SWE3 - Übung 5

WS 2025/26

Aufwand in h: 6

Tim Peko

Inhaltsverzeichnis

1. Aufgabe: Flugreisen

1.1. Lösungsidee

Die Domäne wird über drei Klassen modelliert:

- `Person`: Vor- und Nachname, `Gender` (enum), Alter, Adresse, Kreditkarten-Nummer. Invarianten: Namen/Adresse nicht leer, Alter in `[0, 130]`, Kreditkarte nur Ziffern und Luhn-valide. Ausgabe maskiert die Karte (nur letzte 4 Ziffern sichtbar).
- `Flug`: Flugnummer, Fluglinie, `origin`/`destination`, Abflug-/Ankunftszeit (als Strings für Portabilität), Flugdauer in Minuten (> 0). Invarianten: alle Strings nicht leer, `origin ≠ destination`.
- `Flugreise`: Reisende Person und Sequenz von `Flug`-Segmenten. Invariante: Itinerary ist verbunden (`destination[i] == origin[i+1]`). Aggregationen: gesamte Flugzeit (Summe) sowie Zeitfenster (erste Abflug-/letzte Ankunftszeit).

Die Zeiten werden bewusst als Strings gehalten, um Date/Time-Abhängigkeiten (Zeitzonen/Locale/C++20) zu vermeiden; Validierung bezieht sich daher auf Nicht-Leere und Konsistenz der Orte. Für die Übungen genügt das und erlaubt portable Tests.

1.2. Designentscheidungen

- **Kapselung**: Alle Klassen bieten lesende Getter, Validierung erfolgt in den Konstruktoren (Fehler → `std::invalid_argument`).
- **Einfaches Zeitmodell**: Zeiten als Strings, Dauer als `int` Minuten. Dadurch einfache und robuste Ausgabe, ohne Plattformdetails.
- **Formatierte Ausgaben**: `operator<<` für `Person`, `Flug`, `Flugreise` erzeugen kompakte, menschenlesbare Zeilen.
- **Sicheres Logging**: Kreditkarten werden via `maskedCreditCard()` nur mit letzten vier Ziffern ausgegeben.
- **Domänensprache**: Die Klassen sind nach der Domäne benannt, um die Lesbarkeit zu verbessern. Die Domäne wird in deutscher Sprache benannt und spiegelt sich auch so im Code wieder.

1.3. Validierung & Fehlerbehandlung

- `Person` prüft Zwangsinvarianten inkl. Luhn-Check der Kreditkarte. Dazu existiert `static bool isValidCreditCard(...)`.
- `Flug` verifiziert Pflichtfelder, ungleiche Orte und positive Dauer.
- `Flugreise::addFlight(...)` erzwingt die Verbindung der Segmente (Exception bei Bruch).

Alle Validierungsfehler werden als `std::invalid_argument` signalisiert und in den Unit-Tests explizit geprüft.

1.4. Teststrategie

Die Tests folgen dem AAA-Prinzip (Arrange-Act-Assert).

- **Person**: Konstruktion mit valider Testnummer (`4111 1111 1111 1111`), Maskierung, Luhn-Validierung; Negativfall (falsche Karte).
  - Boundary Ages: $\text{age} \in [0, 130]$ valide; `-1`, `131` invalid.
  - Empty Fields: Leere Vor-/Nachnamen und Adresse werden abgewiesen.
  - Digits-Only Credit Card: Nummern mit Leerzeichen/Bindestrichen/Buchstaben → invalid.
  - Luhn Vektoren: Klassiker `79927398713` (valid) vs. `79927398714` (invalid); gängige Test-PANs (`4242...`, `4012...`) validieren.
  - Masking: Länge > 4 behält die letzten 4 Ziffern; Länge ≤ 4 vollständig maskiert.
  - Streaming: Ausgabe enthält korrekten Gender-String (Male/Female/Diverse).
- **Flug**: Konstruktion, Streaming, Negativfall bei `origin == destination`.
  - Pflichtfelder: Leere Felder (Nummer, Airline, Orte, Zeiten) werden abgewiesen.
  - Konnektivitätsregel (Segment): `origin != destination` wird erzwungen.
  - Dauer: Nicht-positive Dauer (0, negativ) wird abgewiesen.
  - Streaming: Ausgabe enthält Flugnummer, Route und Dauer in Minuten.
- **Flugreise**: Hinzufügen mehrerer Segmente (Linz → Frankfurt → Denver → Las Vegas), Aggregation von Minuten, Zeitfenster, Streaming; Negativfall (gebrochene Konnektivität).
  - Leere Reiseroute: `total_flight_time=0` und kein `window=` in der Ausgabe.
  - Einzelnes Segment: Korrektes Zeitfenster `firstDepartureTime()` → `lastArrivalTime()`.
  - Große Reiseroute: 50 verbundene Segmente; Summe der Minuten, erstes/letztes Zeitfenster korrekt.
  - Copy-Semantik: Streaming bleibt nach Kopie identisch; Invariante bleibt erfüllt.
  - Konnektivität pro Hinzufügen: Bruch der Verbindung wird exakt bei der fehlerhaften `addFlight`-Operation erkannt (Exception); vorherige Segmente bleiben erhalten.

Die `main.cpp` startet GoogleTest; alle Tests laufen automatisiert.

1.5. Ergebnisse

- **Datenmodell**: ist klar, portabel und robust gegen ungültige Eingaben.
- **Ausgaben**: sind kompakt und enthalten alle geforderten Informationen.
- **Testfälle**: decken Konstruktion, Validierung, Formatierung und Reiserouten-Konnektivität ab.

1.5.1. Testfälle

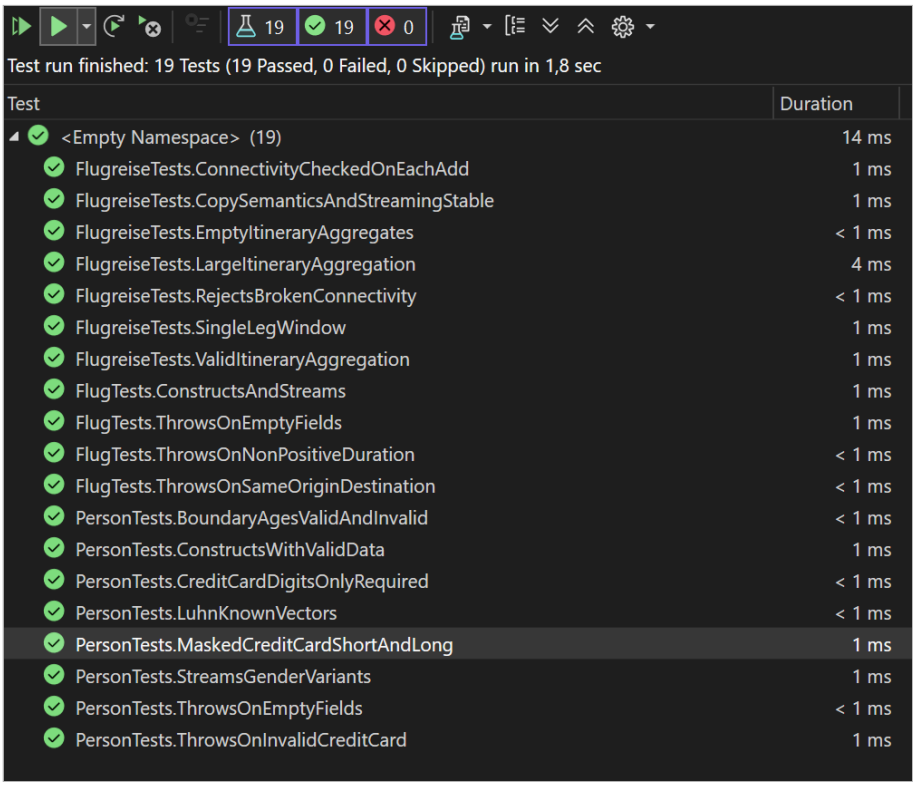

Abbildung 1: Testfälle der Aufgabe 1

1.5.2. Beispielausgabe

```txt
TRIP[
  PERSON[Jane Roe, Female, 28, Street 1, ************1111]
  FLIGHT[OS1, Austrian, Linz → Frankfurt, dep 08:00, arr 09:00, 60 min]
  FLIGHT[UA2, United, Frankfurt → Denver, dep 10:30, arr 18:00, 510 min]
  FLIGHT[WN3, Southwest, Denver → Las Vegas, dep 19:00, arr 20:30, 90 min]
  total_flight_time=660 min, window=08:00 → 20:30
]
```

2. Aufgabe 2: Stücklistenverwaltung (`./partlists`)

2.1. Lösungsidee

Die Aufgabe wird als klassisches Composite realisiert (`namespace PartLists`):

- `Part` (abstrakt): Basisklasse mit Namen, `equalsTo(...)`, `clone()`, `accept(...)` und einfacher Persistenz (`store/load`).
- `CompositePart`: Sammlung von `Part`-Kindern; besitzt und verwaltet Kind-Elemente. `equalsTo` vergleicht rekursiv Struktur und Namen.
- **Formatter** (Strategie/Visitor):
  - `HierarchyFormatter`: gibt die Hierarchie eingerückt aus.
  - `SetFormatter`: zählt alle Blätter (Stückliste als Multiset) in Einfügereihenfolge.
- `Storable` (Interface): definiert `store()` / `load()`; beide Methoden verwenden eine einfache, menschenlesbare Textrepräsentation.

Wesentliche Entwurfsdetails:

- **Klonen statt Kopieren**: `addPart(Part const&)` nutzt `clone()` zur Wahrung des dynamischen Typs (keine Slicing-Probleme).
- **Eigentum**: `CompositePart` besitzt Kinder via `std::unique_ptr<Part>`. `getParts()` liefert konstante Rohzeiger zur sicheren Iteration.
- **Persistenzformat**:
  - Blätter: `P|<name>`
  - Composite:

```
C|<name>
{
  ...
}
```

  `load()` verifiziert aktuell die Wurzel; ein vollständiger Rekonstruktionsparser wäre ein natürlicher Ausbau.

2.1.1. Formatter

- **HierarchyFormatter** (Preorder, 2 Leerzeichen pro Ebene):

```txt
Sitzgarnitur
  Sessel
    Bein (klein)
    Bein (klein)
    Bein (klein)
    Bein (klein)
    Sitzfläche
  Sessel
    Bein (klein)
    Bein (klein)
    Bein (klein)
    Bein (klein)
    Sitzfläche
  Tisch
    Bein (groß)
    Bein (groß)
    Bein (groß)
    Bein (groß)
    Tischfläche
```

- **SetFormatter** (Einfügereihenfolge der Blätter durch Traversierung):

```txt
Sitzgarnitur:
  8 Bein (klein)
  2 Sitzfläche
  4 Bein (groß)
  1 Tischfläche
```

2.2. Teststrategie

Die Tests folgen dem AAA-Prinzip und orientieren sich an den Best-Practices aus den vorigen Übungen. Zusätzlich wurde die Abdeckung deutlich erweitert:

- **Validierung**: Leere Namen werden abgewiesen (`Part`, `CompositePart`).
- **Strukturgleichheit**: `equalsTo` vergleicht Namen und Reihenfolge/Struktur der Kinder; unterschiedliche Reihenfolge → `false`.
- **Klonen**: Deep-Clones sind unabhängig; nach Mutation der Originalstruktur bleibt der Clone unverändert.
- **Formatter-Outputs**:
  - Hierarchie: exakter Stringvergleich inkl. Einrückung.
  - Set: deterministische Reihenfolge über erste Auftretensreihenfolge; exakte Zählwerte.
  - Leere Composite: nur Name bzw. Name mit Doppelpunkt (keine weiteren Zeilen).
- **Persistenz**:
  - Header-Prüfung: `P|<name>` für Blätter, `C|<name>` für Composite.
  - Negativtest: Composite lädt nicht aus `P|...` (Exception).
  - Sanity: `store()`/`load()` werfen keine Exceptions bei korrektem Format.
- **Tiefe Strukturen**: 20 Ebenen tiefer Baum wird korrekt formatiert (Einrückung sichtbar).

2.3. Ergebnisse

Alle definierten Testfälle sind, wie in Abbildung 2 dargestellt, erfolgreich.

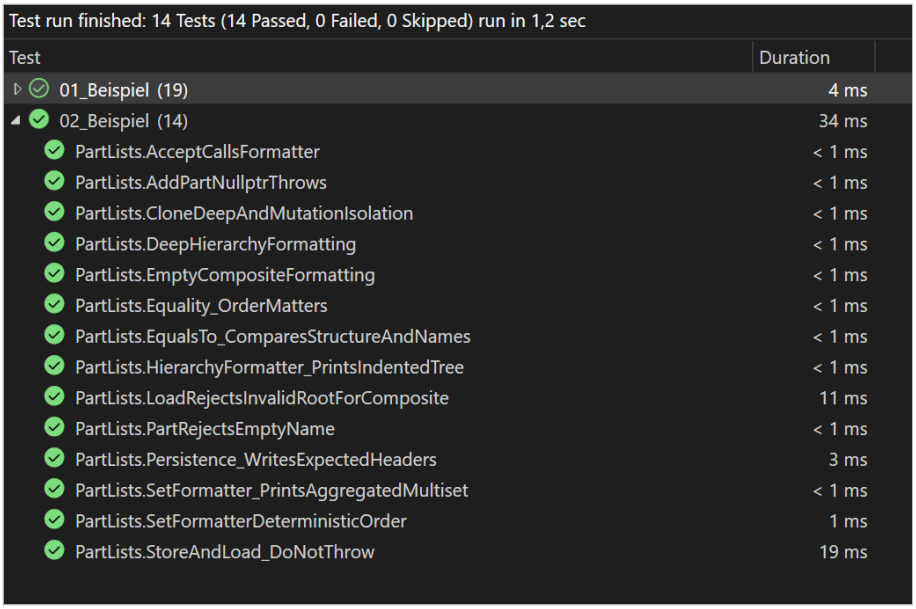

Abbildung 2: Testfälle der Aufgabe 2