Programmierung von CAx-Systemen

David Straub

Gliederung

1. Einführung
2. Topologie
3. Grundformen
4. Kurven
5. Freiformgeometrie
6. Profile
7. **Codequalität**
8. Datenaustausch
9. Robustheit
10. Simulation
11. Optimierung

Codequalität

- **Type Hints:** Fehler sichtbar, bevor der Code läuft
- **Unittests:** jede Änderung automatisch geprüft
- **CI:** die Prüfung läuft bei jedem Push

Durchgängiges Beispiel: der Modul-Code aus sechs Wochen – annotiert und durch Tests abgesichert

Rückblick: der Modul-Code wächst

Sechs Wochen: `ModulParam`, `zelle_bauen`, `stapel_bauen`, `endplatten`, Loft, Sweep.

Inzwischen berührt jede Änderung Bestehendes – der Fillet-Radius von heute entscheidet über die Zell-Passung von letzter Woche.

Heute: zwei Werkzeuge, die Fehler früh melden –

- **Type Hints** zeigen Typfehler schon im Editor
- **Unittests** prüfen die Geometrie bei jeder Änderung

Theorie A: Type Hints

Warum Type Hints?

```python
def finde_zelle(name):
    for z in katalog:
        if z.name == name:
            return z
    # nicht gefunden → kein return → implizit None
```

```python
kapazitaet = finde_zelle("Pouch-40Ah").kapazitaet   # Tippfehler im Namen
# → AttributeError: 'NoneType' has no attribute 'kapazitaet'
```

Der Absturz passiert **weit entfernt** vom eigentlichen Fehler – und erst, wenn der Code läuft.

Warum findet Python das nicht?

Python prüft Typen **zur Laufzeit**, und nur an den Stellen, die tatsächlich ausgeführt werden:

```python
len("hallo")     # → 5
len([1, 2, 3])   # → 3
```

Duck Typing: Beim Aufruf zählt allein, ob das Objekt die verlangte Operation beherrscht. Einen deklarierten Typ gibt es nicht.

`None` beherrscht fast nichts – der Fehler fliegt dort auf, wo das Ergebnis *benutzt* wird.

Dynamisch und statisch typisiert

	dynamisch	statisch
Typ steht fest	zur Laufzeit, pro Objekt	im Quelltext, pro Variable
geprüft wird	beim Ausführen	vor dem Ausführen
Fehler sichtbar	wenn die Zeile dran ist	sofort im Editor / Compiler
Sprachen	Python, JavaScript	C, Java, Rust

In C weist der Compiler `int x = "hallo";` ab, bevor das Programm existiert. Python nimmt jede Zuweisung an und stolpert erst beim Benutzen.

Python: dynamisch typisiert – und annotierbar

Seit Python 3.5 (2015) gibt es eine Syntax, um den erwarteten Typ **hinzuschreiben**:

```python
def finde_zelle(name: str) -> Zelle | None:
    ...
```

`name: str` ist ein Hinweis. Python ignoriert ihn vollständig und führt den Code genauso aus – auch bei falscher Annotation:

```python
def verdopple(x: str) -> int:
    return x * 2

verdopple(5)      # läuft durch, ohne Warnung
```

Statische Codeanalyse: lesen, ohne auszuführen

Die Annotationen wertet ein **zweites Programm** aus. Es liest den Quelltext wie ein Compiler, verfolgt die Typen durch die Funktionen und meldet Widersprüche – ohne eine einzige Zeile auszuführen:

Tool	Wann	Wie
Pylance	beim Tippen	Unterwellenlinie im Editor (VS Code)
mypy	Kommandozeile / CI	`mypy w07/`

```python
kapazitaet = finde_zelle("Pouch-40Ah").kapazitaet
#            ⚠️ Cannot access attribute "kapazitaet" for class "None"
```

Damit fällt das vergessene `None` auf, **bevor** der Code jemals läuft.

mypy: der Prüfer für die Kommandozeile

mypy ist ein eigenes Python-Programm. Man installiert es einmal und ruft es auf einen Ordner auf – es liest die Dateien und gibt eine Liste aus:

```bash
pip install mypy
mypy w07/
```

```
w07/zelle.py:42: error: Item "None" of "Zelle | None" has no attribute "kapazitaet"
w07/stapel.py:17: error: Argument 1 has incompatible type "str"; expected "int"
Found 2 errors in 2 files (checked 6 source files)
```

Dieselbe Prüfung wie im Editor, nur als Kommando – deshalb lässt sie sich in die CI hängen (Theorie B).

Syntax: Parameter, Rückgabe, Optional

```python
def zellmasse(kapazitaet_ah: float, spez_energie: float) -> float:
    return kapazitaet_ah * 3.7 / spez_energie

def exportiere(koerper: cf.Shape, pfad: str) -> None:
    koerper.exportStep(pfad)

def finde_zelle(name: str) -> Zelle | None:   # kann None liefern
    ...
```

- `-> None`: die Funktion gibt keinen Wert zurück – trotzdem hinschreiben
- `Zelle | None`: der senkrechte Strich heißt **„oder“** – der Rückgabewert ist eine `Zelle` **oder** `None`, der Aufrufer muss mit beidem rechnen

CadQuery-Typen

```python
from cadquery import Shape, Solid, Face, Edge, Wire, Vector

def stapel_bauen(p: ModulParam) -> Shape:
    ...

def endplatten(p: ModulParam) -> Shape:
    ...
```

Typ	Bedeutung
`Shape`	Oberbegriff für alle Formen – wenn der genaue Typ offen ist (z. B. nach Boolean)
`Solid` / `Face` / `Edge` / `Wire`	die konkreten Topologie-Elemente aus Einheit 2

Einheiten kann kein Typ ausdrücken – **aussagekräftige Namen** sind der Ersatz: `kapazitaet_ah`, `plattenstaerke`, `winkel_grad`.

Den Typ verengen: `assert isinstance`

Das Analyse-Tool weiß bei `plate - kanal` nur, dass **irgendeine** `Shape` herauskommt – gebraucht wird aber ein `Solid`. Drei Wege, ein guter:

```python
ergebnis = plate - kanal                # Typ: Shape

cast(Solid, ergebnis)                   # behauptet es, prüft nichts
ergebnis  # type: ignore                # blendet die Warnung nur aus

assert isinstance(ergebnis, Solid)      # prüft es wirklich – und verengt den Typ
```

`assert bedingung` bricht mit Fehler ab, wenn die Bedingung falsch ist. Nur diese Zeile scheitert **an Ort und Stelle**, falls das Boolean einmal etwas anderes liefert – die anderen zwei schweigen.

> Führt eine KI ein `# type: ignore` ein, das Sie nicht verstehen: nachfragen.

Praktikum A: den Modul-Code annotieren

Aufgabe 1: Signaturen ergänzen

Annotieren Sie Ihre Funktionen aus den letzten Wochen vollständig – Parameter **und** Rückgabetyp.

1. `zelle_bauen`, `stapel_bauen`, `endplatten`.
2. Wo kann eine Funktion `None` liefern? Machen Sie es mit `| None` sichtbar.

Hinweise: `-> Shape` für Geometrie-Funktionen, `-> None` wenn nichts zurückkommt, `float` / `int` für Maße

Aufgabe 2: mypy laufen lassen und aufräumen

1. Rufen Sie eine Funktion mit falschem Typ auf: `stapel_bauen("12")`. Was zeigt Pylance?
2. Führen Sie auf der Kommandozeile aus und **beheben Sie jede gemeldete Zeile**, bis mypy nichts mehr findet:

```
pip install mypy
mypy w07/ --ignore-missing-imports
```

3. Wo Boolean ein `Shape` liefert, aber ein `Solid` gebraucht wird: mit `assert isinstance` sauber verengen.

Hinweise: typische Meldungen sind fehlende Rückgabetypen und unbehandeltes `| None`

Theorie B: Unittests

Was ein Unittest ist

Ein **Unittest** ist eine gewöhnliche Python-Funktion. Sie ruft eine einzelne Funktion Ihres Codes auf und prüft das Ergebnis mit dem schon bekannten `assert`:

```
import pytest
from w07.zelle import zelle_bauen, ModulParam
```

```python
def test_zelle_volumen():
    zelle = zelle_bauen(ModulParam())
    erwartet = 148 * 98 * 11                        # Quader-Näherung
    assert zelle.Volume() == pytest.approx(erwartet, rel=0.01)
```

Ist die Bedingung falsch, wirft `assert` einen `AssertionError` – genau daran erkennt der Testrunner einen fehlgeschlagenen Test.

pytest: der Testrunner

pytest ist ein Programm, das Ihre Tests **findet und ausführt** – man ruft es ohne Argumente im Projektordner auf:

```bash
pip install pytest
pytest -v
```

```
tests/test_w07.py::test_zelle_volumen   PASSED
tests/test_w07.py::test_stapel_hoehe    FAILED
```

Gefunden wird nach zwei Konventionen:

- Datei heißt `test_*.py`
- Funktion darin heißt `test_*`

Bei einem Fehlschlag zeigt pytest die gescheiterte Zeile mit beiden Werten – Ist und Soll.

Warum? Regression

```
* 2e9d54a  Fillet-Radius auf 20 erhöht          ← test_zelle_volumen scheitert
* c04f81a  Stapel parametrisch gemacht
* f3a17bc  Erste Version der Zelle
```

Regression = etwas war korrekt und ist es nach einer Änderung nicht mehr.

Ohne Tests fällt es beim falschen Bauteil auf. Mit Tests schlägt `pytest` beim nächsten Commit an.

Aufbau: Arrange – Act – Assert

```python
def test_stapel_hoehe():
    p = ModulParam()                            # Arrange
    stapel = stapel_bauen(p)                    # Act
    hoehe = stapel.BoundingBox().zlen
    soll = (p.n_zellen - 1) * (p.zell_t + p.spacer) + p.zell_t
    assert hoehe == pytest.approx(soll, abs=0.1)   # Assert
```

Der Sollwert kommt aus einer **unabhängigen** Rechnung – nicht aus derselben Funktion (sonst wandert derselbe Denkfehler in beide).

`pytest.approx` – Fließkomma

```
assert zelle.Volume() == 159204.1          # spröde – Rundung bricht es
assert zelle.Volume() == pytest.approx(159204, rel=1e-3)   # 0,1 %
assert zelle.Volume() == pytest.approx(159204, abs=1.0)    # ±1 mm³
```

Geometrieoperationen sammeln Rundungsfehler ($0.1 + 0.2 \neq 0.3$) – exakte Gleichheit scheitert zufällig.

Was am CAD-Code prüfen?

```
zelle = zelle_bauen(ModulParam())

assert zelle.isValid()                                  # Kernel-Konsistenz
assert zelle.Volume() == pytest.approx(148*98*11, rel=0.01)
bb = zelle.BoundingBox()
assert bb.xlen == pytest.approx(148.0, abs=0.1)         # Außenmaß
assert len(zelle.Faces()) == 10                         # Konstruktionsabsicht
```

Existenz, Maße, analytisches Volumen, Flächenzahl – jede prüft eine andere Art von Fehler.

`isValid()`: notwendig, nicht hinreichend

Erinnern Sie sich an den `cap`-Stolperstein aus Einheit 6:

```
schale = cf.loft([w1, w2], cap=False)
assert schale.isValid()        # True – und trotzdem falsch!
```

`isValid()` prüft die **topologische Konsistenz**, nicht den Sinn:

- **prüft:** Kanten/Flächen/Vertices korrekt verbunden
- **prüft nicht:** ob das Volumen stimmt, ob ein Boolean leer wurde

⟶ Immer **zusätzlich** `Volume()` und Maße prüfen.

Antipatterns

- **Kein Assert:** `print(...)` statt `assert` – der Test „grünt“ immer, beweist nichts
- **Reimplementierung:** Sollwert aus derselben Formel wie die Funktion – derselbe Denkfehler in beiden
- **Zu viel auf einmal:** eine Testfunktion = ein Verhalten

- **Interne Details testen:** prüft *wie* statt *was* – bricht bei jedem Refactoring

CI: bei jedem Push (GitLab)

Continuous Integration: Auf dem GitLab-Server steht ein Rechner bereit, der nach jedem Push Ihr Projekt frisch auscheckt und die Kommandos aus der `.gitlab-ci.yml` in Ihrem Repo abarbeitet:

```yaml
image: .../ci-image:1        # fertige Umgebung (cadquery, pytest, mypy)

tests:
  script:
    - pytest tests/ -q
```

Diese Datei liegt seit Woche 1 im Repo – die Pipeline lief still mit. Ab heute ist sie Ihr Werkzeug: grünes ✓ oder rotes ✗ direkt am Commit.

Type-Check als zweiter Job

Type Hints sind eingeführt – jetzt kann `mypy` in der CI mitlaufen:

```yaml
typen:
  script:
    - mypy w*/ --ignore-missing-imports
  allow_failure: true        # Hinweis, kein Blocker
```

`allow_failure` macht den Job zum **Hinweisgeber**: Typfehler erscheinen gelb am Commit, blockieren die Pipeline aber nicht. So stört er nicht, solange Ihre Annotationen noch lückenhaft sind.

Praktikum B: das Modul testen

Aufgabe 3: Geometrie-Tests

Schreiben Sie `tests/test_w07.py` mit Tests für:

1. Gültigkeit und Volumen der Zelle.
2. Stapelhöhe gegen den **selbst gerechneten** Sollwert (bei Standardparametern 148,5 mm).
3. Ein Grenzfall: `ModulParam(n_zellen=1)` – bleibt alles gültig?

Hinweise: Dateiname und Testfunktionen beginnen mit `test_`; `.isValid()`, `.Volume()`, `.BoundingBox().zlen`, `pytest.approx(..., abs=0.1)`

Aufgabe 4: Kollision und Massenbudget

1. Testen Sie, dass benachbarte Zellen einander **nicht** durchdringen – ihr Schnittvolumen muss 0 sein.
2. **Massenbudget:** Schreiben Sie `strukturmasse(p)` (Volumen × Dichte über alle Strukturteile) und einen Test, der den Strukturanteil begrenzt: $m_{\text{Struktur}}/m_{\text{gesamt}} < 0,35$ (typisch ≈ 0,30 – der Test fängt grobe Ausreißer).

Hinweise: `&` bildet den Schnitt, `.Volume()`, `pytest.approx(0.0, abs=0.1)`; Zellmasse aus dem Datenblatt (≈ 0,86 kg je Zelle), Stahldichte $7,85 \cdot 10^{-6}$ kg/mm³

Aufgabe 5: CI und die Regression selbst auslösen

1. Committen und pushen Sie `tests/test_w07.py`. Läuft die Pipeline grün?
2. Erhöhen Sie den Fillet-Radius der Zelle auf **10 mm**. Der Test bleibt grün – warum? (Blick auf `rel=0.01`)
3. Erhöhen Sie ihn auf **20 mm**. Jetzt wird er rot. Passen Sie entweder den Sollwert oder die Toleranz begründet an.

Abschluss

Leseauftrag & Ausblick

- **Leseauftrag:** Buch Kapitel 8
- **KI-Werkzeuge:** getypter, getesteter Code macht KI-Assistenten zuverlässiger – Material zum Setup liegt auf der Kursseite (Selbststudium)
- **Wer mehr will:** einen Test *zuerst* schreiben (Red ⟶ Green), dann die Funktion dazu
- **Nächste Woche:** Datenaustausch – Assembly, BOM und STEP-Export des ganzen Moduls
- Bis dahin: `w07/` (Tests + Type Hints) committet und gepusht