LU07d - Funktionales Refactoring

Die Techniken aus LU07c gelten in jeder Sprache und jedem Paradigma. Diese Seite behandelt die drei Umbauten, die für **dieses Modul** zentral sind: von der Schleife zur Pipeline, von unpure zu pure, von mutable zu immutable.

Das Ziel: Nicht «funktional, weil funktional». Sondern: Code, dessen Ergebnis nur von den Eingaben abhängt, lässt sich lesen, ohne den Rest des Programms zu kennen - und testen, ohne ihn vorher in einen bestimmten Zustand zu bringen.

1. Replace Loop with Pipeline

Sehr viele Schleifen tun in Wahrheit nur eines von drei Dingen: **filtern**, **umformen** oder **zusammenfassen**. Genau dafür gibt es Comprehensions und `map`/`filter`/`reduce` (LU04).

Umformen

```
# vorher
namen = []
for e in eintraege:
    namen.append(e["name"].upper())

# nachher
namen = [e["name"].upper() for e in eintraege]
```

Filtern

```
# vorher
aktive = []
for e in eintraege:
    if e["aktiv"]:
        aktive.append(e)

# nachher
aktive = [e for e in eintraege if e["aktiv"]]
```

Zusammenfassen

```
# vorher
total = 0
for e in eintraege:
    total += e["betrag"]
```

```
# nachher
total = sum(e["betrag"] for e in eintraege)
```

Der Unterschied ist nicht die Zeilenzahl. Die Vorher-Version sagt *wie* gerechnet wird (Zähler anlegen, durchlaufen, addieren), die Nachher-Version sagt *was* herauskommen soll. Das ist der Schritt von imperativ zu deklarativ aus LU01a.

Der Umbau Schritt für Schritt

Die Animation zeigt denselben Umbau in fünf Schritten. Achten Sie auf die Konsole: Sie zeigt bei jedem Schritt dasselbe Ergebnis - genau das macht den Umbau zu einem Refactoring. Der zweite Tab zeigt drei Kriterien dafür, wann die Schleife besser stehen bleibt.

Durchgehendes Beispiel

Eine typische Auswertung, wie sie in einem Flask-Projekt vorkommt: Umsatz aller Zubehör-Positionen.

```
verkaeufe = [
    {"artikel": "Maus",     "kategorie": "Zubehoer", "menge": 3, "preis":
25.0},
    {"artikel": "Tastatur", "kategorie": "Zubehoer", "menge": 1, "preis":
80.0},
    {"artikel": "Laptop",   "kategorie": "Geraet",   "menge": 2, "preis":
1200.0},
    {"artikel": "USB-Hub",  "kategorie": "Zubehoer", "menge": 5, "preis":
12.0},
    {"artikel": "Monitor",  "kategorie": "Geraet",   "menge": 1, "preis":
210.0},
]
```

Vorher - eine Schleife, die filtert, rechnet und aufsummiert:

```
def umsatz_zubehoer(daten):
    total = 0.0
    for eintrag in daten:
        if eintrag["kategorie"] == "Zubehoer":
            total = total + eintrag["menge"] * eintrag["preis"]
    return round(total, 2)
```

Nachher - die drei Aufgaben sind getrennt und einzeln benannt:

```
def ist_zubehoer(eintrag):
    return eintrag["kategorie"] == "Zubehoer"
```

```
def zeilenumsatz(eintrag):
    return eintrag["menge"] * eintrag["preis"]

def umsatz_zubehoer(daten):
    return round(sum(zeilenumsatz(e) for e in daten if ist_zubehoer(e)), 2)
```

```
umsatz_zubehoer(verkaeufe)  ->  215.0   (beide Versionen)
```

`ist_zubehoer` und `zeilenumsatz` sind jetzt wiederverwendbar - für die Sortierung, den Filter im Frontend, die Statistikseite. Aus einer Auswertung sind Bausteine geworden.

Mit `filter` und `reduce` geschrieben, liefert dieselbe Logik denselben Wert:

```
from functools import reduce

round(reduce(lambda a, e: a + zeilenumsatz(e),
             filter(ist_zubehoer, verkaeufe),
             0.0), 2)   # -> 215.0
```

Welche der drei Fassungen nehmen? In Python ist die Comprehension mit `sum()` die lesbarste und im Zweifel die richtige Wahl. `map`/`filter`/`reduce` zeigen Sie im Portfolio dort, wo die Kompetenz **C2** das verlangt - nicht überall.

2. Unpure wird pure

Auslöser: globaler Zustand, `global`-Schlüsselwort, Funktion verändert ihre Argumente.

```
# vorher - unpure
warenkorb = []

def hinzufuegen(artikel):
    global warenkorb
    warenkorb.append(artikel)
    print(f"{artikel} hinzugefügt")
```

Diese Funktion ist nicht testbar, ohne vorher `warenkorb` zu präparieren, und sie tut drei Dinge gleichzeitig: rechnen, Zustand ändern, ausgeben.

```
# nachher - funktionaler Kern, dünne Schale
def mit_artikel(korb, artikel):          # pure: neue Liste, kein
Seiteneffekt
    return korb + [artikel]

def hinzufuegen_und_melden(korb, artikel):   # Schale: hier ist der
Seiteneffekt
    print(f"{artikel} hinzugefügt")
    return mit_artikel(korb, artikel)
```

Das Muster heisst **funktionaler Kern, imperative Schale**: Die Berechnung ist pure und getestet, Ein- und Ausgabe (`print`, Datenbank, HTTP-Response) leben am Rand. In Flask heisst das konkret: Die Route nimmt Request-Daten entgegen und gibt eine Response zurück, die Berechnung dazwischen steckt in pure Funktionen, die man ohne laufenden Server testen kann.

3. Mutable wird immutable

Auslöser: eine Funktion verändert eine Liste oder ein Dict, das sie als Argument bekommen hat.

```
# vorher - die Eingabe wird verändert
def preise_erhoehen(produkte, prozent):
    for p in produkte:
        p["preis"] = round(p["preis"] * (1 + prozent / 100), 2)
    return produkte

# nachher - neue Daten, Original bleibt
def preise_erhoehen(produkte, prozent):
    return [
        {**p, "preis": round(p["preis"] * (1 + prozent / 100), 2)}
        for p in produkte
    ]
```

Die Vorher-Version ist der Klassiker aus LU02d: Weil Listen und Dicts by reference übergeben werden, sieht der Aufrufer die Änderung, ob er will oder nicht. Wer die Preisliste danach noch im Originalzustand braucht, hat ein Problem, das erst Wochen später auffällt.

Wann Sie es lassen sollten

Nicht jede Schleife wird als Pipeline besser. Lassen Sie die Schleife stehen, wenn:

- die Comprehension mehr als eine Bedingung und eine Umformung braucht und über zwei Zeilen wächst,
- in der Schleife etwas passiert, das kein Wert ist (Datei schreiben, Mail versenden, Log-Eintrag),
- ein vorzeitiger Abbruch nötig ist (`break`) - dafür gibt es `next()`, `any()` und `all()`, aber nicht jede Variante ist lesbarer,
- mehrere Zwischenergebnisse voneinander abhängen.

Eine dreifach verschachtelte Comprehension ist kein funktionaler Fortschritt, sondern ein neuer Smell.

Checkliste für den Umbau

1. Was tut die Schleife wirklich - filtern, umformen, zusammenfassen oder mehreres davon?

2. Bei mehrerem: zuerst **Split Loop**, dann jeden Teil einzeln umbauen.
3. Bedingung und Umformung je in eine benannte Funktion (`ist_zubehoer`, `zeilenumsatz`).
4. Pipeline schreiben, Tests laufen lassen, Ergebnis mit der alten Version vergleichen.
5. Prüfen: Verändert die neue Fassung noch irgendetwas ausserhalb ihrer selbst? Wenn nein, ist sie pure.

M323-LU07, M323-D1I

© Kevin Maurizi

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/modul/m323/learningunits/lu07/funktionalesrefactoring

Last update: **2026/09/09 13:51**