Java aktuell

iJUG Verbund
www.ijug.eu

Java 25
18 neue JEPs im nächsten Release

Von grünen Systemen
Nachhaltigkeit systematisch denken

Nichts zu verbergen?
Überwachung, Privatsphäre und digitale Selbstbestimmung

Ausgabe 04/2025
Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwicklerinnen und Entwickler
D: 7,95 EUR | CH: 11,99 CHF | A: 9,00 EUR | Benelux: 8,50 EUR
4 191978 307959 04

Shift-Left mit System: Fachliche Anforderungen dokumentieren und testen

Felix Tensing, Nürnberger Versicherung

Frühe Tests versprechen bessere Qualität, schnellere Rückmeldung und dadurch geringere Kosten – das Prinzip „Shift-Left" ist aus modernen Entwicklungsprozessen kaum wegzudenken. Doch in der Praxis scheitert es oft an einem klaren Verständnis davon, was genau getestet werden soll. Dieser Artikel richtet sich an Entwickler:innen, Tester:innen, Product Owner – und alle, die ihre Anforderungen nicht nur besser dokumentieren, sondern auch automatisiert testen und langfristig nachvollziehen wollen. Er zeigt, wie sich fachliche Anforderungen so dokumentieren lassen, dass sie nicht nur verständlich, sondern auch testbar und versionierbar sind – und so eine nachhaltige Grundlage für Testautomatisierung und Zusammenarbeit schaffen.

„Shift-Left" beschreibt das Prinzip, Qualitätssicherung frühzeitig in den Entwicklungsprozess zu integrieren. Früh entdeckte Fehler lassen sich meist deutlich günstiger beheben. Werden sie dagegen erst spät gefunden – etwa in der Produktion – steigen Aufwand und Kosten erheblich. Dabei können Fehler nicht nur Bugs sein, sondern auch fachliche Ungereimtheiten oder eine falsche Implementierung aufgrund missverständlicher Anforderungsdefinition.

Doch Shift-Left bedeutet nicht nur frühzeitiges Testen, sondern auch eine erweiterte Verantwortung für die Entwicklungsteams. Sie müssen bereits während der Implementierung sicherstellen, dass Anforderungen korrekt umgesetzt und potenzielle Probleme rechtzeitig erkannt werden. Das setzt eine enge Verzahnung von Anforderungen, Dokumentation und Tests voraus. Im Artikel werden diese Zusammenhänge näher erläutert und ein Ansatz vorgestellt, der den Einsatz von Shift-Left fördert und gleichzeitig die Aufwände im Team gering hält.

Dokumentation ist nicht gleich Dokumentation

In der IT existieren viele unterschiedliche Arten von Dokumentationen – mit unterschiedlichen Zielgruppen, Formaten und Zwecken. Um Missverständnisse zu vermeiden, lohnt es sich, diese Begriffe kurz zu differenzieren. Die folgende Einordnung ist nicht vollständig, sondern konzentriert sich auf jene Formen von Dokumentation, die für das hier vorgestellte Vorgehen relevant sind – oder häufig zu Verwirrung führen.

Dokumentation im Quellcode umfasst sowohl JavaDoc als auch Kommentare. Diese dienen Entwickler:innen dazu, technische Entscheidungen zu erläutern und den Code verständlich zu machen. JavaDoc dokumentiert dabei insbesondere die öffentliche API von Klassen, Methoden und Schnittstellen – also wie Komponenten genutzt werden können. Solche Dokumentation ist eng an die Implementierung gebunden und ändert sich meist mit ihr. Für das fachliche Verständnis des Gesamtsystems ist sie jedoch nur bedingt geeignet.

Technische Systemdokumentation umfasst Artefakte wie Architekturmodelle, Umsystembeschreibungen, technische Vorgaben oder Architekturentscheidungsprotokolle (Architecture Decision Records, ADRs). Ein bekanntes Beispiel ist das Arc42-Template, das als strukturierter Leitfaden zur Dokumentation von Softwarearchitektur weit verbreitet ist. Diese Dokumentation liefert wichtige technische Orientierung für Entwicklung und Betrieb. Ihr Fokus liegt jedoch auf technischen Zusammenhängen und Infrastruktur, nicht auf der fachlichen Logik oder den Geschäftsregeln eines Systems.

Fachliche Dokumentation beschreibt, was ein System aus Sicht der Anwender:innen oder Fachabteilungen leistet – unabhängig von der technischen Umsetzung. Sie bildet die Grundlage für ein gemeinsames Verständnis der Geschäftsregeln, Prozesse und erwarteten Funktionen. Fachliche Dokumentation ist damit die Basis für die Testbarkeit der Anforderungen.

Wird in diesem Artikel von Dokumentation gesprochen, ist damit die fachliche Dokumentation gemeint, sofern nicht explizit eine andere Form von Dokumentation genannt wird.

Anforderung ist nicht gleich Feature

Anforderungen beschreiben, was ein System aus fachlicher Sicht leisten soll. Sie formulieren ein Ziel oder Bedürfnis, unabhängig von der konkreten technischen Umsetzung. Features hingegen sind konkrete Funktionseinheiten, mit denen Anforderungen umgesetzt werden – oder bereits umgesetzt wurden.

In der Praxis wird der Begriff „Feature" oft schon während der Planung oder Entwicklung verwendet, obwohl die Funktion noch nicht umgesetzt ist. Streng genommen handelt es sich dabei um einen *Feature-Wunsch* – also eine geplante Erweiterung. Im Alltag wird dieser Begriff jedoch häufig verkürzt und ebenfalls als „Feature" bezeichnet. Diese begriffliche Unschärfe ist ein typisches Beispiel für kommunikative Schwächen in Softwareprojekten – und ein häufiger Auslöser für Missverständnisse.

Die Beziehung zwischen Anforderungen und Features ist häufig nicht eindeutig: Ein Feature kann mehrere Anforderungen gleichzeitig erfüllen und eine einzelne Anforderung kann sich auf verschiedene Features verteilen. So führt etwa die Anforderung „Benutzer:innen können sich anmelden" zu Features wie „Login mit Benutzername und Passwort" sowie „Anmeldung über Single Sign-On (SSO)". Gleichzeitig erfüllt das Feature „SSO" auch technische oder sicherheitsbezogene Anforderungen, etwa zur zentralen Authentifizierung.

Was passiert eigentlich mit Anforderungen nach der Umsetzung?

Anforderungen bilden in vielen Projekten den Ausgangspunkt für die Umsetzung: Sie beschreiben gewünschte Funktionen, Ziele oder Rahmenbedingungen. In frühen Projektphasen dienen sie als Grundlage für Planung, Aufwandsschätzung und technische Konzeption.

Doch sobald die Umsetzung erfolgt ist und ein Feature entstanden ist, verliert die ursprüngliche Anforderung oft an Sichtbarkeit – oder verschwindet ganz aus dem Fokus.

In der Praxis werden Anforderungen selten versioniert oder aktualisiert. Bei Weiterentwicklungen kommen neue Anforderungen hinzu, die bestehende Anforderungen ändern oder erweitern – ohne dass die ursprünglichen Anforderungen systematisch angepasst wurden. So entsteht ein wachsender Strom an Änderungswünschen, während die alte Anforderungslage in den Hintergrund tritt oder schlicht veraltet.

Das führt dazu, dass Anforderungen in vielen Projekten nicht weiter gepflegt werden – und deshalb als dauerhafte Referenz oder Dokumentationsbasis nicht geeignet sind.

Im Sprachgebrauch ist mit „der Anforderung" oft nicht ein einzelner Wunsch gemeint, sondern ein ganzes Themenfeld – etwa „Login" oder „Zahlungsabwicklung". Dabei bleibt meist unklar, welche konkreten Funktionen, Regeln oder Ausnahmen damit gemeint sind. Diese Unschärfe erschwert die Aussage, ob eine Anforderung vollständig umgesetzt oder getestet wurde – und ist damit ein zentrales Problem bei der Anforderungsabdeckung (siehe Kasten: Anforderungsabdeckung erklärt).

Anforderungsabdeckung erklärt

Anforderungsabdeckung (Requirements Coverage) bezeichnet das Ausmaß, in dem definierte Anforderungen durch Tests überprüft werden. Ziel ist es, sicherzustellen, dass alle spezifizierten Anforderungen im Testprozess berücksichtigt werden. Eine klassische Definition findet sich bei Kaner, Falk & Nguyen (1999):
„Requirements coverage is a measure of the extent to which the requirements are exercised by a test suite." *[1]*

In der Praxis zeigt sich jedoch, dass das reine Messen der Anforderungsabdeckung oft wenig Aussagekraft besitzt, da Anforderungen häufig unscharf formuliert sind oder sich im Verlauf der Softwareentwicklung verändern. Stattdessen ist die Featureabdeckung – also die Abdeckung der konkreten, implementierten Funktionalitäten – oft ein sinnvollerer Maßstab für Qualitätssicherung.

Allerdings ist der Begriff „Featureabdeckung" im üblichen Sprachgebrauch kaum etabliert, sodass in vielen Teams weiterhin von „Anforderungsabdeckung" gesprochen wird, obwohl eigentlich die Featureabdeckung gemeint ist.

Auch in diesem Artikel wird der Begriff Anforderungsabdeckung verwendet, wenn im Grunde die Abdeckung der Features gemeint ist.

Die Fachfunktion

Wie bereits gezeigt, sind Anforderungen vor allem ein Arbeitswerkzeug zur Umsetzung und nicht als Grundlage für dauerhafte Tests geeignet.

Features – also die tatsächlich umgesetzten Funktionalitäten – sind die relevanten Objekte. Die Dokumentation der Features bildet die Basis für eine verlässliche und nachhaltige Anforderungsabdeckung. In diesem Artikel führe ich den Begriff Fachfunktion (siehe Kasten: Fachfunktion kurz und knapp) für eine strukturierte und nachvollziehbare Dokumentation der Features ein, die als verbindliche Grundlage für Tests und Qualitätssicherung dient. Dieser neue Begriff hilft, eine sprachliche Trennung zwischen Anforderungen und Features zu schaffen und macht deutlich, dass eine Fachfunktion ein Feature dokumentiert. In der Praxis hat sich gezeigt, dass diese Benennung dafür sorgt, dass allen Beteiligten klar ist, worum es geht, wenn von einer Fachfunktion die Rede ist.

Von der Anforderung zur Fachfunktion

Die Anforderung bildet die fachliche Grundlage, aus der in der Praxis meist mehrere User Stories oder Arbeitspakete abgeleitet werden. Während die Anforderung die Planungsebene darstellt, fokussieren sich Stories auf die konkrete Umsetzungsebene.

Entwickler:innen setzen diese Stories in Code um. Jede fertiggestellte Story führt dabei zu einer Änderung oder Ergänzung der Fachfunktionen. Da nur die Entwickler:innen genau wissen, welche Umsetzung tatsächlich erfolgt ist, dokumentieren sie in den Fachfunktionen den aktuellen Zustand der Software aus fachlicher Sicht. So entsteht eine Dokumentation, die stets den tatsächlichen Stand der Software widerspiegelt. Damit das funktioniert, ist es entscheidend, dass die Fachfunktion Teil des Codes ist – also nach dem Prinzip „Documentation as Code" (siehe Kasten: *Documentation as Code*) verwaltet wird. Nur so lässt sich sicherstellen, dass Dokumentation und Implementierung synchron bleiben.

Documentation as Code

„Documentation as Code" (oder kurz auch Docs-as-Code) bezeichnet einen Ansatz, bei dem Dokumentation nach denselben Prinzipien wie Quellcode behandelt wird. Sie liegt in textbasierter Form vor (zum Beispiel Markdown, AsciiDoc), wird versioniert in einem Versionskontrollsystem verwaltet, kann gemeinsam mit dem Code entwickelt und automatisiert geprüft, erzeugt oder verarbeitet werden.

Ziel ist es, Dokumentation zuverlässig, aktuell und integrativ in die Entwicklungsprozesse einzubinden – ohne Medienbrüche, manuelle Nachpflege oder separate Werkzeuge. Damit wird Dokumentation zum festen Bestandteil der Software – nachvollziehbar, überprüfbar und teamübergreifend nutzbar.

Da sie im gleichen Repositorium wie der Code liegt, kann die Dokumentation im Rahmen von Code-Reviews direkt mitgeprüft und versioniert nachvollzogen werden.

Der Ansatz eignet sich nicht nur für fachliche Dokumentation, sondern auch für Architektur-, Betriebs- oder Entwicklerdokumentation – überall dort, wo Nähe zum Code von Vorteil ist.
In vielen Teams und Organisationen gilt „Documentation as Code" heute als Best Practice.

Im Sinne des bereits erwähnten Shift-Left-Prinzips liegen fachliche Dokumentation und dementsprechend auch das Testen im Aufgabenbereich der Entwickler:innen. Damit das gut funktioniert, müssen die Aufwände dafür möglichst geringgehalten werden. Eine strukturierte Dokumentation hilft dabei, den Fokus auf die Inhalte zu legen, ohne durch unnötiges Layout oder administrative Aufgaben abzulenken.

Die Fachfunktion bietet einen solchen strukturierten Rahmen: Sie ermöglicht es, fachliche Funktionen klar und einheitlich zu beschreiben. Im Beispielprojekt (siehe Kasten: *Beispielprojekt*) des Autors wird YAML als Dateiformat genutzt. Denkbar sind aber auch andere Formate wie JSON oder auch XML. In *Listing 1* ist eine einfache Fachfunktion aufgeführt.

Beispielprojekt

Die hier gezeigten Codebeispiele sind frei verfügbar auf GitHub *[7]*.

Die Implementierung ist bewusst minimal gehalten, um die Verständlichkeit zu fördern. Weiterentwicklungen wie die Nutzung von Annotationen zur Testmarkierung wurden absichtlich nicht umgesetzt, um das Grundprinzip klar zu vermitteln.

Das Projekt kann als Vorlage zur Umsetzung in anderen Programmiersprachen dienen, solange das zugrundeliegende Prinzip erhalten bleibt. So hat der Autor beispielsweise auch eine Variante mit TypeScript, Vitest und Storybook für ein Next.js-Projekt realisiert.

Feedback, Rückfragen und Vorschläge sind willkommen – ebenso wie Merge-Requests. Nutzen Sie gern die GitHub-Issue-Funktion, um mit dem Autor und der Community in Kontakt zu treten.

Im Folgenden werden die einzelnen Bestandteile kurz erläutert:

name
Der Name benennt die Fachfunktion eindeutig. Er sollte möglichst präzise und fachlich sprechend sein.

kurzbeschreibung
Die Kurzbeschreibung gibt einen kompakten Überblick über die fachliche Aufgabe der Funktion. Sie ermöglicht ein schnelles Verständnis des Kontexts, ohne Details vorwegzunehmen.

akzeptanzkriterien
Die Akzeptanzkriterien beschreiben konkret, wann die Fachfunktion als korrekt umgesetzt gilt. Sie bilden die Grundlage für Tests und müssen entsprechend testbar formuliert sein. Ziel im Sinne von Shift-Left ist es, dass alle Kriterien auch automatisiert überprüft werden können. Dazu später mehr. Jedes Kriterium wird mit einer ID versehen, um gezielt darauf referenzieren zu können.

tags
Tags helfen, Fachfunktionen thematisch zu gruppieren oder mit zusätzlichen Attributen zu versehen. Sie können für Filterung, Auswertung oder einfach zur besseren Orientierung genutzt werden.

Diese einfache Struktur erlaubt es, Fachfunktionen konsistent und nachprüfbar zu dokumentieren – direkt im Code, versionierbar und testbar. Eine Erweiterung der Struktur an projektspezifische Anforderungen ist problemlos möglich – etwa um Angaben zu Rollen oder Berechtigungen zu ergänzen.

Zusätzlich zur strukturierten Dokumentation benötigt jede Fachfunktion eine ausführlichere fachliche Beschreibung. Sie dient dazu, Zusammenhänge zu erläutern, Hintergründe zu dokumentieren oder auf externe Quellen zu verweisen.

Im Beispielprojekt wird hierfür das Format AsciiDoc *[2]* verwendet. Die Beschreibung wird dabei als eigene Datei abgelegt, um eine bessere Unterstützung in der IDE zu ermöglichen – etwa durch Syntax-Highlighting, Vorschau und Validierung.

AsciiDoc bietet eine einfache, aber wirkungsvolle Möglichkeit zur ansprechenden und gut lesbaren Gestaltung technischer Texte – direkt im Editor. Es unterstützt unter anderem Überschriften, Formatierungen, Quellcode-Auszüge mit Syntax-Highlighting und Tabellen. Darüber hinaus lassen sich mit PlantUML *[3]* auch Diagramme und Abläufe direkt einbetten – vollständig textbasiert, versionierbar und ohne externe Tools. So können auch komplexe fachliche Zusammenhänge gut verständlich dokumentiert werden.

Listing 2 zeigt ein Beispiel der Beschreibung in AsciiDoc.

```
---
name: Benutzer Login
kurzbeschreibung: Authentifizierung von Benutzer:innen anhand von Zugangsdaten.
akzeptanzkriterien:
 - 01: "Ein gültiger Benutzername und ein korrektes Passwort führen zur erfolgreichen Anmeldung."
 - 02: "Ein ungültiger Benutzername oder ein falsches Passwort führen zu einer Fehlermeldung."
 - 03: "Benutzer:innen ohne Aktivierung können sich nicht anmelden."
 - 04: "Die Anmeldung erzeugt ein gültiges JWT-Token mit Ablaufzeit."
tags:
 - Benutzername
 - Login
 - Passwort
```

Listing 1: FF-PROJ-0001.yaml, beispielhafte Fachfunktion zum Login

```
Der Login ist der Einstiegspunkt für alle geschützten Funktionen im System.
Benutzer:innen geben ihre Zugangsdaten ein und erhalten bei erfolgreicher Anmeldung ein Authentifizierungs-Token.


*Beispiel: Authentifizierungs-Token*
[source,json]
----
{
 "sub": "user123",
 "iat": 1616161616,
 "exp": 1616165216,
 "sessionId": "abc123xyz",
 "roles": ["user", "admin"]
}
----
```

Listing 2: FF-BSPP-0001.adoc, AsciiDoc Beispiel der Beschreibung einer Fachfunktion

Von der Dokumentation zur Testabdeckung

Im nächsten Schritt verbinden wir die Fachfunktion mit konkreten Tests – und schaffen damit die Grundlage zur Berechnung der Anforderungsabdeckung.

Dank der klaren Struktur verfügt jedes Akzeptanzkriterium über eine eindeutige ID. Wie gezeigt , setzt sich diese aus der Kennung der Fachfunktion und der Nummer des Akzeptanzkriteriums zusammen. Als Trenner wird ein Hashtag verwendet – etwa FF-BSPP-0001#02.

Automatisierte Tests können auf diese ID referenzieren. Im Beispielprojekt wird das Akzeptanzkriterium direkt im Namen des Tests mit aufgeführt *(siehe Listing 3)*. Diese Variante ist unkompliziert, gut verständlich und lässt sich ohne zusätzliche Implementierung umsetzen.

Natürlich kann ein Test auch mehrere Akzeptanzkriterien abdecken – insbesondere bei komplexeren Abläufen oder integrierten Testfällen. In diesem Fall lassen sich einfach mehrere IDs angeben.

Generell ist es wichtig, die Testnamen als aussagekräftige, gut verständliche Sätze zu formulieren. So lässt sich später aus der generierten Dokumentation unmittelbar erkennen, wie der Test auf die jeweiligen Akzeptanzkriterien einzahlt – ganz ohne den Testcode lesen zu müssen.

... und etwas Glue-Code

Mit der strukturierten Dokumentation der Fachfunktionen und der Verknüpfung der Tests mit den Akzeptanzkriterien sind die wichtigsten Grundlagen geschaffen, um eine Dokumentation mit Anforderungsabdeckung zu generieren.

Was für eine vollständige Lösung noch fehlt, sind ein Template zum Rendern, ein Skript zum Sammeln und Verarbeiten der vorhandenen Daten sowie einige Maven-Plugins zur Integration in den Build-Prozess.

Das Skript lässt sich vereinfacht wie folgt zusammenfassen:

- Scannen des Dokumentationsordners nach YAML-Dateien mit Fachfunktionen
- Ablegen der Informationen in einer internen Map
- Einlesen des Surefire-XML-Reports und Verknüpfen der Testergebnisse mit den Fachfunktionen in der Map
- Berechnung der prozentualen Anforderungsabdeckung: Ein Akzeptanzkriterium gilt als erfolgreich getestet, wenn mindestens

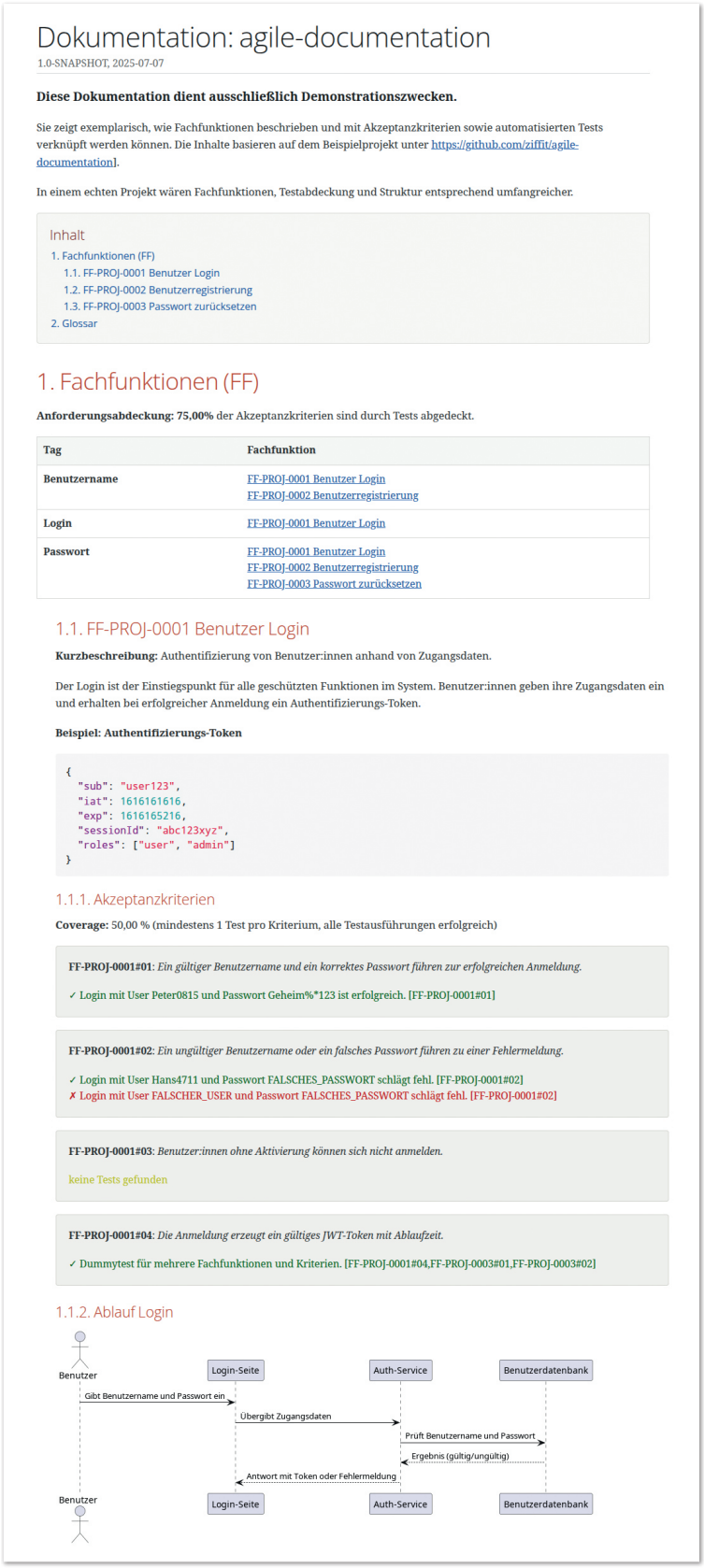
```
package de.fx.agiledocumentation

import spock.lang.Specification

class LoginTest extends Specification {

  def 'Login mit User Peter0815 und Passwort Geheim%*123 ist erfolgreich. [FF-PROJ-0001#01]'() {
        expect:
        true // Hier sollte die Logik für den Login-Test stehen, z.B. ein Aufruf einer Login-Methode
  }

  def 'Login mit User Hans4711 und Passwort FALSCHES_PASSWORT schlägt fehl. [FF-PROJ-0001#02]'() {
        expect:
        true // Hier sollte die Logik für den Login-Test stehen, z.B. ein Aufruf einer Login-Methode
  }

  def "Login mit User FALSCHER_USER und Passwort FALSCHES_PASSWORT schlägt fehl. [FF-PROJ-0001#02]"() {
        expect:
        false //Absichtlich fehlschlagender Test
  }

}
```

Listing 3: Referenzieren von Akzeptanzkriterien in Spock-Tests [4]

ein Test darauf referenziert und alle diese Tests erfolgreich ausgeführt wurden.
- Rendern einer AsciiDoc-Datei mithilfe eines Templates

Im Build-Prozess führt Maven vor dem Skript den Testrunner aus und nach dem Skript das AsciiDoc-Plugin, um aus der generierten AsciiDoc-Datei einen HTML-Report zu erzeugen. Dieser Report kann beispielsweise über ein weiteres Plugin nach Confluence exportiert werden. Alternativ ist auch die Umwandlung in ein PDF möglich, das als Artefakt im Artifactory abgelegt werden kann.

Das Ergebnis, der Report im HTML-Format, ist in *Abbildung 1* zu sehen.



Dokumentation: agile-documentation

1.0-SNAPSHOT, 2025-07-07

Diese Dokumentation dient ausschließlich Demonstrationszwecken.

Sie zeigt exemplarisch, wie Fachfunktionen beschrieben und mit Akzeptanzkriterien sowie automatisierten Tests verknüpft werden können. Die Inhalte basieren auf dem Beispielprojekt unter https://github.com/ziffit/agile-documentation].

In einem echten Projekt wären Fachfunktionen, Testabdeckung und Struktur entsprechend umfangreicher.

Inhalt

1. Fachfunktionen (FF)
   1.1. FF-PROJ-0001 Benutzer Login
   1.2. FF-PROJ-0002 Benutzerregistrierung
   1.3. FF-PROJ-0003 Passwort zurücksetzen
2. Glossar

1. Fachfunktionen (FF)

Anforderungsabdeckung: 75,00% der Akzeptanzkriterien sind durch Tests abgedeckt.

Tag	Fachfunktion
Benutzername	FF-PROJ-0001 Benutzer Login FF-PROJ-0002 Benutzerregistrierung
Login	FF-PROJ-0001 Benutzer Login
Passwort	FF-PROJ-0001 Benutzer Login FF-PROJ-0002 Benutzerregistrierung FF-PROJ-0003 Passwort zurücksetzen

1.1. FF-PROJ-0001 Benutzer Login

Kurzbeschreibung: Authentifizierung von Benutzer:innen anhand von Zugangsdaten.

Der Login ist der Einstiegspunkt für alle geschützten Funktionen im System. Benutzer:innen geben ihre Zugangsdaten ein und erhalten bei erfolgreicher Anmeldung ein Authentifizierungs-Token.

Beispiel: Authentifizierungs-Token

```
{
  "sub": "user123",
  "iat": 1616161616,
  "exp": 1616165216,
  "sessionId": "abc123xyz",
  "roles": ["user", "admin"]
}
```

1.1.1. Akzeptanzkriterien

Coverage: 50,00 % (mindestens 1 Test pro Kriterium, alle Testausführungen erfolgreich)

FF-PROJ-0001#01: *Ein gültiger Benutzername und ein korrektes Passwort führen zur erfolgreichen Anmeldung.*

✓ Login mit User Peter0815 und Passwort Geheim%*123 ist erfolgreich. [FF-PROJ-0001#01]

FF-PROJ-0001#02: *Ein ungültiger Benutzername oder ein falsches Passwort führen zu einer Fehlermeldung.*

✓ Login mit User Hans4711 und Passwort FALSCHES_PASSWORT schlägt fehl. [FF-PROJ-0001#02]
✗ Login mit User FALSCHER_USER und Passwort FALSCHES_PASSWORT schlägt fehl. [FF-PROJ-0001#02]

FF-PROJ-0001#03: *Benutzer:innen ohne Aktivierung können sich nicht anmelden.*

keine Tests gefunden

FF-PROJ-0001#04: *Die Anmeldung erzeugt ein gültiges JWT-Token mit Ablaufzeit.*

✓ Dummytest für mehrere Fachfunktionen und Kriterien. [FF-PROJ-0001#04,FF-PROJ-0003#01,FF-PROJ-0003#02]

1.1.2. Ablauf Login

Abbildung 1: Ausschnitt des erzeugten HTML-Reports (© Felix Tensing)

Fachfunktion (FF) kurz und knapp

Merksatz:
„Eine Fachfunktion beschreibt, was die Software leistet – nicht, wie sie bedient wird."

Eine Fachfunktion:
- Hat eine eindeutige Kennung, z. B. FF-PROJ-0001 (FF: Fachfunktion, PROJ: Projekt- oder Artefakt-Kürzel, z. B. aus Jira, 0001: laufende Nummer)
- Behält ihre Kernfunktion über den gesamten Lebenszyklus weitgehend unverändert bei
- Ändert sich im Laufe der Weiterentwicklung
- Verfügt über testbare Akzeptanzkriterien
- Beschreibt nicht die Bedienung
- Enthält keine subjektiven Bewertungen oder Beschreibungen
- Erläutert fachliche Hintergründe
- Verweist auf zugehörige Fachfunktionen

Die Bedeutung der Detailtiefe bei der Formulierung von Fachfunktionen und Akzeptanzkriterien lässt sich folgendermaßen verdeutlichen: Gibt man einer Entwicklerin/einem Entwickler die Fachfunktionen und ausreichend Zeit, entsteht eine Software, die sich anders bedient, anders programmiert ist und in Details unterschiedlich funktioniert – aber alle geforderten Geschäftsprozesse abbildet.

Welche Tests eignen sich für die Anforderungsabdeckung?

Die Verknüpfung eines Tests mit einem Akzeptanzkriterium ist einfach, stellt aber zunächst nur eine semantische Verbindung dar. Doch welche Tests eignen sich zur Auswertung der Anforderungsabdeckung?

Prinzipiell kann jeder Test verwendet werden, der im Rahmen eines Testrunners ausgeführt wird und dessen Name sowie Ergebnis (Success/Failed/Skipped) in einem Report erscheint. Inhaltlich gibt es jedoch Unterschiede. Gehen wir die (vereinfachte) Testpyramide von unten nach oben durch.

Abbildung 2 zeigt eine Übersicht der Teststufen und deren Verwendung zur Anforderungsabdeckung.

Unit-Tests laufen schnell und eignen sich für viele Akzeptanzkriterien gut. Hier ist jedoch Vorsicht geboten: False-Positives (also Tests, die ein erfolgreiches Verhalten vortäuschen, obwohl es technisch nicht zutrifft) sind möglich. Angenommen, ein Akzeptanzkriterium besagt: „Der Login eines Benutzers

muss der Regex /^[a-zA-Z0-9]{5,20}$/ entsprechen." Die Prüfung des eingegebenen Usernamens erfolgt in einer Methode, die durch einen Unit-Test abgedeckt wird. Wird diese Methode allerdings im Code nicht verwendet, besteht die Gefahr, dass der Test grün ist und somit fälschlicherweise eine erfolgreiche Abdeckung signalisiert. In solchen Fällen helfen Code-Reviews oder Tests auf höherer Ebene, zum Beispiel Komponententests.

Komponententests prüfen nicht nur einzelne Klassen, sondern ganze Komponenten – oft mit gemockten Umsystemen. Über API-Calls oder ähnliche Schnittstellen wird das Akzeptanzkriterium getestet. So könnte geprüft werden, ob es möglich ist, einen Benutzer mit dem Namen „Hugo?#012" anzulegen. Das Mocken der Umsysteme ist wichtig, damit die Tests in der CI/CD-Pipeline reproduzierbar sind, beliebig oft parallel ausgeführt werden können und unabhängig von externen Systemen funktionieren. Ziel ist es, die Dokumentation selbst zu testen, nicht die Kompatibilität mit Umsystemen. Tests, die Umsysteme einbeziehen, sind zwar sinnvoll für das Gesamtprojekt, dürfen aber nicht Teil der Anforderungsabdeckung sein.

Aus diesem Grund sind **Systemtests** oder **Systemintegrationstests** zur Verifikation von Akzeptanzkriterien ungeeignet. Lässt sich ein Akzeptanzkriterium nur mit Systemintegrationstests prüfen, ist es sehr wahrscheinlich falsch formuliert, da es externe Abhängigkeiten hat.

Ob ein Test das Akzeptanzkriterium inhaltlich gut testet, lässt sich nicht automatisch bestimmen – hier sind Code-Reviews und reflektiertes Mitdenken unabdingbar. Ein „assert true" würde auch zu einem erfolgreichen Test und damit zu einer erfolgreichen Anforderungsabdeckung führen.

In komplexeren Projekten empfiehlt es sich, im Testkonzept klar festzulegen, welche Tests wann und wie zur Verifikation herangezogen werden.

Weitere Empfehlungen aus der Praxis

Um den vollen Mehrwert der Dokumentation zu nutzen, muss sie jederzeit und für alle Projektbeteiligten transparent und einfach zugänglich sein. Nur so können Entwickler:innen, Tester:innen, Product Owner und weitere Beteiligte effektiv zusammenarbeiten und auf derselben Wissensbasis agieren. Eine zentrale, aktuelle Dokumentation unterstützt schnelle Abstimmungen, reduziert Missverständnisse und verhindert Doppelarbeit. Deshalb empfiehlt es sich,

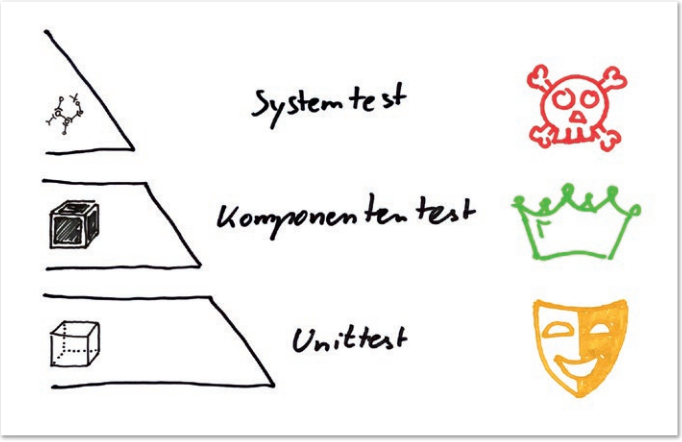


Abbildung 2: Einordnung von Testarten zur Anforderungsabdeckung (© Felix Tensing)

die Publizierung der Dokumentation – sei es als HTML- oder PDF-Version – fest in die Build- oder Release-Pipeline zu integrieren und so einen automatisierten, stets aktuellen Zugriff sicherzustellen.

In der Praxis zeigt sich schnell, dass Fachfunktionen zu einer gemeinsamen Referenz im Team werden. Statt lange Erklärungen zu liefern, heißt es dann einfach: „Fachfunktion 15". Durch die eindeutige Kennung ist für alle klar, welche Funktionalität gemeint ist. Auch User Stories nehmen direkt Bezug auf Fachfunktionen und deren Akzeptanzkriterien – etwa mit Formulierungen wie: „FF-BSPP-0015#05 entfällt, dafür kommen die beiden Kriterien ... hinzu." Dieser gemeinsame Bezugspunkt hilft, Missverständnisse zu vermeiden und macht die Kommunikation im Team effizienter.

Zwar liegt die Verantwortung für die finale Formulierung von Fachfunktionen und Akzeptanzkriterien bei den Entwickler:innen, doch sinnvoll strukturierte Vorgaben können diesen Prozess erheblich erleichtern. Besonders hilfreich sind Vorlagen, die sich am gewünschten Endergebnis orientieren oder sogar direkt aus der Story übernommen werden können.

Die letztgültige Beschreibung muss jedoch von denjenigen kommen, die die Umsetzung kennen – also von den Entwickler:innen selbst. Nur sie wissen genau, wie eine Anforderung tatsächlich implementiert wurde. Deshalb ist es entscheidend, dass sie die Beschreibung so präzise und fachlich verständlich wie möglich formulieren – als verbindliche Referenz für Tests, Dokumentation und spätere Weiterentwicklung.

Stellt sich in der täglichen Arbeit die Frage: „Ist das Verhalten meiner Anwendung ein Feature oder ein Bug?", sollte man als Erstes einen Blick in die Dokumentation werfen. Findet sich dort keine klare Antwort, wurde eine Dokumentationslücke entdeckt, die idealerweise zeitnah geschlossen werden sollte. Auch bei Bugs gilt: Zuerst die Dokumentation prüfen, dann die Tests und zuletzt die Implementierung untersuchen.

Eine häufige Frage ist der passende Dokumentations-Schnitt bei größeren Projekten – etwa bei Microservice-Architekturen oder typischen Client-Server-Systemen mit separatem Frontend und Backend. Hier sollte jedes Artefakt eigenständig dokumentiert werden. Erstens, weil die Testbarkeit von Fachfunktionen über Systemgrenzen hinweg sehr komplex wäre. Und zweitens, weil das Ziel ist, jeweils genau ein System zu beschreiben. Möchte man beispielsweise das Frontend durch eine App ersetzen, ist es essenziell, genau zu wissen, wie das Backend funktioniert. Eine Vermischung von Backend- und Frontend-Dokumentation widerspricht dieser Systematik.

Natürlich ist es möglich, zusätzlich eine übergreifende Dokumentation zu erstellen, die Fachfunktionen aus mehreren Systemen verlinkt und deren Zusammenhänge erläutert. Diese Art der Dokumentation ist klassisch und enthält in der Regel keine Akzeptanzkriterien oder Verknüpfungen zu Tests.

Typische Fallstricke

Auch wenn die Verknüpfung von Tests mit Fachfunktionen die Dokumentation robuster macht, gibt es typische Fehlerquellen, die man kennen sollte.

Ein inhaltlich unpassender oder ungenauer Test kann eine trügerische Sicherheit vermitteln. Wird etwa ein Test fälschlich mit dem Akzeptanzkriterium #04 statt #05 verknüpft – oder prüft nicht exakt das beschriebene Verhalten – erscheint das Kriterium als abgedeckt, obwohl es inhaltlich nicht korrekt getestet wurde. Noch schwerwiegender sind fehlende Akzeptanzkriterien: Sie tauchen in der Auswertung nicht auf und ihre Abwesenheit bleibt daher unbemerkt.

Ein guter Hinweis für die Praxis: Entwickler:innen sollten sich beim Schreiben von Code – insbesondere bei *if*- oder *switch*-Anweisungen – bewusst fragen, ob die jeweilige Logik durch ein konkretes Akzeptanzkriterium beschrieben ist. So lassen sich fehlende oder lückenhafte Kriterien frühzeitig erkennen.

Auch unpräzise formulierte Akzeptanzkriterien führen zu Problemen – insbesondere, wenn sie unbemerkt mehrere Anforderungen enthalten. Mehr dazu im Kasten: *Akzeptanzkriterien richtig formulieren*.

Akzeptanzkriterien richtig formulieren

Gute Akzeptanzkriterien sind präzise, atomar und testbar. Die folgenden Regeln helfen bei der Formulierung:

- **Atomar statt verschachtelt**
  Vermeide „und", „oder" sowie Aufzählungen mit Kommas. Sie deuten oft auf mehrere Aussagen hin – besser ist die Aufteilung in eigene Kriterien.
- **Keine vagen Begriffe**
  Formulierungen wie „intuitiv", „schnell" oder „komfortabel" sind subjektiv und nicht testbar. Stattdessen messbare Bedingungen verwenden (z. B. „Antwortzeit < 300ms").
- **Positiv und eindeutig formulieren**
  Negative Aussagen („darf nicht …") allein reichen nicht – was stattdessen passieren soll, muss klar benannt werden.
- **Fachlich statt technisch**
  Beschreibe, was aus Sicht der Nutzer:innen passiert, nicht wie es technisch umgesetzt ist.
- **Bedingungen trennen**
  Wenn ein Verhalten von Bedingungen abhängt („Wenn X, dann Y und Z"), sollten daraus getrennte Kriterien entstehen.

Ein häufiger Fehler beim Umgang mit Identifikatoren: Weder Fachfunktions- noch Kriteriumskennungen sollten erneut vergeben werden, wenn eine Funktion zuvor entfallen ist. Häufig werden diese Kennungen in E-Mails, Tickets oder Issues referenziert. Wird eine ehemals vergebene ID später neu vergeben, führt das zu Unklarheiten über den tatsächlichen Inhalt. Umgekehrt gilt: Ist eine ID in der aktuellen Dokumentation nicht mehr enthalten, wurde sie entfernt – ein völlig normaler Vorgang, der sich mit älteren Dokumentationsversionen nachprüfen lässt.

Schließlich ist auch von einer inhaltlichen Kodierung in der ID (zum Beispiel zur fachlichen Gruppierung) abzuraten. Fachfunktionen ändern sich mit der Zeit – ihre anfängliche Klassifikation kann dabei schnell veralten. Stattdessen empfiehlt sich eine einfache fortlaufende Nummerierung. Für eine inhaltliche Strukturierung bieten sich Tags an, wie sie auch in *Listing 2* und *Abbildung 1* gezeigt werden.

Die Integration von Dokumentation, Tests und Code im beschriebenen Ansatz bringt zahlreiche Vorteile für Teams und Projekte:

- **Effiziente Code-Reviews:** Dokumentation, Tests und Code bilden eine Einheit (siehe Abbildung 2) und werden gemeinsam geprüft, was die Qualitätssicherung deutlich verbessert.
- **Automatisierte Anforderungsabdeckung:** Das Team erhält eine verlässliche Selbstkontrolle und kann Schwachstellen frühzeitig erkennen. Der Testnachweis unterstützt insbesondere in regulierten Umgebungen häufigere und schnellere Releases.
- **Frühe Fehlererkennung:** Bereits das Lesen der entstandenen Dokumentation nach der Umsetzung zeigt mögliche Missverständnisse oder Fehlumsetzungen auf. Das Lesen der Dokumentation ist somit ein wichtiger erster Schritt im Review-Prozess.
- **Einheitliche Team-Sprache:** Fachfunktionen und Akzeptanzkriterien schaffen klare Vorgaben und reduzieren Missverständnisse. Dabei ist darauf zu achten, dass ein gemeinsamer Wortschatz verwendet wird.
- **Verbesserte Nachvollziehbarkeit:** Fachfunktions-IDs als Stichworte in Tickets helfen, alle zugehörigen Stories und Bugs schnell zu finden – eine Rückverfolgbarkeit, die auch in regulierten Umgebungen gefordert wird.
- **Best Practices von Documentation as Code:** Versionierbarkeit, textbasierte Änderungsverfolgung (Diffs) und die Einbindung von Code-Snippets unterstützen das Schreiben verständlicher Dokumentation mit geringem Aufwand.
- **Flexibilität:** Das Vorgehen lässt sich um eigene Felder in Fachfunktionen erweitern. Zudem ist eine Integration in Frameworks wie Storybook [5] oder Docusaurus [6] möglich.

Fazit

Die konsequente und strukturierte Dokumentation fachlicher Anforderungen als Fachfunktionen bildet die Grundlage für eine effektive Testabdeckung und eine nachhaltige Qualitätssicherung im Sinne des Shift-Left-Prinzips. Sie hilft Entwickler:innen, die Frage zu beantworten: Wie viel muss ich testen? Durch die enge Verzahnung *(siehe Abbildung 3)* von Dokumentation, Tests und Code entsteht ein transparentes, nachvollziehbares und stets aktuelles Abbild der Softwarefunktionalität.

Der Ansatz fördert die Kommunikation im Team, reduziert Missverständnisse und ermöglicht eine verlässliche Anforderungsabdeckung – selbst in komplexen Projekten und regulierten Umgebungen. Die Integration in bestehende Entwicklungsprozesse mit modernen Werkzeugen sorgt dafür, dass Dokumentation nicht als lästige Pflicht, sondern als wertvoller Teil der täglichen Arbeit erlebt wird. Den Mehrwert einer guten Dokumentation erkennen Teams meist bereits nach kurzer Zeit.

Das vorgestellte Beispielprojekt bietet einen guten Ausgangspunkt, um das Vorgehen in das eigene Projekt zu übernehmen und an die eigenen Bedürfnisse anzupassen. Es lädt zudem dazu ein, Erfahrungen und Erweiterungen mit der Community zu teilen – so profitiert nicht nur das eigene Team, sondern auch die gesamte Entwicklergemeinschaft.

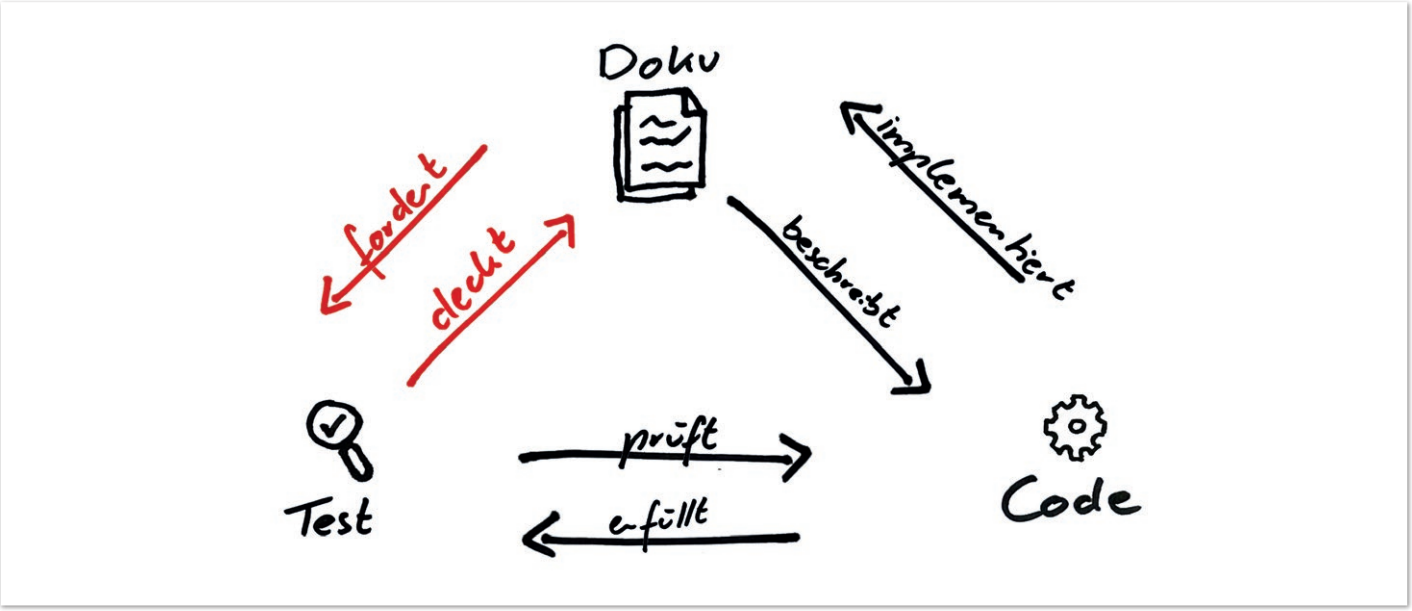


Abbildung 3: Zusammenhänge zwischen Code, Test und Dokumentation (© Felix Tensing)

Insgesamt zeigt sich: Wer frühzeitig Anforderungen klar, präzise und testbar dokumentiert, legt den Grundstein für stabile Software, effiziente Reviews und schnellere, qualitativ hochwertige Releases.

Quellen

[1] Kaner, Falk, Nguyen (1999): Testing Computer Software, 2nd Edition. Wiley.

[2] AsciiDoc-Dokumentation: *https://asciidoc.org*

[3] PlantUML-Dokumentation: *https://plantuml.com/de*

[4] Spock – the enterprise ready specification framework *https://spockframework.org*

[5] Storybook *https://storybook.js.org*

[6] Docusaurus *https://docusaurus.io*

[7] Beispielprojekt auf GitHub: *https://github.com/ziffit/agile-documentation*

Felix Tensing

Nürnberger Versicherung

felix.tensing@nuernberger.de

Felix Tensing ist Java Fullstack-Entwickler mit einer Leidenschaft für Testen, Dokumentation sowie agiles und cross-funktionales Arbeiten. Seit über 20 Jahren ist er in der IT tätig, vor allem im regulierten Umfeld. Wenn er nicht gerade als Speaker auf einer Konferenz auftritt, verbringt er seine Freizeit mit seinen Kindern, beim Angeln oder beim Umsetzen kreativer Projekte am 3D-Drucker – ganz ohne Dokumentation, Tests und doppelten Boden.