LU07c - Refactoring-Techniken

Jede Technik hat einen Namen, einen Auslöser (den Smell) und ein festes Vorgehen. Der Name ist wichtig: Er macht aus «ich habe da mal aufgeräumt» eine Aussage, die man im Team und im Portfolio begründen kann.

Alle Beispiele auf dieser Seite sind so aufgebaut, dass die Version *nachher* exakt dasselbe liefert wie *vorher*.

1. Extract Function

Auslöser: lange Funktion, Abschnittskommentare, duplizierter Code.

```
# vorher
def rechnung_drucken(positionen):
    total = 0
    for p in positionen:
        total += p["menge"] * p["einzelpreis"]
    print("Rechnung")
    print("--------")
    print(f"Total: {round(total, 2)} CHF")
```

```
# nachher
def gesamtsumme(positionen):
    return round(sum(p["menge"] * p["einzelpreis"] for p in positionen), 2)

def rechnung_drucken(positionen):
    print("Rechnung")
    print("--------")
    print(f"Total: {gesamtsumme(positionen)} CHF")
```

Der Gewinn ist nicht die kürzere Funktion, sondern dass `gesamtsumme` jetzt **pure** und damit einzeln testbar ist. Die Ausgabe bleibt der einzige Seiteneffekt und steht an einer Stelle.

2. Rename

Auslöser: Namen wie `d`, `tmp`, `process()`, `data2`.

```
# vorher
def calc(l, x):
    return [i * x for i in l]

# nachher
def preise_mit_faktor(preise, faktor):
```

```
    return [preis * faktor for preis in preise]
```

Rename ist das am meisten unterschätzte Refactoring. Nutzen Sie dafür die IDE (in PyCharm `Shift+F6`), nie Suchen-und-Ersetzen: Die IDE benennt nur die tatsächlichen Vorkommen um, nicht zufällig gleich heissende Strings.

3. Extract Variable

Auslöser: Bedingung oder Ausdruck, den man zweimal lesen muss.

```
# vorher
if kunde["alter"] >= 18 and kunde["status"] == 1 and bestellwert > 100:
    ...

# nachher
volljaehrig = kunde["alter"] >= 18
aktiv = kunde["status"] == 1
grossbestellung = bestellwert > 100

if volljaehrig and aktiv and grossbestellung:
    ...
```

Der Variablenname ersetzt den Kommentar. Bei mehrfacher Verwendung wird daraus besser gleich eine Funktion (`ist_aktiv(kunde)`).

4. Replace Magic Number with Named Constant

Auslöser: nackte Zahlen im Code.

```
# vorher
def brutto(netto):
    return round(netto * 1.081, 2)

# nachher
MWST_SATZ_PROZENT = 8.1

def brutto(netto):
    return round(netto * (1 + MWST_SATZ_PROZENT / 100), 2)
```

```
brutto(100)  ->  108.1
```

Wenn der Satz ändert, ändert genau eine Zeile. Und man sieht am Namen, ob 1.081 der Mehrwertsteuersatz oder ein Zuschlag war.

5. Guard Clauses

Auslöser: tief verschachtelte Bedingungen.

```
# vorher
def rabatt(kunde, betrag):
    if kunde is not None:
        if kunde["aktiv"]:
            if betrag >= 100:
                return betrag * 0.1
            else:
                return 0.0
        else:
            return 0.0
    else:
        return 0.0
```

```
# nachher
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 betrag * 0.1
```

Die Sonderfälle stehen oben und sind schnell abgehakt, der Normalfall steht unten auf Ebene eins. Beide Versionen liefern für alle vier Fälle identische Werte - genau das prüft man nach dem Umbau mit Tests.

6. Parameterize Function

Auslöser: zwei fast identische Funktionen.

```
# vorher
def rabatt_10(betrag):
    return betrag * 0.9

def rabatt_20(betrag):
    return betrag * 0.8

# nachher
def mit_rabatt(betrag, prozent):
    return betrag * (1 - prozent / 100)
```

Wenn die Aufrufstellen dadurch unleserlich werden, hilft ein Closure als Funktionsfabrik (LU05c): `rabatt_10 = lambda b: mit_rabatt(b, 10)`.

7. Introduce Parameter Object

Auslöser: lange Parameterliste mit zusammengehörenden Werten.

```
# vorher
def zeilenpreis(artikel, menge, einzelpreis, rabatt_prozent):
    ...

# nachher
from dataclasses import dataclass

@dataclass(frozen=True)
class Position:
    artikel: str
    menge: int
    einzelpreis: float
    rabatt_prozent: float = 0.0

def zeilenpreis(pos: Position) -> float:
    brutto = pos.menge * pos.einzelpreis
    return round(brutto * (1 - pos.rabatt_prozent / 100), 2)
```

```
zeilenpreis(Position("Maus", 3, 25.0, 10))  ->  67.5
```

`frozen=True` macht das Objekt unveränderlich - die Funktion kann die Eingabe nicht versehentlich mutieren (siehe LU02e).

8. Replace Temp with Query

Auslöser: temporäre Variable, die nur ein Zwischenergebnis hält.

```
# vorher
def bericht(positionen):
    total = sum(p["menge"] * p["einzelpreis"] for p in positionen)
    if total > 1000:
        return f"Grossauftrag: {total}"
    return f"Auftrag: {total}"

# nachher
def gesamtsumme(positionen):
    return sum(p["menge"] * p["einzelpreis"] for p in positionen)

def bericht(positionen):
    if gesamtsumme(positionen) > 1000:
        return f"Grossauftrag: {gesamtsumme(positionen)}"
    return f"Auftrag: {gesamtsumme(positionen)}"
```

Abwägung: Die Nachher-Version rechnet mehrfach. Bei einer teuren Berechnung ist die temporäre Variable die bessere Wahl - oder Sie kombinieren beides mit `@cache` (LU07g). Genau solche Abwägungen sind auf Niveau **D1A** gefragt.

9. Separate Query from Modifier

Auslöser: eine Funktion liefert einen Wert und verändert dabei etwas.

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

# nachher
def restbestand(bestand, artikel):          # nur lesen
    return bestand[artikel]

def nach_entnahme(bestand, artikel):        # neuer Zustand, ohne Mutation
    return {**bestand, artikel: bestand[artikel] - 1}
```

Die Vorher-Version ist doppelt gefährlich: Zwei Aufrufe hintereinander liefern verschiedene Werte, und wer nur nachschauen wollte, hat das Lager verändert.

10. Split Loop

Auslöser: eine Schleife erledigt zwei unabhängige Aufgaben.

```
# vorher
summe = 0
namen = []
for e in eintraege:
    summe += e["betrag"]
    namen.append(e["name"])

# nachher
summe = sum(e["betrag"] for e in eintraege)
namen = [e["name"] for e in eintraege]
```

Zwei Durchläufe statt einem - das ist bei üblichen Datenmengen irrelevant und macht jeden Teil einzeln verständlich und wiederverwendbar. Erst wenn eine Messung zeigt, dass es weh tut, dreht man das zurück (LU07f).

Auswahl in der Praxis

Reihenfolge, die fast immer funktioniert: zuerst umbenennen, dann Konstanten

benennen, dann Guard Clauses, dann extrahieren. Wenn die Namen stimmen, sieht man erst, welche Teile überhaupt zusammengehören.

Die IDE nimmt Ihnen die mechanische Arbeit ab. In PyCharm: `Shift+F6` (Rename), `Ctrl+Alt+M` (Extract Method), `Ctrl+Alt+V` (Extract Variable), `Ctrl+Alt+C` (Extract Constant). Ein Werkzeug-Refactoring vergisst keine Aufrufstelle - ein manuelles schon.

M323-LU07, M323-D1B, M323-D1I