Diese Methode ermöglicht es auch, Tests parallel auszuführen und damit die Laufzeit zu verkürzen. Innerhalb des Runners stehen zudem weitere Konfigurationsmöglichkeiten zur Verfügung.

4.2.4 Karate & IDE Plugins

Sowohl für Visual Studio Code als auch IntelliJ bestehen Plugins, um eine Ausführung der Feature-Files direkt in der IDE zu ermöglichen. Diese Plugins haben sowohl kostenfreie als auch Bezahlfunktionen. Für Eclipse existiert hingegen keines, allerdings ist auch hier die Ausführung der Feature-Files mit dem Standard Cucumber Plugin möglich. Damit dies gelingt, ist jedoch zusätzlicher Aufwand notwendig: Es bedarf einer eigenen Main Class, in der die Karate Main aufgerufen wird. Diese muss sich im package „cucumber.api.cli“ befinden.

4.2.5 API Testing mit Karate

4.2.5.1 Matching Responses

- Karate bietet umfangreiche Funktionen zur Validierung von API-Antworten, insbesondere JSON- und XML-Daten. Mit der Gherkin-Syntax können erwartete Datenstrukturen direkt im Test definiert werden. Karate überprüft anschließend automatisch, ob die tatsächliche Antwort der erwarteten entspricht. Fehler bei der Validierung werden im integrierten Report deutlich angezeigt, inklusive des genauen Steps, in dem der Fehler aufgetreten ist. Eingeleitet wird dieses Matching durch das Schlüsselwort „match“ anschließend folgt welche Response z.B. response[0] und welches Element dem Erwartungswert entsprechen soll. Auch möglich ist das Abprüfen aller beinhalteten Elemente einer Response. Dies wird über das == und der anschließenden Inhaltsvorgabe durchgeführt.

4.2.5.2 Variablen

- In Karate können Variablen einfach innerhalb der Gherkin-Tests gesetzt und verwendet werden. Diese Variablen ermöglichen es, dynamische Werte (z. B. Tokens, IDs) für eine spätere Wiederverwendung zu speichern . Diese Werte lassen sich dann in späteren Requests oder Validierungen wiederverwenden. Variablen sind flexibel und sowohl lokal innerhalb eines Tests als auch global im Testlauf nutzbar. Eingeleitet wird eine Variable mit dem * def Variablenname. Anschließend folgt ihr Inhalt. Zur Verwendung bedarf es nachfolgender Syntax: '#(Variablenname)'. Auch ganze JSON Objekte lassen sich in mehrzeilige Variablen speichern, hierfür werden “““ zur Öffnung und ebenso “““ zum Schließen benötigt.

4.2.5.3 POST Methoden

- Karate unterstützt die direkte Ausführung von HTTP POST-Anfragen durch spezielle Keywords in der Gherkin-Syntax. Der Request-Body (z.B. JSON) ist definierbar und lässt sich direkt an die API senden. Karate validiert automatisch die Response und verarbeitet sie anschließend. Für POST Request reicht es in Gherkin method post anzugeben.

4.2.5.4 PUT Methoden

- Ähnlich wie bei POST-Requests ermöglicht Karate das Senden von PUT-Anfragen mit einem definierten Body (z.B. JSON). Die Response kann anschließend geprüft werden, um die Aktualisierung sicherzustellen. Ähnlich wie schon bei POST ist die Übertragungsmethode mit method get hier anzugeben.

4.2.5.5 DELETE Methode,

- Karate unterstützt auch DELETE-Anfragen, um Ressourcen zu entfernen. Diese Requests sind ohne Body ausführbar. Optional sind weitere Parameter mit übertragbar. Die Response wird durch Karate auf Erfolg bzw. Fehler geprüft und durch die integrierte Validierung ist sichergestellt, dass die Löschoperationen erfolgreich waren. Im Gherkin Szenario ist hier der zu löschende Eintrag festzulegen und die method auf delete zu setzen.

4.2.5.6 Tags

- In Karate können Tests mit Tags versehen werden, um sie gezielt zu gruppieren oder nur bestimmte Tests auszuführen. Dies ist vergleichbar mit den Tags von TestNG oder JUnit. Diese Tags werden direkt in den Gherkin-Feature-Dateien über @TagName vor den Szenarien oder Features definiert. Dies ermöglicht eine flexible Steuerung der Testläufe, z. B. um nur Regressionstests, Smoke-Tests oder spezielle Szenarien auszuführen. Es ist auch möglich, mehrere Tags an ein Szenario zu schreiben. Somit ist eine einfache und effiziente Testverwaltung gegeben.

4.2.5.7 Externe Szenarien Laden

- Auch das Laden von externen Szenarien oder Feature-Dateien ist in Karate möglich. Dies ermöglicht die einfache Wiederverwendung von Testfällen und verbessert die Organisation. Mit dem Befehl read('Dateiname') werden diese in ein Projekt integriert. Bei Bedarf erfolgt so die Einbindung einzelner oder mehreren externer Szenarien in einen Testlauf.

4.3 Datengetriebenes Testen mit Karate

Karate unterstützt datengetriebenes Testen. Diese Tests sind mit verschiedenen Eingabedaten durchführbar. Dabei werden mehrere Datensätze in einer Tabelle oder in externen Dateien (z. B. CSV, JSON) definiert. Vergleichbar wie schon zuvor bei den Cucumber Datentabellen laufen diese automatisch und iterativ nacheinander durch.

In Karate ist ebenfalls die bekannte Struktur des Scenario Outlines verwendbar. Alternativ lassen sich auch externe Datenquellen einbinden. Dadurch ist eine dynamische Einbindung unterschiedlicher Parameter in API-Requests möglich. Dies erhöht die Testabdeckung und Wiederverwendbarkeit erheblich, da verschiedene Szenarien mit minimalem Aufwand getestet werden können.

4.4 Verwendung von Hooks

Hooks in Karate sind spezielle Funktionen oder Anweisungen, die vor oder nach bestimmten Testphasen ausgeführt werden. Vergleichbar wie bei Cucumber können wiederkehrende Aufgaben automatisiert oder der Testablauf gesteuert werden.

Hooks in Karate:

- Karate.callSingle('classpath:.../my.feature'; wird in der karate.config.js hinterlegt um ein Szenario einmalig vor einem Szenario auszuführen
- Background um Vorbereitungen für alle Szenarien vor dem ersten Step einmalig durchzuführen
- configure um Schritte einmalig nach einem Szenario auszuführen

Karate bietet keine expliziten @Before oder @After-Annotationen wie Cucumber, sondern setzt auf diese Strukturen.

4.5 Integration von Mocking-Funktionen

4.5.1 Mocks

Mocks sind simulierte Objekte, die im Softwaretest verwendet werden, um Abhängigkeiten zu isolieren. Sie imitieren das Verhalten realer Komponenten und ermöglichen kontrollierte Tests. Im Allgemeinen werden unterschiedliche Mockarten unterschieden. Beispielsweise beinhalten sogenannte Fakes nur die Basisimplementierung eines Objekts mit stark eingegrenzter Funktionalität. Für diesen Syllabus wird der Begriff Mock im allgemeinen für ein simuliertes Objekt verwendet, um die notwendige Testfunktionalität zu gewährleisten.

4.5.2 Vorteile von Mocks

Unabhängigkeit von anderen Teilen:

Mocks isolieren die zu testende Komponente, sodass Tests nicht durch andere Systemteile beeinflusst werden.

Erlaubt frühes Testen:

Mit Mocks können Komponenten bereits in frühen Entwicklungsphasen getestet werden, bevor alle benötigten Komponenten für einen Test fertiggestellt sind. Unfertige Teile werden durch Mocks ersetzt.

Provozieren seltener Situationen:

Mocks ermöglichen das Simulieren ungewöhnlicher oder schwer erreichbarer Szenarien, die im echten System selten auftreten.

Spart Ressourcen:

Durch den Einsatz von Mocks entfallen aufwendige Setups und der Bedarf an echten Systemen, was Zeit und Kosten reduziert.

4.5.3 Karate und Mocking

Karate unterstützt ebenfalls das Mocken von API Endpunkten. Hierfür liefert Karate einen eigenen Mockserver mit, der in Java über ein Lifecyclemanagement verwaltet werden kann.
Durch bestehende Funktionalität sind unterschiedliche Request und Responses vor definierbar. Somit können Tests gegen diesen aufgesetzten Mockserver stattfinden. Auch das Festlegen von Requestparametern ist durchführbar, was die API-Tests sehr flexibel werden lässt.

Innerhalb eines Tests muss der erstellte Mockserver aufgerufen werden. Anschließend lässt sich über die Builder Class der Pfad für das Feature-file übergeben und der Test ausführen. Mit der entsprechenden Assertion ist der Testfall schließlich prüfbar. Es eignet sich hier auf FailCount zu testen, um fehlgeschlagene Requests auszuschließen. Mit getFailCount lässt sich überprüfen, ob grundsätzlich Fehler aufgetreten sind.

4.6 Best practices, Allgemeine Optimierungen und Vor- und Nachteile

4.6.1 Best Practices

Um Mocks effektiv im Softwaretest einzusetzen, sollten sie nur bei Bedarf verwendet werden, um unnötige Komplexität zu vermeiden. Dabei ist es wichtig, die Mocks nicht zu komplex zu gestalten, damit sie übersichtlich und wartbar bleiben. Zudem sollten Mocks sich nicht gegenseitig beeinflussen, um die Zuverlässigkeit und Unabhängigkeit von Tests zu behalten. Regelmäßiges Refactoring der Mocks trägt dazu bei, ihre Aktualität und Klarheit zu bewahren. Schließlich ist es

ratsam, auch das echte Objekt zu testen, um sicherzustellen, dass die Integration reibungslos funktioniert und keine unerwarteten Probleme auftreten. Unvorhergesehene Dinge können am echten Objekt trotz sorgfältigem Mocktesting auftreten.

4.6.2 Mocks & Testgeschwindigkeit

Mocks tragen wesentlich zur Steigerung der Testgeschwindigkeit bei, da sie in der Regel schneller sind als echte Systeme. Sie benötigen weniger Setup-Zeit, was die Vorbereitung der Tests vereinfacht und beschleunigt. Die eigentliche Testausführung mit Mocks ist deutlich kürzer, da keine komplexen Prozesse oder externe Ressourcen eingebunden werden müssen. Dennoch ist es wichtig, auch Tests mit echten Objekten durchzuführen, um die tatsächliche Integration und das Zusammenspiel im System zu überprüfen. Eine ausgewogene Kombination aus Mock-Tests und echten Systemtests sorgt für effiziente Entwicklungszyklen und zuverlässige Ergebnisse.

4.6.3 Mock Optimierungen

Um Mocks effizient zu nutzen, sollten sie - soweit möglich - wiederverwendbar gestaltet sein. Dies vermeidet Redundanz und erhöht die Wartbarkeit. Profiling und Monitoring der Mocks helfen dabei, ihre Leistung zu überwachen und potenzielle Engpässe frühzeitig zu erkennen. Es ist wichtig, die Nutzung der Mocks regelmäßig zu überprüfen, um sicherzustellen, dass sie stets optimal funktionieren. Zur Schonung von Ressourcen sollten Mocks nur dann geladen werden, wenn ein tatsächlicher Bedarf besteht. Dies verkürzt im allgemeinen auch die Testlaufzeiten. Diese Optimierungen tragen dazu bei, die Testumgebung stabiler und effizienter zu gestalten.

4.6.4 Vor- und Nachteile

- Vorteile von Mocks:

Der Einsatz von Mocks ermöglicht sehr frühes Testen, noch bevor alle Komponenten vollständig entwickelt sind. Dadurch können Fehler und Missverständnisse frühzeitig erkannt und behoben werden. Mocks bieten eine hohe Testgeschwindigkeit, da sie schnelle und einfache Simulationen realer Komponenten ermöglichen. Sie erlauben, dass isolierte Testen einzelner Einheiten, was die Fehlersuche erleichtert. Zudem sind Edgecases leichter testbar. Spezielle Szenarien sind gezielt simulierbar. Insgesamt vereinfachen Mocks das Testen komplexer Systeme und fördern eine effiziente Entwicklung.

- Nachteile von Mocks:

Allerdings können Mock-Objekte selbst sehr komplex werden, insbesondere bei umfangreichen oder dynamischen APIs. Dies führt zu einem hohen Wartungsaufwand, da Änderungen an der API auch die Mock-Implementierungen betreffen und regelmäßig angepasst werden müssen. Zudem besteht die Gefahr, dass potenzielle Fehler aufgrund der Abstraktion im Code unentdeckt bleiben. Tests mit Mocks bilden nicht immer alle realen Interaktionen korrekt ab. Somit können ein falsches Sicherheitsgefühl oder unerwartete Probleme im echten System entstehen.

Kapitel 5 – CI/CD mit Cucumber

Schlüsselbegriffe

Continuous Integration (CI), Continuous Deployment (CD)

LO-5.1	K1	Ein solides Verständnis der Grundlagen ist entscheidend, um die Bedeutung und den Nutzen von CI/CD in der Softwareentwicklung zu erkennen
LO-2.2	K3	Die Teilnehmer können eine einfache CI/CD-Pipeline mit Jenkins einrichten. Praktische Fähigkeiten zur Einrichtung einer Pipeline sind notwendig, um das theoretische Wissen in die Praxis umzusetzen.
LO-5.3	K2	Verständnis über den BDD-Workflow und dessen Integration in Jenkins ist vorhanden
LO-5.4	K2	Teilnehmende können Testberichte aus Cucumber-Tests generieren und interpretieren. Testberichte sind wichtig für das Feedback an das Team und helfen bei der Identifizierung von Problemen im Entwicklungsprozess.
LO-5.5	K1	Verständnis über die Bedeutung einer strukturierten Testorganisation innerhalb eines CI/CD-Prozesses besteht. Ein klares Verständnis der Struktur hilft dabei, Effizienz zu steigern und die Wartbarkeit der Tests zu gewährleisten.

5.1 Continuous Integration / Continuous Deployment

5.1.1 Continuous Integration (CI)

Continuous Integration (CI) ist eine bewährte Praxis in der Softwareentwicklung. Hierbei werden Codeänderungen regelmäßig - oft mehrmals täglich - in ein gemeinsames Repository integrieren. Ziel ist es, die Qualität der Software zu verbessern und den Entwicklungsprozess effizienter zu gestalten.

Integration von neuen Komponenten

Bei CI werden neue Komponenten oder Funktionen kontinuierlich in das bestehende System integriert. Entwickler führen somit ihre Änderungen frühzeitig und häufig zusammen. Konflikte und Integrationsprobleme lassen sich dadurch frühzeitig erkennen. Dies erleichtert das Hinzufügen neuer Komponenten erheblich, da sie schrittweise getestet und integriert werden können.

Steigerung der Softwarequalität durch Prüfung

Ein zentraler Vorteil von CI ist die automatische Prüfung des Codes bei jedem Commit. Nach dem Code-Commit durchläuft die Änderung mehrere Stages: Build, Unit Tests und Integration Tests. Diese automatisierten Tests stellen sicher, dass neue Änderungen keine bestehenden Funktionen beeinträchtigen und die Software stabil bleibt.

Vereinfachung des Hinzufügens neuer Komponenten

Durch die Automatisierung der Test- und Integrationsprozesse wird das Hinzufügen neuer Komponenten vereinfacht. Entwickler können schnell Feedback erhalten, ob ihre Änderungen funktionieren und keine Fehler verursachen. Dies beschleunigt den Entwicklungszyklus und fördert eine agile Arbeitsweise.

5.1.2 Continuous Deployment (CD)

Continuous Deployment (CD) ist ein Prozess in der Softwareentwicklung, bei dem die automatisierte Auslieferung von Software-Änderungen direkt in die Produktionsumgebung erfolgt. Nach einem erfolgreichen Continuous Integration (CI)-Prozess werden die geprüften Komponenten automatisch

in die Produktion überführt, sodass neue Funktionen und Updates schnell und zuverlässig für die Anwender verfügbar sind.

Software-Auslieferungsprozess nach CI

Der CD-Prozess baut auf dem CI auf und umfasst im allgemeinen mehrere Schritte, um sicherzustellen, dass nur qualitativ hochwertige Software in die Produktion gelangt:

Review:

Bevor eine Version in die Staging-Umgebung deployt wird, erfolgt eine Überprüfung des Codes durch Entwickler oder automatisierte Systeme. Hierbei werden etwa Code-Qualität, Sicherheitsaspekte und Funktionalität geprüft.

Staging:

Nach erfolgreichem Review wird die Software in einer Staging-Umgebung bereitgestellt. Diese Umgebung simuliert die Produktionsumgebung möglichst nah. Hier finden letzte Tests statt, um sicherzustellen, dass alles reibungslos funktioniert und keine unerwarteten Probleme auftreten.

Production:

Nach erfolgreichem Abschluss aller Tests und Reviews wird die Software automatisch oder manuell in die Produktionsumgebung ausgerollt. Die Nutzer profitieren somit schnell von neuen Features und Verbesserungen.

Die genannten Prozesse in CI und CD sind lediglich die Basis und können durch weitere Prozesse wie z. B. Reporting erweitert werden.

5.2 Jenkins

[JE1]

Der Jenkins Buildserver ist eine Open-Source-Automatisierungsplattform, die vor allem im Bereich Continuous Integration (CI) und Continuous Delivery (CD) eingesetzt wird. Er ermöglicht es, Softwareprojekte automatisch zu bauen, zu testen und bereitzustellen. Somit lässt sich der Entwicklungsprozess effizienter gestaltet. Jenkins unterstützt umfangreiche Reporting-Funktionen, um den Status und die Qualität der Builds transparent darzustellen. Die Konfiguration erfolgt flexibel über sogenannte Jenkinsfiles. In diesen Files sind Pipelines und Automatisierungsprozesse versioniert und nachvollziehbar definiert. Damit ist Jenkins eine leistungsstarke Lösung für automatisierte Softwareentwicklung, CI und CD.

5.3 BDD Workflow mit Jenkins

In der bereitgestellten Jenkins Oberfläche kann ein Buildprozess über den Reiter „New Item“ angelegt werden. Mit dem Feld „Build now“ ist es möglich, einen bestehenden Buildprozess anzustoßen. Zudem zeigt die Queue, welche Jobs gerade aktiv sind. Innerhalb der Jenkins Oberfläche können auch bestehende Reports angezeigt werden.

Jenkins bietet die Möglichkeit, Maven Projekte einzubinden und anschließend auszuführen. Mit der Startbedingung innerhalb der Buildkonfiguration lässt sich einstellen, wann ein Build beginnt und welche Optionen dabei berücksichtig werden. Sowohl die Maven POM als auch die entsprechenden Maven Goals sind konfigurierbar.

In den Post-Build-Actions ist es möglich, die Vorgaben für Reportgenerierung wie z. B. eines Cucumber Reports festzulegen.

Im allgemeinen ruft Jenkins den JUnit oder TestNG Runner auf, welcher wiederum die Featuresfiles ausführt. Nach dieser Ausführung erfolgt das Erstellen der Testreports. Da diese Berichte für gewöhnlich als Json und HTML abgelegt sind, lassen sie sich auch komfortabel an weitere Systeme oder Personen übertragen.

5.4 Berichterstattung und Testberichte generierens

Für die Testberichterstellung wird eine Report Json benötigt und muss durch den entsprechenden Runner eingestellt sein. Geg. Ist auch die Anpassung der Runnerclass notwendig. In einigen Fällen kann es ebenfalls erforderlich sein, die Maven Pom anzupassen um den entsprechenden Erstellprozess auszulösen. Mit dieser Json können dann Berichte in HTML oder anderen Formaten erstellt werden.

5.5 Best Practices Best Practices für die Organisation von Tests

- Benachrichtigungen:

Kritische Fehlern sollten zu einer automatischen Benachrichtigung führen. Beispielsweise der Versand per E-Mail oder Slack eignet sich hier besonders und kann automatisch in die Buildpipeline integriert werden. So können Teams schnell auf Probleme reagieren und die Qualität der Software hochhalten.

- Fehlerbehandlung**:**

Es ist ratsam, eine sinnvolle Protokollierung innerhalb der Pipelines zu implementieren. Detaillierte Logs helfen dabei, Fehlerursachen schnell zu identifizieren und die Ursachenanalyse effizient durchzuführen.

- Vermeidung von großen gemeinsam genutzten Bibliotheken:

Es empfiehlt sich, umfangreiche gemeinsam genutzte Bibliotheken oder Abhängigkeiten zu minimieren. Stattdessen sollten Komponenten modular gehalten und lediglich die erforderlichen Teile in den jeweiligen Pipelines verwendet werden. Das steigert die Flexibilität, reduziert Komplexität und minimiert potenzielle Konflikte.

References

Spezifische Referenzen

Referenzen	Quelle
[BT1]	A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives - By L. Anderson, P. W. Airasian, and D. R. Krathwohl (Allyn & Bacon 2001)
[BT2]	https://www.apu.edu/live_data/files/333/blooms_taxonomy_action_verbs.pdf
[MS1]	Softwaretesting kompakt: Grundlagen von Tests und Testautomatisierung mit Java (IT kompakt) – By Pascal Moll and Daniel Sonnet (Springer Vieweg Wiesbaden, 2025)
[CC1]	https://cucumber.io/docs
[KA1]	https://www.karatelabs.io/
[JE1]	https://www.jenkins.io/
[AW1]	https://aws.amazon.com/what-is/api/
[TE1]	https://testng.org/