LU01a - Deklarative vs. Imperative Programmierung

Einstieg: Taxi oder Navi?

Stellen Sie sich vor, Sie sitzen in einem Taxi. Es gibt zwei Arten, ans Ziel zu kommen:

- **„Fahren Sie 300 Meter geradeaus, dann links, an der Ampel rechts, nach dem Kreisel die zweite Ausfahrt …„**
- **„Zum Hauptbahnhof, bitte.“**

Beide bringen Sie ans Ziel. Im ersten Fall bestimmen *Sie* jeden Schritt – Sie müssen den Weg kennen und tragen die Verantwortung für jeden Abzweiger. Im zweiten Fall beschreiben Sie nur das *Ziel* – wie der Fahrer dorthin kommt, ist seine Sache.

Genau das ist der Unterschied zwischen **imperativer** und **deklarativer** Programmierung. Der erste Stil beschreibt das **Wie**, der zweite das **Was**.

Diskussionsfrage für den Einstieg

Wann ist welche Variante besser? Sammeln Sie Argumente, bevor Sie weiterlesen.

Mögliche Antworten: Bei einer Baustelle auf der Strecke will man vielleicht selbst steuern (Kontrolle). Kennt man den Weg nicht, ist die Zielangabe klar überlegen (Abstraktion). Der Taxifahrer kennt zudem Abkürzungen, auf die man selbst nie käme (Optimierung durch das System).

Aktivität: Der menschliche Roboter

Diese Übung macht den Unterschied körperlich erfahrbar.

Aufbau: Eine Person spielt den Roboter, eine Person gibt die Befehle. Ziel: Der Roboter soll einen Stift vom Pult holen und der Kommandogeberin überreichen.

Runde	Erlaubte Sprache	Beobachtung
1 – Imperativ	Nur Einzelbefehle: `SCHRITT VOR`, `DREHE LINKS 90°`, `ARM HEBEN`, `GREIFEN`, `ARM SENKEN`, `LOSLASSEN`. Kein Nennen von Objekten oder Zielen.	Es braucht viele Befehle. Ein Zählfehler („noch ein Schritt„) führt zum Absturz. Die Kommandogeberin muss den Zustand des Roboters ständig im Kopf mitführen.

Runde	Erlaubte Sprache	Beobachtung
2 – Deklarativ	Nur ein einziger Satz, der das Ergebnis beschreibt: `Gib mir den roten Stift vom Pult.`	Ein Befehl genügt. Der Roboter (also das „System“) entscheidet selbst über Weg, Reihenfolge und Griff. Dafür gibt die Kommandogeberin die Kontrolle über das Wie ab.

Auswertung im Plenum:

1. In welcher Runde war der Befehl kürzer? *(Runde 2)*
2. In welcher Runde wussten Sie genau, was passieren wird? *(Runde 1)*
3. Was passiert in Runde 2, wenn zwei rote Stifte auf dem Pult liegen? *(Das System muss entscheiden – Mehrdeutigkeit ist der Preis der Abstraktion.)*
4. Welche Runde war anfälliger für Fehler? *(Runde 1 – mehr Einzelschritte, mehr Fehlerquellen.)*

Imperative Programmierung

Imperative Programmierung wird oft als die traditionellste Art der Programmierung betrachtet. Sie ist eine Art Anweisungsliste oder Rezept, das dem Computer sagt, was er tun soll.

1. **Schritt-für-Schritt-Ansatz**: Der Programmierer gibt eine genaue Sequenz von Anweisungen, die in einer bestimmten Reihenfolge ausgeführt werden sollen.
2. **Statusverwaltung**: Der Zustand des Programms wird häufig durch Variablen verändert, die Werte speichern und ändern.
3. **Kontrollstrukturen**: Schleifen und bedingte Anweisungen werden verwendet, um die Kontrolle über den Programmablauf zu steuern.
4. **Beispiel**: Ein Sortieralgorithmus wie Bubble Sort, der jeden Schritt im Detail beschreibt.

Deklarative Programmierung

Deklarative Programmierung ist eher eine Art, dem Computer zu sagen, was er erreichen soll, ohne zu beschreiben, wie er es tun soll.

1. **Was, nicht Wie**: Der Schwerpunkt liegt auf dem, was erreicht werden soll, nicht auf dem, wie es erreicht werden soll.
2. **Ausführung durch das System**: Wie das Ergebnis konkret berechnet wird, entscheidet die Sprache bzw. die Laufzeitumgebung – zum Beispiel der Query-Optimizer einer Datenbank.
3. **Höhere Abstraktion**: Oft wird ein höheres Abstraktionsniveau verwendet, das komplexere Operationen in einfacheren Ausdrücken darstellt.
4. **Beispiel**: Eine SQL-Abfrage, die beschreibt, was man von einer Datenbank möchte, ohne die Schritte zur Erlangung dieser Informationen zu beschreiben.

Das durchgehende Beispiel: Die Notenliste

Wir lösen ein und dieselbe Aufgabe in beiden Stilen. Diese Aufgabenstellung begleitet uns durch das

ganze Kapitel.

Aufgabe: Aus einer Klassenliste sollen die Namen aller Lernenden mit einer Note von 5.0 oder besser ermittelt werden – in Grossbuchstaben.

```
noten = [("Alina", 5.5), ("Ben", 3.5), ("Chiara", 4.0), ("Dario", 5.0),
("Elif", 4.5)]
```

Variante 1: Imperativ

```
bestanden = []                              # 1. Leeres Gefäss vorbereiten
for eintrag in noten:                       # 2. Jedes Element einzeln ansteuern
    name = eintrag[0]                       # 3. Bestandteile herauslösen
    note = eintrag[1]
    if note >= 5.0:                         # 4. Bedingung prüfen
        bestanden.append(name.upper())      # 5. Zustand verändern

print(bestanden)                            # ['ALINA', 'DARIO']
```

Fünf gedankliche Schritte, und der Zustand von `bestanden` ändert sich während des Durchlaufs mehrfach.

Live-Demo: Zeichnen Sie die Liste `bestanden` an die Tafel und lassen Sie die Klasse nach jedem Schleifendurchgang den neuen Inhalt diktieren. Genau das ist ein *Trace Table* – siehe LU01d.

Variante 2: Deklarativ (List Comprehension)

```
bestanden = [name.upper() for name, note in noten if note >= 5.0]

print(bestanden)                            # ['ALINA', 'DARIO']
```

Gelesen wird das als Satz: *„Nimm den Namen in Grossbuchstaben, für jeden Namen und jede Note in der Notenliste, sofern die Note mindestens 5.0 ist.„*

Variante 3: Deklarativ (map und filter)

```
bestanden = list(map(lambda p: p[0].upper(),
                     filter(lambda p: p[1] >= 5.0, noten)))

print(bestanden)                            # ['ALINA', 'DARIO']
```

Diese Bausteine – `map`, `filter` und `reduce` – sind das Herzstück des Moduls und werden später ausführlich behandelt.

Variante 4: Dieselbe Frage als SQL

```sql
SELECT UPPER(name) FROM noten WHERE note >= 5.0;
```

Auffällig: Die SQL-Abfrage und die List Comprehension sind fast identisch aufgebaut – *Was auswählen*, *woher*, *unter welcher Bedingung*. Deklarativ ist also keine Erfindung der funktionalen Programmierung, sondern ein Prinzip, das Sie längst kennen.

Die Schlüsselfrage: Was ist verschwunden?

Stellen Sie der Klasse diese Frage und vergleichen Sie Variante 1 mit Variante 2:

In Variante 1 vorhanden	**In Variante 2 verschwunden, weil …**
Die leere Liste `bestanden = []`	… kein Zwischenzustand mehr aufgebaut werden muss.
Die Schleife `for`	… die Iteration im Sprachkonstrukt steckt.
Der Index-Zugriff `eintrag[0]`	… direkt entpackt wird.
Das append	… nichts mehr verändert, sondern ein neuer Wert erzeugt wird.
Die Reihenfolge der fünf Schritte	… nur noch das Ergebnis beschrieben wird.

Merksatz für die Wandtafel: *Imperativ baut ein Ergebnis Schritt für Schritt auf. Deklarativ beschreibt das Ergebnis.*

Warum das relevant ist: der Klassiker-Bug

Dieses Beispiel eignet sich hervorragend, um den Preis der Zustandsverwaltung zu zeigen. Lassen Sie den Code laufen, *bevor* Sie ihn erklären:

```python
def bestanden_imperativ(liste, ergebnis=[]):
    for name, note in liste:
        if note >= 5.0:
            ergebnis.append(name)
    return ergebnis

print(bestanden_imperativ(noten))   # ['Alina', 'Dario']
print(bestanden_imperativ(noten))   # ['Alina', 'Dario', 'Alina', 'Dario']
← ?!
```

Derselbe Aufruf, dasselbe Argument – zwei verschiedene Ergebnisse. Der Grund: Die Liste `ergebnis` wird in Python nur *einmal* beim Definieren der Funktion angelegt und bleibt zwischen den Aufrufen bestehen. Sie ist veränderlicher Zustand, der die Funktion überlebt.

Genau solche Fehler will die funktionale Programmierung mit **Unveränderlichkeit** und **puren Funktionen** ausschliessen – siehe LU01c und LU02.

Alltagsbeispiele zum Durchspielen

Diese Paare eignen sich gut zum Sammeln an der Wandtafel. Lassen Sie die Klasse jeweils raten, welche Spalte welchem Paradigma entspricht.

Situation	**Imperativ (das Wie)**	**Deklarativ (das Was)**
Kaffee	„Mahle 18 g Bohnen, verteile sie gleichmässig, presse mit 30 kg an, brühe 25 Sekunden …“	„Einen Espresso, bitte.„
Möbel	Die IKEA-Anleitung mit 34 nummerierten Schritten	„Stell mir ein Bett ins Zimmer.“
Bildbearbeitung	Über jedes Pixel iterieren und den Helligkeitswert um 20 erhöhen	Schieberegler „Helligkeit +20 %„
Webseite	Mit Canvas Linien, Rechtecke und Text an Koordinaten zeichnen	`<h1>Titel</h1>` – der Browser entscheidet über Schriftgrösse und Position
Wäsche	„Fülle Wasser ein, heize auf 40 °C, drehe die Trommel 12 Minuten …“	Programm „Buntwäsche 40°„ wählen
Daten	Über alle Datensätze schleifen und passende sammeln	`SELECT … WHERE …`

Erkennungsmerkmale: Der Spickzettel

Woran erkennt man den Stil beim Blick auf fremden Code?

Merkmal	**Deutet auf**
`for` / `while` mit Zählvariable	imperativ
Eine Variable wird mehrfach überschrieben (`summe += x`)	imperativ
`append`, `insert`, `sort()`, `del` – die Daten selbst werden verändert	imperativ
List Comprehension, `map`, `filter`, `sum`, `sorted()`	deklarativ
Es wird ein *neuer* Wert erzeugt statt ein bestehender geändert	deklarativ
Der Code liest sich wie ein Satz über das Ergebnis	deklarativ

Ein guter Test: **Kann man den Code laut vorlesen und es klingt wie eine Beschreibung des Ergebnisses – oder wie eine Anleitung?**

Die häufigste Fehlvorstellung

Achtung: deklarativ heisst nicht automatisch seiteneffektfrei

Seiteneffektfreiheit ist keine Eigenschaft der deklarativen Programmierung an sich, sondern der *puren funktionalen* Programmierung – also einer Teilmenge davon.

SQL ist deklarativ, aber ein `UPDATE kunden SET status = 'aktiv';` verändert sehr wohl den Zustand der Datenbank. Und HTML ist deklarativ, kennt aber gar keine Funktionen.

Richtig ist: *Deklarativ* beschreibt das Was statt das Wie. *Pur funktional* verlangt zusätzlich, dass eine Funktion nichts ausserhalb ihrer selbst verändert. Mehr dazu in LU01c und in LU02 (Pure Functions).

Vergleich

- **Kontrolle vs. Abstraktion**: Während die imperative Programmierung eine präzise Kontrolle über das „Wie“ der Problemlösung bietet, ermöglicht die deklarative Programmierung eine höhere Abstraktion über das „Was„.
- **Lesbarkeit**: Deklarative Programme sind oft kürzer und leichter zu verstehen, während imperative Programme detaillierter sind.
- **Effizienz**: Imperative Programme können oft effizienter sein, da der Programmierer die genaue Kontrolle hat. Deklarative Programme sind möglicherweise weniger effizient, aber schneller zu schreiben und zu warten.
- **Fehleranfälligkeit**: Der detaillierte Charakter der imperativen Programmierung kann fehleranfälliger sein, da mehr Möglichkeiten für menschliche Fehler bestehen.

Schlussfolgerung

Während die imperative Programmierung den Programmierern die vollständige Kontrolle über den Prozess gibt, bietet die deklarative Programmierung eine elegante und oft intuitivere Methode zur Lösung von Problemen. Die Wahl zwischen diesen Stilen hängt oft von der Art des Problems, den Anforderungen an die Effizienz und die Vorlieben des Programmierers ab. In der modernen Softwareentwicklung werden oft Elemente beider Stile kombiniert, um eine effiziente und wartbare Lösung zu schaffen.

M323-LU01, M323-AG1, M323-BG1

© Kevin Maurizi

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

Permanent link:
https://wiki.bzz.ch/modul/m323/learningunits/lu01/deklarativimperativ?rev=1787034424

Last update: **2026/08/18 08:27**