Dokumentation — Modul 323 Projekt

Wahl der imperativen Programmiersprache

- Programmiersprache: Java 17
- Funktionale Elemente in Java:
  - Lambda Expressions: anonyme Funktionen, z. B. `x -> x + 1`
  - Streams API: `filter`, `map`, `flatMap`, `collect`, `reduce` etc.
  - Method References: `String::toLowerCase`
  - Optionals: `Optional<T>` als Null-sicheres Container-Konzept
  - Immutability durch `final`-Variablen und unveränderliche Klassen (konzeptionell)

Beispiel (Quelle: eigene Implementierung):

```
List<Integer> xs = List.of(1,2,3,4);
int sumSquares = xs.stream()
    .map(x -> x * x)
    .reduce(0, Integer::sum);
```

Projektantrag (Kurzfassung)

- Projekttitel: Analyse von Fussballdaten – Spieler- und Mannschaftsstatistiken
- Ausgangslage: Öffentliche Fussballdaten enthalten viele Dimensionen (Spiele, Teams, Tore, Karten). Ziel ist es, strukturierte Statistiken abzuleiten.
- Datengrundlage: Beispiel-CSV-Dateien (`matches.csv`, `players.csv`). Erweiterbar durch öffentliche Quellen wie opendata.swiss, football-data.co.uk, Kaggle.
- Produktfunktionen:
  - Filter: Spiele nach Liga/Saison
  - Map: Tordifferenz je Spiel
  - Reduce: Gesamttore einer Mannschaft, Durchschnittstore pro Spiel
  - Vergleich: Heim-/Auswärtssiege/Unentschieden in Prozent
  - Top-Scorer: Top-Spieler aus Spielerdaten
- Technologien: Java, Konsolenoutput; Funktionale Elemente: Streams, Lambdas

Datenstruktur

- `v*/data/matches.csv`:
  - `date,season,league,home_team,away_team,home_goals,away_goals,home_yellow_cards,away_yellow_cards,home_possession,away_possession`
- `v*/data/players.csv`:
  - `player,team,league,season,goals`

Output (Beispiel)

- Top-Scorer (Liga=Super League, Saison=2024)
- Durchschnittstore pro Spiel (alle Spiele im Datensatz)
- Mannschaft mit den meisten Gesamttoren (z. B. FC Basel, Saison=2024)
- Filter Super League 2024 (Liste der Spiele)
- Tordifferenz je Spiel
- Verteilung: Heimsiege vs. Auswärtssiege vs. Unentschieden

Die Versionen V1 (imperativ) und V2 (funktional) liefern denselben Output.

Fazit

- Nutzen funktionaler Elemente:
  - Kürzerer, ausdrucksstärkerer Code für Daten-Pipelines (filter/map/reduce)
  - Weniger mutable State, geringere Fehleranfälligkeit bei Aggregationen
  - Leichter kombinierbar und umformulierbar (andere Filter/Sortierungen)
- Refactoring-Ergebnis:
  - Vereinfachung der Implementierung in V2 durch Streams und Collector-APIs
  - Lesbarkeit steigt bei komplexeren Aggregationen (z. B. Gruppierungen)
- Wiederverwendung:
  - Ja, insbesondere für analytische Tasks und Datenverarbeitung in Pipelines
  - Im Betrieb: Reports, Auswertung von Logs/Events

Ausführen

Siehe `README.md`.