LU01.L11 - Spaghetticode entwirren

1. Analyse des Ausgangscodes

Zeile / Konstrukt	Grundstruktur	Bemerkung
`i = 0`, `fertig = False`, `total = 0`	Sequenz	Drei Zustandsvariablen, die von Hand gepflegt werden müssen.
`while not fertig`	Iteration	Endlosschleife mit Abbruchflagge statt echter Laufbedingung.
`if i >= len(bestellungen)`	Selektion	Wird nur gebraucht, weil die Schleife nicht über die Elemente läuft.
`if b["status"] == "storniert"` und `if b["menge"] <= 0`	Selektion	Sachlich sinnvoll, aber unnötig verschachtelt.
`continue`	**keine der drei**	Imitiert einen Sprung an den Schleifenanfang – genau das, was die strukturierte Programmierung vermeiden will.
`i = i + 1` an drei Stellen	Sequenz	Klassische Fehlerquelle: Eine vergessene Stelle erzeugt eine Endlosschleife.
`fertig = True`	Sequenz	Flaggenvariable als Ersatz für eine Schleifenbedingung.

2. Musterlösung

main.py

main.py

```
"""LU01.L11 - Musterloesung: Spaghetticode entwirren."""

from referenz import BESTELLUNGEN


def ist_verrechenbar(bestellung):
    """
    Entscheidet, ob eine Bestellung zum Umsatz zaehlt.

    :param bestellung: dict mit artikel, status, menge, preis
    :return: True oder False
    """
    return bestellung['status'] != 'storniert' and bestellung['menge']
> 0


def umsatz(bestellungen):
    """
```

```
    Summiert Menge mal Preis ueber alle verrechenbaren Bestellungen.

    :param bestellungen: Liste von Bestellungen
    :return: Umsatz als Zahl, bei leerer Liste 0
    """
    total = 0
    for bestellung in bestellungen:
        if ist_verrechenbar(bestellung):
            total += bestellung['menge'] * bestellung['preis']
    return total


def main():
    """Gibt den Umsatz der Beispielbestellungen aus."""
    print(f'Umsatz neu: {umsatz(BESTELLUNGEN):.2f}')


if __name__ == '__main__':
    main()
```

Übrig bleiben genau die drei Grundstrukturen: Sequenz (Initialisierung und Rückgabe), Iteration (`for`), Selektion (`if`).

3. Was durch den Umbau verschwunden ist

- **Der Index `i`.** Damit auch jede Möglichkeit, ihn falsch zu initialisieren, an einer Stelle nicht hochzuzählen (Endlosschleife!) oder um eins danebenzuliegen.
- **Die Flagge `fertig`.** Die `for`-Schleife kennt ihr Ende selbst.
- **Die Sprünge.** Ohne `continue` liest sich der Ablauf von oben nach unten.
- **Zwei Verschachtelungsebenen.** Die beiden Bedingungen sind zu einer benannten Frage zusammengefasst, die man laut vorlesen kann: *Wenn die Bestellung verrechenbar ist, zähle sie dazu.*

4. Wie geprüft wird

Die Gleichwertigkeit prüft `test_umsatz_wie_referenz` gegen `umsatz_alt` aus `referenz.py` – über vier Datensätze, inklusive leerer Liste, komplett stornierter Liste und ungültiger Mengen.

Die *Struktur* lässt sich mit einem gewöhnlichen Test nicht messen: Beide Fassungen liefern 362.70. Darum lesen zwei Tests die Datei `main.py` als Syntaxbaum ein:

main_test.py

main_test.py

```
"""Tests zu LU01.A11 - Spaghetticode entwirren."""

import ast
import pathlib

import pytest

import main
import referenz

ALLE_STORNIERT = [
    {'artikel': 'Maus', 'status': 'storniert', 'menge': 5, 'preis':
10.0},
    {'artikel': 'Dock', 'status': 'storniert', 'menge': 1, 'preis':
99.0},
]

MENGE_UNGUELTIG = [
    {'artikel': 'Maus', 'status': 'offen', 'menge': 0, 'preis': 10.0},
    {'artikel': 'Dock', 'status': 'offen', 'menge': -2, 'preis': 99.0},
]


def syntaxbaum():
    """Liest main.py als Syntaxbaum ein."""
    pfad = pathlib.Path(__file__).with_name('main.py')
    return ast.parse(pfad.read_text(encoding='utf-8'))


def knoten(*typen):
    """Sammelt alle Knoten der angegebenen Typen aus main.py."""
    return [k for k in ast.walk(syntaxbaum()) if isinstance(k, typen)]


def test_umsatz_beispiel():
    assert main.umsatz(referenz.BESTELLUNGEN) == pytest.approx(362.70)


def test_umsatz_leer():
    assert main.umsatz([]) == 0


def test_umsatz_alle_storniert():
    assert main.umsatz(referenz.BESTELLUNGEN) == pytest.approx(362.70)
    assert main.umsatz(ALLE_STORNIERT) == 0


def test_umsatz_menge_ungueltig():
    assert main.umsatz(referenz.BESTELLUNGEN) == pytest.approx(362.70)
```

```
    assert main.umsatz(MENGE_UNGUELTIG) == 0


def test_umsatz_wie_referenz():
    for daten in (referenz.BESTELLUNGEN, [], ALLE_STORNIERT,
MENGE_UNGUELTIG):
        assert main.umsatz(daten) ==
pytest.approx(referenz.umsatz_alt(daten))


def test_ist_verrechenbar():
    assert main.ist_verrechenbar({'status': 'offen', 'menge': 3,
'preis': 1.0}) is True
    assert main.ist_verrechenbar({'status': 'geliefert', 'menge': 1,
'preis': 1.0}) is True
    assert main.ist_verrechenbar({'status': 'storniert', 'menge': 3,
'preis': 1.0}) is False
    assert main.ist_verrechenbar({'status': 'offen', 'menge': 0,
'preis': 1.0}) is False


def test_iteration_ohne_while():
    assert main.umsatz(referenz.BESTELLUNGEN) == pytest.approx(362.70)
    assert knoten(ast.For), 'main.py enthaelt keine for-Schleife'
    assert not knoten(ast.While), 'main.py enthaelt noch eine while-
Schleife'


def test_keine_spruenge():
    assert main.umsatz(referenz.BESTELLUNGEN) == pytest.approx(362.70)
    assert not knoten(ast.Continue, ast.Break), 'main.py enthaelt noch
continue oder break'
```

`test_iteration_ohne_while` verlangt eine `for`-Schleife und verbietet `while`, `test_keine_spruenge` verbietet `continue` und `break`. Beide prüfen zusätzlich das korrekte Ergebnis – sonst wären sie mit einer leeren Datei erfüllbar.

Eine Abgabe, die richtig rechnet, aber die alte Struktur behält, kommt damit auf 9 von 11 Testpunkten.

5. Antworten auf die Zusatzfragen

Frage 1: Vergessenes Hochzählen

Entfernt man beispielsweise das `i = i + 1` im Storno-Zweig, bleibt die Schleife für immer bei derselben stornierten Bestellung stehen – eine **Endlosschleife**. Das Programm hängt ohne

Fehlermeldung. In der neuen Fassung ist dieser Fehler gar nicht mehr möglich, weil die Iteration nicht mehr von Hand gesteuert wird.

Frage 2: Böhm und Jacopini

Der Ausgangscode brauchte scheinbar Sprünge (`continue`), um einen Fall zu überspringen. Die neue Fassung zeigt: Dieselbe Wirkung entsteht durch **Selektion** – statt zu überspringen, wird nur im passenden Fall etwas getan. Das ist genau die Aussage des Theorems: Jeder berechenbare Algorithmus lässt sich allein mit Sequenz, Selektion und Iteration ausdrücken. Sprünge sind bequem, aber nie zwingend nötig.

Frage 3: Deklarative Formulierung

Der Umsatz ist die Summe aus Menge mal Preis über alle nicht stornierten Bestellungen mit positiver Menge.

In Python direkt so ausdrückbar:

```
def umsatz_deklarativ(bestellungen):
    return sum(b['menge'] * b['preis'] for b in bestellungen if
ist_verrechenbar(b))
```

Weder Zwischensumme noch Schleifenkörper sind noch sichtbar – der Code liest sich wie der Satz darüber. Die Bausteine dahinter (`map`, `filter`, `reduce`, Comprehensions) behandeln wir ausführlich in LU04.
