# LU07e - Sicher refactoren

Refactoring ohne Sicherheitsnetz ist Raten. Diese Seite zeigt, wie Sie belegen, dass das Verhalten gleich geblieben ist - und welche Umbauten es still und leise doch verändern.

Für das Portfolio verlangt **D1I** genau das: Vorher/Nachher aus Ihrer Commit-Historie, *gleiches Verhalten per Test belegt*. Ohne Test ist der Nachweis unvollständig.

## Das Sicherheitsnetz: Tests

Ein Test hält fest, was der Code *heute* tut. Nach dem Umbau muss dasselbe herauskommen.

```python
# rabatt.py
def rabatt(kunde, betrag):
    if kunde is None:
        return 0.0
    if not kunde["aktiv"]:
        return 0.0
    if betrag < 100:
        return 0.0
    return round(betrag * 0.1, 2)
```

```python
# test_rabatt.py
import pytest
from rabatt import rabatt

AKTIV = {"aktiv": True}
GESPERRT = {"aktiv": False}

@pytest.mark.parametrize("kunde, betrag, erwartet", [
    (None,     200,   0.0),
    (GESPERRT, 200,   0.0),
    (AKTIV,     99.99, 0.0),
    (AKTIV,    100,   10.0),
    (AKTIV,    250,   25.0),
])
def test_rabatt(kunde, betrag, erwartet):
    assert rabatt(kunde, betrag) == erwartet
```

```
$ pytest -q
.....                                                                [100%]
5 passed in 0.01s
```

Beachten Sie die Auswahl der Fälle: nicht fünf beliebige Zahlen, sondern **die Grenzen** (99.99 und 100) und **jeder Zweig** der Funktion. Genau dort geht beim Umbau etwas kaputt.

# Code ohne Tests: Characterization Test

In bestehendem Code gibt es meist keine Tests - und Sie wissen nicht, ob das aktuelle Verhalten überhaupt richtig ist. Trotzdem können Sie es festhalten. Ein **Characterization Test** behauptet nicht, dass das Ergebnis *korrekt* ist, sondern nur, dass es *gleich bleibt*.

Vorgehen:

1. Funktion mit realistischen Eingaben aufrufen und die Ausgabe protokollieren.
2. Diese Ausgabe als erwarteten Wert in den Test schreiben, auch wenn sie seltsam aussieht.
3. Test laufen lassen: grün.
4. Refactoren. Bleibt der Test grün, hat sich nichts verändert.

```
def test_bestandsbericht_bleibt_gleich():
    # festgehalten am 09.09.2026, vor dem Refactoring
    assert bestandsbericht(BEISPIELDATEN) == "Total: 215.0 CHF (3
Positionen)"
```

Findet der Test später einen echten Fehler im alten Verhalten: **erst refactoren, dann in einem separaten Commit den Bug beheben** und den erwarteten Wert im Test anpassen. So bleibt nachvollziehbar, welche Änderung was bewirkt hat.

# Fünf Umbauten, die das Verhalten still verändern

Diese Fälle sehen wie Refactorings aus, sind aber keine. Sie gehören zu **D1A** («unerwünschte Nebeneffekte vermeiden»).

## 1. sorted() durch sort() ersetzt

```
namen = ["Zoe", "Ali", "Bea"]

sortiert = sorted(namen)   # -> ['Ali', 'Bea', 'Zoe'],  namen unverändert
sortiert = namen.sort()    # -> None,                    namen ist jetzt
sortiert
```

`sort()` verändert die Liste und gibt `None` zurück. Wer die Zeile «vereinfacht», bekommt einen `None`-Fehler an ganz anderer Stelle - oder schlimmer: gar keinen Fehler, aber eine veränderte Originalliste.

## 2. Liste durch Generator ersetzt

```
werte = (n * n for n in range(4))
print(sum(werte))   # 14
print(sum(werte))   # 0   <- der Generator ist aufgebraucht
```

Generatoren sparen Speicher, aber sie sind **einmal** durchlaufbar. Wo das Ergebnis zweimal gebraucht wird (einmal für die Summe, einmal für die Anzeige), muss es eine Liste bleiben.

### 3. Mutable Default-Wert eingeführt

```
def add(x, ziel=[]):
    ziel.append(x)
    return ziel

print(add(1))   # [1]
print(add(2))   # [1, 2]   <- nicht [2]
```

Der Standardwert wird **einmal** beim Definieren der Funktion ausgewertet, nicht bei jedem Aufruf (siehe die Animation in LU02e). Richtig ist `ziel=None` mit `ziel = [] if ziel is None else ziel`.

### 4. Query und Modifier zusammengelegt

```
def entnehmen(bestand, artikel):
    bestand[artikel] -= 1
    return bestand[artikel]

lager = {"maus": 3}
print(entnehmen(lager, "maus"))   # 2
print(entnehmen(lager, "maus"))   # 1
```

Derselbe Aufruf, verschiedene Ergebnisse. Wer diese Funktion beim Refactoring an eine zweite Stelle kopiert, um «nur kurz nachzuschauen», verändert den Bestand.

### 5. Geteilte Referenz statt Kopie

```
def preisliste_kopieren(original):
    return original          # keine Kopie, dieselbe Liste
    # return list(original)  # flache Kopie
```

Beide Varianten laufen fehlerfrei durch. Der Unterschied zeigt sich erst, wenn jemand die «Kopie» verändert.

## Ablauf, der Sie schützt

1. **Vorher committen.** Ein sauberer Stand, zu dem Sie zurückkönnen.
2. **Tests schreiben oder prüfen.** Grün, bevor Sie irgendetwas anfassen.
3. **Ein Refactoring, ein Commit.** Commit-Message mit Technik: `refactor: Extract Function gesamtsumme()`.
4. **Tests nach jedem Schritt.** Rot heisst: sofort zurück, nicht «schnell noch fixen».

5. **Diff lesen.** `git diff` vor dem Commit - sehen Sie nur, was Sie ändern wollten?

**Diese Commit-Historie ist Ihr Portfolio-Nachweis.** Ein Screenshot von `git diff` mit dem zugehörigen grünen Testlauf belegt D1I vollständig. Nachträglich lässt sich das nicht rekonstruieren.

## Werkzeuge

| Werkzeug | Wofür |
|---|---|
| **PyCharm-Refactorings** | Rename `Shift+F6`, Extract Method `Ctrl+Alt+M`, Extract Variable `Ctrl+Alt+V` - erfasst alle Aufrufstellen |
| **pytest** | Sicherheitsnetz, `pytest -q` nach jedem Schritt |
| **ruff** oder **flake8** | findet ungenutzte Variablen, tote Importe, zu komplexe Funktionen |
| **black** | einheitliche Formatierung, damit der Diff nur echte Änderungen zeigt |
| **git** | kleine Commits, `git diff`, Rückweg zum letzten grünen Stand |

---

M323-LU07, M323-D1I, M323-D1A