LU07.L10 - Refactoring-Review

Paar	Urteil	Was passiert
1	**Verhaltensänderung**	`namen.sort()` gibt `None` zurück und sortiert die übergebene Liste an Ort und Stelle. Der Aufrufer bekommt `None` statt einer Liste - und seine Originalliste ist verändert
2	**Verhaltensänderung**	Der Generator ist nach `sum` aufgebraucht. `max` bekommt eine leere Folge und wirft `ValueError`
3	**Verhaltensänderung**	Der mutable Standardwert wird einmal beim Definieren erzeugt. Der zweite Aufruf sieht die Werte des ersten
4	**Echtes Refactoring**	Gleiche Liste, gleiche Reihenfolge, gleiche Objekte
5	**Kommt darauf an**	Siehe unten
6	**Echtes Refactoring** mit Nebengewinn	Gleiches Ergebnis. `any` hört beim ersten Treffer auf und baut keine Zwischenliste

Die Tests, die es aufgedeckt hätten

```
def test_paar1_original_bleibt():
    namen = ["Zoe", "Ali"]
    assert sortierte_namen(namen) == ["Ali", "Zoe"]
    assert namen == ["Zoe", "Ali"]           # scheitert nachher

def test_paar2_zweimal_lesbar():
    werte = quadrate(1000)
    assert sum(werte) > 0
    assert max(werte) > 0                     # scheitert nachher mit
ValueError

def test_paar3_frischer_start():
    assert sammeln(1) == [1]
    assert sammeln(2) == [2]                  # scheitert nachher: [1, 2]
```

Alle drei Tests haben dieselbe Bauart: Sie prüfen nicht nur den Rückgabewert, sondern auch **was daneben passiert** - der Zustand der Eingabe, die zweite Verwendung, der zweite Aufruf. Genau dort verstecken sich unerwünschte Nebeneffekte.

Zu Paar 5

Die Index-Fassung ist ein echtes Refactoring **nur unter zwei Bedingungen**:

1. **`kunde_id` existiert immer.** Die Vorher-Version lässt Bestellungen ohne passenden Kunden still weg, die Nachher-Version wirft `KeyError`.
2. **`id` ist eindeutig.** Bei doppelten IDs liefert die Vorher-Version mehrere Zeilen pro Bestellung, die Nachher-Version nur eine.

```
bestellungen = [{"kunde_id": 1, "betrag": 10}, {"kunde_id": 9, "betrag":
```

```
20}]
kunden = [{"id": 1, "name": "A"}, {"id": 1, "name": "A2"}]

# vorher -> [('A', 10), ('A2', 10)]      Bestellung 9 fällt still weg
# nachher -> KeyError: 9
```

Beide Fassungen sind vertretbar - aber es sind **verschiedene** Fachregeln. Wer den Umbau macht, muss entscheiden, welche gelten soll, und das dokumentieren:

```
def zuordnen(bestellungen, kunden):
    index = {k["id"]: k for k in kunden}
    # bewusst: Bestellungen ohne passenden Kunden werden übersprungen (wie
bisher)
    return [(index[b["kunde_id"]]["name"], b["betrag"])
            for b in bestellungen if b["kunde_id"] in index]
```

Das ist der Kern von D1A. Ein Refactoring wird nicht dadurch richtig, dass die Tests grün sind - sondern dadurch, dass Sie wissen, welche Fälle die Tests *nicht* abdecken. Fehlende Schlüssel, Duplikate, leere Eingaben und mehrfache Verwendung sind die vier Stellen, an denen es fast immer passiert.

M323-LU07, M323-D1A