LU16.L02 – BDD in der eigenen Sprache

Lösungsbeispiel zu LU16.A02 – BDD in der eigenen Sprache. Weil jede Person Sprache und Framework selbst wählt, gibt es keine einheitliche Lösung. Diese Seite zeigt ein vollständiges Beispiel in JavaScript mit **Cucumber.js** und listet, worauf bei der Bewertung geachtet wird.

Auftrag 1 – Framework und Setup

- **Sprache:** JavaScript (Node.js 20 oder neuer)
- **Framework:** Cucumber.js, Version 11
- **Installation:**

```
npm init -y
npm install --save-dev @cucumber/cucumber
```

- **Ausführen:** `npx cucumber-js` oder `npm test` (Script in `package.json`)

Projektstruktur

```
warenkorb-bdd/
  ├── package.json
  ├── src/
  │     └── cart.js
  └── features/
        ├── cart.feature
        └── step_definitions/
              └── cart_steps.js
```

Auftrag 2 – Miniprojekt Warenkorb

features/cart.feature

Cucumber unterstützt Gherkin auf Deutsch. Die erste Zeile `# language: de` schaltet die deutschen Schlüsselwörter ein (`Funktionalität`, `Szenario`, `Angenommen`, `Wenn`, `Dann`, `Und`).

```
# language: de
Funktionalität: Warenkorb
  Als Kundin eines Onlineshops
  möchte ich Artikel in den Warenkorb legen
  damit ich vor dem Bezahlen das Total sehe.

  Szenario: Total mit mehreren Artikeln
    Angenommen ein leerer Warenkorb
```

```
    Wenn ich 2 Stück "Apfel" zu 0.80 Franken hinzufüge
    Und ich 1 Stück "Brot" zu 4.35 Franken hinzufüge
    Dann beträgt das Total 5.95 Franken

  Szenario: Ungültige Menge
    Angenommen ein leerer Warenkorb
    Wenn ich 0 Stück "Apfel" zu 0.80 Franken hinzufüge
    Dann erhalte ich die Fehlermeldung "Die Menge muss mindestens 1 sein"
    Und beträgt das Total 0.00 Franken
```

features/step_definitions/cart_steps.js

```
const assert = require('node:assert');
const { Given, When, Then } = require('@cucumber/cucumber');
const { Cart } = require('../../src/cart');

Given('ein leerer Warenkorb', function () {
  this.cart = new Cart();
  this.error = null;
});

When('ich {int} Stück {string} zu {float} Franken hinzufüge', function
(quantity, name, price) {
  try {
    this.cart.add(name, price, quantity);
  } catch (error) {
    this.error = error;
  }
});

Then('beträgt das Total {float} Franken', function (expected) {
  assert.ok(Math.abs(this.cart.total() - expected) < 0.005, `Total
${this.cart.total()} statt ${expected}`);
});

Then('erhalte ich die Fehlermeldung {string}', function (message) {
  assert.ok(this.error, 'Es wurde kein Fehler ausgelöst');
  assert.strictEqual(this.error.message, message);
});
```

src/cart.js

```
// Ein Warenkorb mit Artikeln und Mengen
class Cart {
  constructor() {
    this.items = [];
  }
```

```
  add(name, price, quantity = 1) {
    if (quantity < 1) {
      throw new Error('Die Menge muss mindestens 1 sein');
    }
    this.items.push({ name, price, quantity });
  }

  total() {
    return this.items.reduce((sum, item) => sum + item.price *
item.quantity, 0);
  }
}

module.exports = { Cart };
```

Testlauf

```
$ npx cucumber-js
........

2 scenarios (2 passed)
8 steps (8 passed)
```

Auftrag 3 – Reflexion (Beispielantworten)

1. **Auswahl:** Cucumber ist das Original, aus dem behave entstanden ist. Es ist für viele Sprachen verfügbar und gut dokumentiert.
2. **Aufwand:** gering, ein npm-Paket. Länger dauert es, die Ausdrücke `{int}`, `{float}` und `{string}` in den Schritten richtig zu verwenden.
3. **Unterschiede zu behave:** gleiche Gherkin-Syntax und ähnliche Ordnerstruktur. Die Schritte heissen `step_definitions` statt `steps`, geteilte Daten liegen in `this` (World) statt in `context`. Platzhalter sind Cucumber Expressions statt `parse`-Formate.
4. **Stolpersteine:** Pfeilfunktionen `() ⇒ {}` haben kein eigenes `this`. Schritte müssen deshalb mit `function () {}` geschrieben werden. Fliesskommazahlen brauchen eine Toleranz.
5. **Vorteile:** Die Szenarien sind auch für Nicht-Programmierende lesbar und dienen als lebende Spezifikation.
6. **Ausbildungsbetrieb:** Akzeptanztests für Webshops oder Formulare, bei denen Fachpersonen die Szenarien mitschreiben.
7. **Verbesserung:** Für kleine Funktionen ist der Aufwand höher als bei Unit Tests. Eine Kombination ist sinnvoll: BDD für Abläufe, Unit Tests für Details.
8. **Berufsalltag:** ja, wenn Fachabteilung und Entwicklung eng zusammenarbeiten. Sonst genügen oft Unit- und Integrationstests.

Bewertungshinweise

Eine gute Abgabe

- nennt Framework, Version und Installationsbefehle,
- enthält mindestens ein Feature mit zwei Szenarien in korrekter Gherkin-Syntax (Given – When – Then),
- testet nicht nur den Normalfall, sondern auch einen Fehlerfall,
- zeigt einen erfolgreichen Testlauf (Screenshot oder Log),
- beantwortet alle acht Reflexionsfragen begründet und mit Bezug zum eigenen Projekt.

M450-LU16, M450-B1A, M450-C2I

Kevin Maurizi

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/modul/m450/learningunits/lu16/loesungen/own-bdd

Last update: **2026/11/17 17:45**