SWE3 - Übung 4

WS 2025/26

Aufwand in h: 4

Tim Peko

1. Aufgabe 1: Klasse `rational_t` erweitern

In `01_Beispiel` wurde die bisherige, auf `int` spezialisierte Klasse durch einen generischen Datentyp `rational_t<T>` ersetzt und um die geforderten Funktionalitäten erweitert.

1.1. Lösungsidee

Die Klasse definiert minimale Anforderungen an `T` über ein Concept. Invarianten werden in `normalize()` sichergestellt (Vorzeichen am Zähler, $\frac{0}{x} \longrightarrow \frac{0}{1}$, Reduktion per Euklid). Vergleiche basieren auf Kreuzmultiplikation und sind damit unabhängig von der Kürzung robust. Division wird als Multiplikation mit dem Kehrwert umgesetzt, was Implementierung und Korrektheit vereinfacht. Operatoren sind als zweistellige `friend`-Funktionen (Barton-Nackman) realisiert.

1.2. Implementierung

- `rational_t<T>` ist vollständig als Template im Header `rational_t.hpp` implementiert (inline), inklusive:
  - `value_type`-Export (`using value_type = T`)
  - Methode `inverse()` (Kehrwert mit Fehler bei Null)
  - Normalisierung: Vorzeichen am Nenner wird nach oben gezogen, `0/x → 0/1`. Reduktion auf Normalform erfolgt über den euklidischen Algorithmus (mit `%`).
  - Vergleiche via Kreuzmultiplikation: $\left(\frac{a}{b} = \frac{c}{d}\right) \Longleftrightarrow ad = cb$ (unabhängig von Reduktion), $<, \leq, >, \geq$ entsprechend.
  - Division als Multiplikation mit dem Kehrwert (Nutzung von `inverse()`).
  - Operatoren als zweistellige `friend`-Funktionen (Barton–Nackman): `+,-,*,/` sowie $=, \neq, <, \leqslant, >, \geqslant$, außerdem `<<` / `>>`.
- Variante 2 (Concept): Ein Concept beschreibt die Minimalanforderungen an `T` (siehe unten). Es wird bewusst keine `ops`-Schicht verwendet.
- `matrix_t<T>`: zu Testzwecken als 1×1-Matrix umgesetzt (`matrix_t.hpp`). Die Operatoren sind als zweistellige `friend`-Funktionen inline implementiert. `matrix_t<T>::one()` und `::zero()` liefern Einheits- bzw. Nullmatrix. Streams verwenden die Form `[x]`.

1.3. Anforderungen an den Typ T

Damit `rational_t<T>` funktioniert, stellt die Implementierung möglichst geringe Anforderungen an `T`. Das Concept fordert:

- Konstruktion aus `int` (`T{0}`, `T{1}`)

- Arithmetik: `+`, `-`, `*`, `/`, `%`, unäres `-`
- Vergleich: `==`, `!=`, `<`
- Streams: `<<`, `>>`

Zusätzlich gilt:

- Die Reduktion erfolgt über den euklidischen Algorithmus (benötigt `%`).
- Gleichheit/Ordnung funktionieren korrekt über Kreuzmultiplikation, unabhängig von Reduktion.
- Für `matrix_t<T>` (1×1):
  - Alle Operatoren leiten auf den enthaltenen Skalar `T` weiter (inkl. `%`).
  - Damit kann `rational_t<matrix_t<T>>` Kehrwerte bilden, arithmetisch arbeiten und korrekt vergleichen.

Hinweis: Die Reduktion auf Normalform basiert auf `%` und dem euklidischen Algorithmus. Vergleiche werden unabhängig davon korrekt über Kreuzmultiplikation durchgeführt. Division ist als Multiplikation mit dem Kehrwert implementiert.

1.4. Tests

Es existiert ein gemeinsames Google-Test-Projekt (`test.cpp`) mit Testfällen pro Teilaspekt (nach Teilaufgaben gruppiert):

- Konstruktion, Normalisierung, Streams, Arithmetik, Vergleich, Kettenausdrücke (für `rational_t<int>`)
- Kehrwert (`inverse`) inkl. Fehlerfall bei Null; Division nutzt `inverse()`
- Tests für `rational_t<long>` (Generizität) und `rational_t<matrix_t<int>>` (1×1) inkl. Kehrwert und Arithmetik

Jeder Test enthält aussagekräftige Namen und prüft erwarteten gegen tatsächlichen Output (über `as_string()`/Vergleiche). Fehlersituationen (z. B. Nenner `0`, Invertieren von `0`) werden explizit getestet und dokumentiert. Matrizen-Streams verwenden `[x]`.

1.5. Ergebnisse

Die Ergebnisse der Testfälle, dargestellt in Abbildung 1, zeigen keine Fehler in der Implementierung nach.

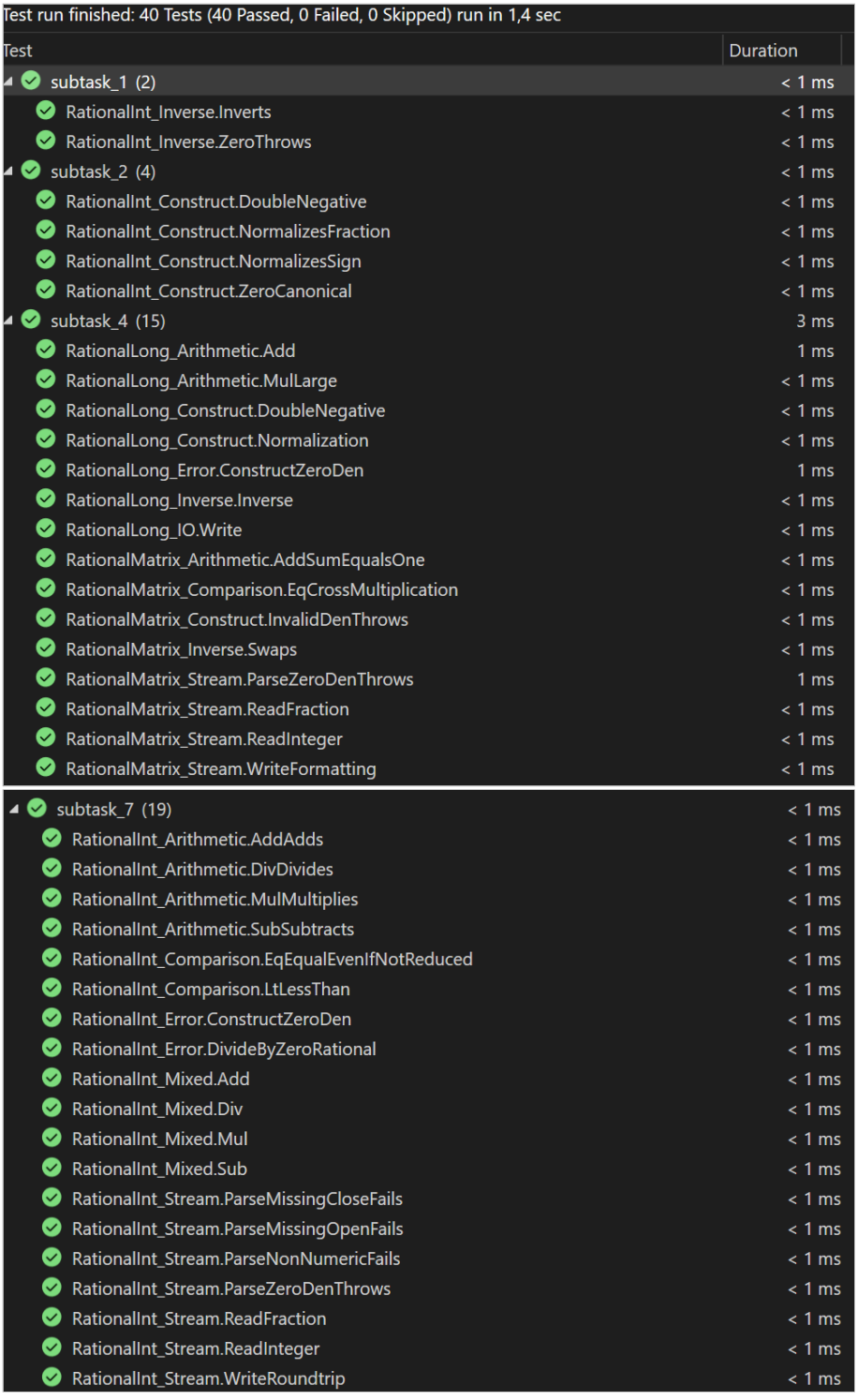

Abbildung 1: Ergebnisse der Testfälle