Programmierung von CAx-Systemen

David Straub

Software Engineering Basics

1. Versionsverwaltung mit Git
2. **Unittests mit Pytest**
3. Type Hints und statische Codeanalyse

Ein erster Unittest

```
from build123d import *
import math, pytest

def rohr(r, l):
    pfad = Polyline((0,0,0), (l,0,0))
    return sweep(pfad.location_at(0) * Circle(r), pfad)

def test_rohr_volumen():
    r, l = 5, 100
    erwartet = math.pi * r**2 * l        # Länge × Querschnittsfläche
    assert rohr(r, l).volume == pytest.approx(erwartet, rel=1e-3)
```

```
$ pytest -v
PASSED  test_rohr.py::test_rohr_volumen
```

Warum Unittests?

Regression: Das stille Problem

```
* 2e9d54a  Fix path orientation for curved pipes   ← test_rohr_volumen scheitert
* c04f81a  Add curved path support
* f3a17bc  Initial working version
```

Regression = eine Funktion war korrekt – und ist es nach einer unbeabsichtigten Änderung nicht mehr.

Ohne Tests: Fehler bleibt unbemerkt bis zum falschen Bauteil.
Mit Tests: `pytest` schlägt beim nächsten Commit sofort an.

Was Tests leisten

Eigenschaft	Effekt
Regression erkennen	Jede Änderung wird automatisch geprüft
Refactoring ermöglichen	Code umbauen ohne Risiko
Verhalten dokumentieren	Test zeigt, was die Funktion garantiert
Spezifikation	Erwartetes Verhalten vor dem Code festhalten

Testgetriebene Entwicklung (TDD)

Red → Green → Refactor

1. **Red** – Test schreiben, der noch scheitert (Funktion existiert noch nicht)
2. **Green** – minimale Implementierung, die den Test besteht
3. **Refactor** – Code verbessern, Tests müssen grün bleiben

Tests legen das **erwartete Verhalten** fest, bevor die Implementierung entsteht.

Was ist ein Unittest?

Unittest: testet eine einzelne Funktion isoliert – ohne Dateisystem, Netzwerk oder andere Module.

	Unittest	Integrationstest
Testet	eine Funktion	Zusammenspiel mehrerer Komponenten
Abhängigkeiten	keine	Dateisystem, DB, externe APIs …
Geschwindigkeit	Millisekunden bis Sekunden	Sekunden bis Minuten
Fehlerquelle	klar lokalisiert	schwerer einzugrenzen

Aufbau: Arrange – Act – Assert

```python
def test_rohr_volumen():
    # Arrange: Eingaben und Sollwert vorbereiten
    r, l = 5, 100
    erwartet = math.pi * r**2 * l

    # Act: Funktion aufrufen
    teil = rohr(r, l)

    # Assert: Ergebnis prüfen
    assert teil.volume == pytest.approx(erwartet, rel=1e-3)
```

Arrange: Eingaben und Sollwert vorbereiten
Act: Die zu testende Funktion aufrufen
Assert: Ergebnis gegen Sollwert prüfen

Was testen

Nicht nur den Normalfall:

```python
def test_rohr_normalfall():
    assert rohr(5, 100).volume == pytest.approx(math.pi * 25 * 100, rel=1e-3)
```

```python
def test_rohr_duenn():
    assert rohr(0.5, 100).volume == pytest.approx(math.pi * 0.25 * 100, rel=1e-2)

def test_rohr_kurz():
    assert rohr(5, 1).volume == pytest.approx(math.pi * 25 * 1, rel=1e-2)
```

Grenzfälle (sehr klein, sehr groß, null) decken häufige Bugs auf, die im Normalfall unsichtbar bleiben.

Pytest

Dateistruktur

```
projekt/
├── rohr.py          ← zu testende Funktion
└── test_rohr.py     ← Testdatei
```

Pytest findet Tests automatisch wenn: - Dateiname beginnt mit `test_` oder endet mit `_test.py` - Funktionsname beginnt mit `test_`

```python
# test_rohr.py
import math, pytest
from rohr import rohr

def test_rohr_volumen():
    r, l = 5, 100
    assert rohr(r, l).volume == pytest.approx(math.pi * r**2 * l, rel=1e-3)
```

Ausführen

```
pytest                                          # alle Tests
pytest test_rohr.py                             # nur diese Datei
pytest -v                                       # ausführliche Ausgabe
pytest -v test_rohr.py::test_rohr_normalfall    # einzelner Test
```

```
PASSED  test_rohr.py::test_rohr_normalfall     [ 50%]
FAILED  test_rohr.py::test_rohr_kurz           [100%]

FAILED test_rohr.py::test_rohr_kurz
  AssertionError: assert 166 == 190
```

Pytest zeigt genau welcher Test scheitert – und warum.

pytest.approx – Fließkommazahlen

```python
# Schlecht – zufällige Fehler durch Fließkomma-Rundung:
assert rohr(5, 100).volume == 7853.981633974483

# Gut:
assert rohr(5, 100).volume == pytest.approx(7853.98, rel=1e-3)   # 0,1 % Toleranz
assert rohr(5, 100).volume == pytest.approx(7853.98, abs=1.0)    # ±1 mm³
```

Warum?

```
>>> 0.1 + 0.2 == 0.3
False
>>> 0.1 + 0.2
0.30000000000000004
```

Geometrieoperationen akkumulieren solche Rundungsfehler.

CI – Continuous Integration (GitLab)

Bei jedem Push führt GitLab automatisch `pytest` aus:

```yaml
# .gitlab-ci.yml
test:
  script:
    - pip install pytest build123d
    - pytest
```

Ergebnis: grünes oder rotes direkt am Commit bzw. Merge Request.
Kein manuelles Ausführen nötig – Regression wird sofort sichtbar.

CI – Continuous Integration (GitHub)

GitHub Actions erfordert explizite Schritte – Checkout und Runner-Umgebung müssen angegeben werden:

```yaml
# .github/workflows/tests.yml
name: Tests
on: [push]
jobs:
  test:
```

```yaml
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: pip install pytest build123d
      - run: pytest
```

Prinzip identisch: Pipeline-Konfiguration liegt als Datei im Repository.

Antipatterns

Reimplementierung

```python
def zylindervolumen(r, h):
    return math.pi * r**2 * h

def test_zylindervolumen():
    r, h = 5, 10
    assert zylindervolumen(r, h) == pytest.approx(math.pi * r**2 * h, rel=1e-6)
```

Funktion und Test kommen aus derselben Quelle. Steckt `π·r·h` statt `π·r²·h` im Kopf, steht das in beiden – Test grün, Ergebnis falsch.

Weitere Antipatterns

- **Kein Assert** – `print(teil.volume)` statt `assert` – Test „grünt" immer, beweist nichts.
- **Zu viel auf einmal** – Wenn der Test scheitert, bleibt unklar was falsch ist. Eine Testfunktion = ein Verhalten.
- **Interne Details testen** – Testet *wie* die Funktion implementiert ist, nicht *was* sie zurückgibt. Bricht bei Refactoring, auch wenn das Verhalten korrekt bleibt.
- **Reihenfolgeabhängigkeit** – Test B setzt Zustand aus Test A voraus. Jeder Test muss unabhängig und in beliebiger Reihenfolge ausführbar sein.
- **Trivialen Code testen** – Einfache Zuweisungen ohne Logik brauchen keinen Test – kein Mehrwert, nur mehr Wartungsaufwand.

CAD-Code testen

Testbarer Code: Reine Funktionen

```python
# Schwer zu testen – Geometrie, Export und Anzeige vermischt
def zelle_bauen():
    zelle = revolve(...)
    show(zelle)
    export_step(zelle, "zelle.step")
    return zelle

# Leicht zu testen – reine Funktion, nur Geometrie
def make_18650(r_aussen: float, h_zelle: float) -> Part:
    kurve = Polyline(...)
    return revolve(Plane.XZ * make_face(kurve), axis=Axis.Z)
```

Geometrie erzeugen und Export/Anzeige trennen. Der Test ruft nur die Geometriefunktion auf.

Was lässt sich prüfen?

```python
zelle = make_18650(r_aussen=9.0, h_zelle=65.0)

# 1. Existenz und geometrische Gültigkeit
assert zelle.volume > 0
assert zelle.is_valid          # OCCT-interner Konsistenzcheck

# 2. Volumen – analytische Näherung als unabhängiger Sollwert
assert zelle.volume == pytest.approx(math.pi * 9**2 * 65, rel=0.05)

# 3. Außenmaße – Begrenzungsrahmen
bb = zelle.bounding_box()
assert bb.size.Z == pytest.approx(65.0, abs=0.1)
assert bb.min.X  == pytest.approx(-9.0, abs=0.1)

# 4. Flächenanzahl – kodiert Konstruktionsabsicht
assert len(zelle.faces()) == 5
```

BoundBox – Eigenschaften

`shape.bounding_box()` gibt den achsenparallelen Hüllquader in **Weltkoordinaten** zurück.

Attribut	Typ	Bedeutung
`bb.size`	`Vector`	Ausmaße (X, Y, Z)
`bb.min` / `bb.max`	`Vector`	Ecken des Quaders in Weltkoordinaten
`bb.center()`	`Vector`	Mittelpunkt
`bb.diagonal`	`float`	Länge der Raumdiagonalen

```
assert bb.size.Z == pytest.approx(65.0, abs=0.1)   # Höhe
assert bb.size.X == pytest.approx(18.0, abs=0.1)   # Durchmesser
assert bb.min.Z  == pytest.approx(0.0,  abs=0.1)   # liegt auf XY-Ebene
```

`is_valid` – Was steckt dahinter?

OCCT-interner Konsistenzcheck des Geometriekerns.

Prüft: - Topologische Konsistenz – Kanten, Flächen, Vertices korrekt verbunden - Übereinstimmung von 3D-Kurven und 2D-Parameterkurven auf Flächen

Prüft nicht: - `volume > 0` – Nulldicke kann gültig sein - Selbstdurchdringungen und geometrischen Sinn - Ob ein fehlgeschlagener Boolean lautlos ein leeres `Compound` erzeugt hat

→ Notwendige, nicht hinreichende Bedingung. Immer zusätzlich `volume > 0` und Maßchecks.

Vollständiges Beispiel

```
import math, pytest
from build123d import *
from mein_projekt import make_18650

def test_18650_geometrie():
    zelle = make_18650(r_aussen=9.0, h_zelle=65.0)

    assert zelle.volume > 0
```

```
    assert zelle.is_valid

    bb = zelle.bounding_box()
    assert bb.size.Z == pytest.approx(65.0, abs=0.1)    # Höhe
    assert bb.size.X == pytest.approx(18.0, abs=0.1)    # Durchmesser

    # Boden + Mantel + Schulter + Terminal-Mantel + Deckel
    assert len(zelle.faces()) == 5
```

Randwerte: Robustheit prüfen

Parametrische Funktionen werden mit vielen Eingaben aufgerufen. Teste nicht nur den Normalfall:

```
def test_18650_standard():
    assert make_18650(r_aussen=9.0, h_zelle=65.0).is_valid

def test_21700_format():
    assert make_18650(r_aussen=10.5, h_zelle=70.0).is_valid

def test_kleinformat():
    assert make_18650(r_aussen=5.0, h_zelle=20.0).is_valid
```

Ein fehlgeschlagener Boolean oder ein entarteter Sweep kann eine topologisch inkonsistente Form erzeugen, die im Viewer plausibel aussieht – `is_valid` fängt das.

Spiel und Kollisionsprüfung

Boolescher Schnitt zweier Körper: Volumen > 0 bedeutet Kollision.

```
def test_keine_kollision_im_modul():
    zelle = make_18650(r_aussen=9.0, h_zelle=65.0)
    nachbar = Pos(20, 0) * zelle        # 20 mm Achsabstand
    schnitt = zelle & nachbar
    assert schnitt.volume == pytest.approx(0.0, abs=0.1)

def test_zelle_passt_in_ausschnitt():
    zelle = make_18650(r_aussen=9.0, h_zelle=65.0)
```

```
    bb = zelle.bounding_box()
    ausschnitt_radius = 9.5             # 0,5 mm Spiel
    assert bb.size.X / 2 < ausschnitt_radius
    assert bb.size.Y / 2 < ausschnitt_radius
```

Aufgabe: Rundzelle testen

Teil 1 – Tests schreiben

Gegeben ist diese einfache Rundzelle aus zwei Zylindern:

```
def make_round_cell(r_outer=9.0, h_cell=65.0, r_terminal=2.5, h_terminal=1.0):
    body = Cylinder(r_outer, h_cell)
    terminal = Pos(0, 0, h_cell/2) * Cylinder(r_terminal, h_terminal,
                   align=(Align.CENTER, Align.CENTER, Align.MIN))
    return body + terminal
```

Schreiben Sie Unittests, die prüfen:

1. Geometrische Gültigkeit (`is_valid`, `volume > 0`)
2. Gesamtvolumen (analytischer Sollwert: $\pi r_a^2 h + \pi r_t^2 h_t$)
3. Gesamthöhe über `bounding_box()`
4. Anzahl der Flächen

Teil 2 – Refactoring

Ersetzen Sie `Cylinder(...)` durch `extrude(Circle(...), h)` – die Tests müssen danach grün bleiben.

```
def make_round_cell(r_outer=9.0, h_cell=65.0, r_terminal=2.5, h_terminal=1.0):
    body = extrude(Plane.XY.offset(-h_cell/2) * Circle(r_outer), h_cell)
    terminal = Pos(0, 0, h_cell/2) * extrude(Circle(r_terminal), h_terminal)
    return body + terminal
```

Führen Sie `pytest` erneut aus – gleiche Tests, andere Implementierung.

Teil 3 – Crimping-Nut

Erweitern Sie `make_round_cell` um eine umlaufende Einschnürung am oberen Rand:

```
def make_round_cell(r_outer=9.0, h_cell=65.0, r_terminal=2.5, h_terminal=1.0,
                    crimp_depth=0.5, crimp_height=2.0):
    align_bot = (Align.CENTER, Align.CENTER, Align.MIN)
    body = Cylinder(r_outer, h_cell)
```

```
    terminal = Pos(0, 0, h_cell/2) * Cylinder(r_terminal, h_terminal,
                   align=align_bot)
    cell = body + terminal
    crimp_z = h_cell/2 - crimp_height
    nut = (Pos(0, 0, crimp_z) * Cylinder(r_outer, crimp_height, align=align_bot)
           - Pos(0, 0, crimp_z) * Cylinder(r_outer - crimp_depth, crimp_height, align=align_
    return cell - nut
```

Sind die Tests noch grün?