LU16e - Behaviour Driven Development (BDD) in Python

Einführung

Behaviour Driven Development (BDD) ist eine Weiterentwicklung von Test Driven Development (TDD). Beim BDD steht das Verhalten eines Systems aus Sicht der Benutzer:innen im Zentrum. Tests werden in einer natürlichen Sprache formuliert, die von allen verstanden wird — nicht nur von Programmierer:innen.

Ein populäres Framework für BDD in Python ist **behave**. Damit lassen sich **lesbare Akzeptanztests** in einer sogenannten **Gherkin-Syntax** schreiben und mit Python-Code verknüpfen.

Ziel von BDD

- Verständliche Tests schreiben, die auch Fachpersonen lesen können
- Anforderungen und Tests in einer gemeinsamen Sprache beschreiben
- Entwicklungsschritte von der Benutzerstory bis zum Test automatisieren
- Früh Feedback erhalten, ob das System das gewünschte Verhalten zeigt

Ausgangssituation: Eine kleine App

Stell dir vor, du entwickelst eine kleine **Applikation**, z. B. einen **Rechner**, der einfache Operationen (Addition, Subtraktion usw.) ausführt. Diese App soll so getestet werden, dass auch **nicht-technische Personen** verstehen, was überprüft wird.

Das Ziel ist nicht, die App zu programmieren – sondern **ihr Verhalten zu beschreiben und zu testen**. wie sie sich aus Sicht der Benutzer:innen verhalten *sollte*.

Aufbau eines Behave-Projekts

Ein Behave-Projekt ist typischerweise so aufgebaut:

- .feature-Dateien enthalten Szenarien in Gherkin-Sprache
- .py-Dateien im Unterordner steps/ enthalten die Python-Schritte dazu
- Das App-Modul (app/calculator.py) ist die getestete Anwendung

Gherkin-Syntax

Gherkin ist eine leicht lesbare Sprache, um das gewünschte Verhalten eines Systems zu beschreiben. Jede Feature-Datei beschreibt **eine Funktionalität** oder **Benutzerstory**.

Beispiel:

```
Feature: Einfacher Taschenrechner
Um einfache Rechnungen durchführen zu können
Möchte ich zwei Zahlen addieren können

Scenario: Zwei Zahlen addieren
Given ich habe 50 in die App eingegeben
And ich habe 70 in die App eingegeben
When ich die Additionsfunktion ausführe
Then sollte das Ergebnis 120 auf dem Bildschirm erscheinen
```

Schlüsselwörter:

- Feature → beschreibt die Funktion / das Ziel
- Scenario → einzelner Testfall
- Given → Ausgangslage (Vorbedingungen)
- When → Aktion (was passiert)
- Then → erwartetes Ergebnis
- And / But → Erweiterungen

Schrittdefinitionen und getestete App

Feature Steps (Testcode)

App-Code (getestete Anwendung)

```
from behave import given, when,
then
from app.calculator import
Calculator # zu testende App

@given('ich habe {zahl:d} in die
App eingegeben')
def step_input_number(context,
# app/calculator.py

class Calculator:
    def __init__(self):
        # interner Speicher für

eingegebene Zahlen
self.numbers = []
```

https://wiki.bzz.ch/ Printed on 2025/10/24 02:46

```
zahl):
                                          def enter number(self, value):
   # Erstellt (falls nötig) eine
                                              # Zahl zur Eingabeliste
neue App-Instanz
                                      hinzufügen
    if not hasattr(context,
                                              self.numbers.append(value)
"calc"):
        context.calc = Calculator()
                                          def add(self):
    context.calc.enter number(zahl)
                                              # Beispiel-Implementierung:
                                      summiert alle Werte
@when('ich die Additionsfunktion
                                              result = sum(self.numbers)
ausführe')
                                              # leert Eingaben für
                                      nächsten Vorgang
def step perform add(context):
   # Führt die gewünschte
                                              self.numbers.clear()
                                              return result
Operation aus
    context.result =
context.calc.add()
@then('sollte das Ergebnis
{expected:d} auf dem Bildschirm
erscheinen')
def step verify result(context,
expected):
   # Überprüft das sichtbare
Resultat der App
    assert context.result ==
expected
```

Hinweise:

- Das Modul calculator.py ist die **zu testende Applikation**, nicht Teil des Tests selbst.
- Die Datei calculator steps.py beschreibt nur das **erwartete Verhalten**.
- Der context erlaubt den Austausch von Objekten (z. B. der App) zwischen den Schritten.
- Jeder Given/When/Then-Schritt entspricht einem Teil der Benutzerinteraktion.

Tests ausführen

Im Terminal:

behave

Ergebnis (Auszug):

```
Feature: Einfacher Taschenrechner
Scenario: Zwei Zahlen addieren
Given ich habe 50 in die App eingegeben
And ich habe 70 in die App eingegeben
When ich die Additionsfunktion ausführe
Then sollte das Ergebnis 120 auf dem Bildschirm erscheinen
```

Last update: 2025/10/23 09:34

1 feature passed, 0 failed

Vorteile von Behave

- Tests sind verständlich für Entwickler:innen und Fachpersonen
- Klarer Bezug zu Anforderungen oder Benutzerstories
- Schritte sind wiederverwendbar für mehrere Szenarien
- Einfach **automatisierbar** in CI/CD-Pipelines (z. B. GitHub Actions, Jenkins)

Vergleich zu Unit Tests

Aspekt	Unit Test	BDD (Behave)
Fokus	einzelne Funktion / Methode	Verhalten aus Nutzersicht
Sprache	Code (Python)	Gherkin (natürliche Sprache)
Zielgruppe	Entwickler:innen	Fachpersonen + Entwickler:innen
Beispiel		"Given ich habe 2 und 3 eingegeben, When ich addiere, Then erhalte ich 5"

Installation

Installation mit pip:

pip install behave

Typische Fehlerquellen

- Schrittdefinition fehlt oder Schreibweise weicht ab
- Python-Datei liegt nicht im steps/-Ordner
- Context-Variable nicht initialisiert
- Feature-Datei hat keine Endung . feature

Weiterführende Links

• Offizielle Dokumentation: https://behave.readthedocs.io

Gherkin-Syntax Referenz: https://cucumber.io/docs/gherkin/reference/

Vergleich zu pytest-bdd: https://pytest-bdd.readthedocs.io

https://wiki.bzz.ch/ Printed on 2025/10/24 02:46

MXXX-LU16e



From:

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

Permanent link:

https://wiki.bzz.ch/modul/m450/learningunits/lu16/behant_python

Last update: 2025/10/23 09:34

