



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Kapitel 03

ÄQUIVALENZKLASSEN: INTELLIGENTE TESTSTRATEGIEN

Wo wir stehen

Was wir bisher gelernt haben:

-  Erschöpfendes Testen ist unmöglich (32 Milliarden Jahre!)
-  Wir brauchen intelligente Strategien zum effizienten Testen
-  Prinzipien für sauberen Testcode (AAA-Muster, beschreibende Namen)

Heutige Frage:

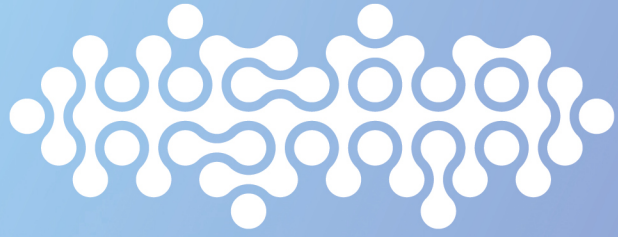
 *Wenn wir nicht alles testen können, **wie wählen wir aus, was wir testen?***

Das Kernproblem

```
def reciprocal(x: float) -> float:  
    """Returns 1/x"""  
    if x == 0:  
        raise ZeroDivisionError("Cannot divide by zero")  
    return 1.0 / x
```

Fragen:

- Wie viele mögliche Eingaben? **Unendlich!** (alle Floats)
- Wie viele Tests können wir schreiben? **Endlich** (vielleicht 5-10)
- Wie wählen wir aus, welche 5-10 Eingaben wir testen? 🤔



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Äquivalenzklassen: Die Grundidee

1

Die Antwort: Äquivalenzklassen

Zentrale Erkenntnis:

Nicht alle Eingaben sind gleich interessant. Viele Eingaben produzieren **ähnliches Verhalten**.

Definition (verständlich):

Eine **Äquivalenzklasse** ist eine Gruppe von Eingaben, die:

- Ähnliches Ausgabeverhalten produzieren
- Denselben Code-Pfad testen
- Dieselben mathematischen Eigenschaften haben

Strategie:

Teste **einen Repräsentanten** aus jeder Äquivalenzklasse statt alle möglichen Eingaben!

Schlüsseleigenschaften

1. Disjunkt (keine Überlappung)

Jede Eingabe gehört zu **genau einer** Äquivalenzklasse

2. Vollständig (alles abgedeckt)

Jede mögliche Eingabe gehört zu **irgendeiner** Äquivalenzklasse

3. Repräsentatives Testen

Das Testen **eines** Werts aus einer Klasse reicht, um die ganze Klasse zu testen

Ergebnis: 3 Repräsentanten testen statt ∞ Eingaben! 🎯

Mathematische Grundlage (Optionaler Exkurs)

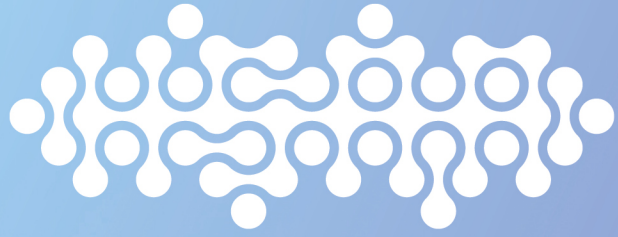
Eine **Äquivalenzrelation** $\sim$ auf einer Menge X erfüllt:

1. **Reflexiv:** $x \sim x$ (jedes Element ist zu sich selbst äquivalent)
2. **Symmetrisch:** Wenn $x \sim y$, dann $y \sim x$
3. **Transitiv:** Wenn $x \sim y$ und $y \sim z$, dann $x \sim z$

Beispiel: Für $\text{reciprocal}(x)$:

- Definiere: $x \sim y$ wenn $\text{sign}(x) = \text{sign}(y)$
- Dies erzeugt 3 Äquivalenzklassen: positiv, negativ, null

Keine Sorge, wenn das abstrakt wirkt - wir sehen gleich konkrete Beispiele!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Numerische Beispiele

2

Beispiel 1: reciprocal(x) - Der einfachste Fall

Interaktive Frage:

🤔 Wie viele **verschiedene Verhaltensweisen** siehst du in diesem Code?

Denk nach:

- Was passiert wenn `x > 0` ?
- Was passiert wenn `x < 0` ?
- Was passiert wenn `x == 0` ?

reciprocal(x): Drei Äquivalenzklassen

Positive Zahlen	<code>x > 0</code>	<code>x = 5.0</code>	Gibt positiven Kehrwert zurück (<code>0.2</code>)
Negative Zahlen	<code>x < 0</code>	<code>x = -5.0</code>	Gibt negativen Kehrwert zurück (<code>-0.2</code>)
Null	<code>x == 0</code>	<code>x = 0.0</code>	Wirft <code>ZeroDivisionError</code>

Zentrale Erkenntnis: Alle positiven Zahlen verhalten sich gleich (nur unterschiedliche Größenordnungen)!

reciprocal(x): Test-Code

```
def test_reciprocal_positive():  
    """Test mit positiver Zahl (Repräsentant: 5.0)"""  
    assert reciprocal(5.0) == 0.2  
  
def test_reciprocal_negative():  
    """Test mit negativer Zahl (Repräsentant: -5.0)"""  
    assert reciprocal(-5.0) == -0.2  
  
def test_reciprocal_zero():  
    """Test mit Null (Repräsentant: 0.0)"""  
    with pytest.raises(ZeroDivisionError):  
        reciprocal(0.0)
```

Ergebnis: 3 Tests statt ∞ Tests! 

Häufiger Fehler: Über-Testen

✗ Schlechter Ansatz (verschwenderisch):

```
def test_reciprocal_1(): assert reciprocal(1.0) == 1.0
def test_reciprocal_2(): assert reciprocal(2.0) == 0.5
def test_reciprocal_5(): assert reciprocal(5.0) == 0.2
def test_reciprocal_10(): assert reciprocal(10.0) == 0.1
def test_reciprocal_100(): assert reciprocal(100.0) == 0.01
# Alle testen die GLEICHE Äquivalenzklasse (positive Zahlen)!
```

✓ Guter Ansatz (effizient):

```
def test_reciprocal_positive():
    assert reciprocal(5.0) == 0.2 # Ein Repräsentant reicht!
```

Beispiel 2: Mehrere Parameter

```
def reciprocal_sum(x: float, y: float, z: float) -> float:
    """Returns 1 / (x + y + z)"""
    total = x + y + z
    if abs(total) < 1e-10:
        raise ZeroDivisionError("Sum is too close to zero")
    return 1.0 / total
```

Interaktive Frage:

🤔 *Wie viele Äquivalenzklassen hat diese Funktion?*

- *3 Parameter, jeder kann +/-/0 sein, also... $3^3 = 27$ Klassen?*

Die Überraschung: Immer noch nur 3 Klassen!

Zentrale Erkenntnis: Die **Summe** ist entscheidend, nicht die einzelnen Parameter!

Positive Summe	$x+y+z > 0$	(1.0, 2.0, 3.0)	Gibt positiven Wert zurück (0.167)
Negative Summe	$x+y+z < 0$	(-1.0, -2.0, -3.0)	Gibt negativen Wert zurück (-0.167)
Summe Null	$x+y+z \approx 0$	(1.0, -0.5, -0.5)	Wirft Fehler

Warum? Weil `reciprocal_sum(1,2,3)` und `reciprocal_sum(2,1,3)` das **gleiche Verhalten** produzieren (gleiche Summe).

Häufiger Fehler: Alle Vorzeichenkombinationen testen

✗ Schlechter Ansatz (verschwenderisch - 8 Tests):

```
test_all_positive()      # (+, +, +)
test_all_negative()     # (-, -, -)
test_pos_pos_neg()      # (+, +, -)
test_pos_neg_pos()      # (+, -, +)
test_neg_pos_pos()      # (-, +, +)
test_pos_neg_neg()      # (+, -, -)
test_neg_pos_neg()      # (-, +, -)
test_neg_neg_pos()      # (-, -, +)
```

Problem: Diese testen alle die **gleichen 3 Verhaltensweisen** (positive/negative/Null-Summe)!

Guter Ansatz: Summen-Kategorien testen

✓ Guter Ansatz (effizient - 3 Tests):

```
def test_positive_sum():
    """Repräsentant: (1, 2, 3) → Summe = 6"""
    assert reciprocal_sum(1.0, 2.0, 3.0) == pytest.approx(0.167, abs=0.01)

def test_negative_sum():
    """Repräsentant: (-1, -2, -3) → Summe = -6"""
    assert reciprocal_sum(-1.0, -2.0, -3.0) == pytest.approx(-0.167, abs=0.01)

def test_zero_sum():
    """Repräsentant: (1, -0.5, -0.5) → Summe ≈ 0"""
    with pytest.raises(ZeroDivisionError):
        reciprocal_sum(1.0, -0.5, -0.5)
```

Schlüsselprinzip: Ausgabe-Kategorien bestimmen Eingabe-Klassen

Falsches Denken:

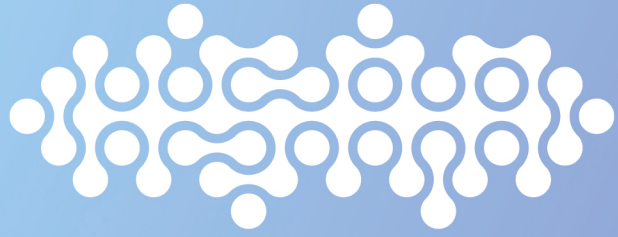
"Ich habe 3 Parameter → Ich brauche $3^3 = 27$ Tests"

Korrektes Denken:

"Was sind die verschiedenen **Ausgabe-Verhaltensweisen**?" → 3 Verhaltensweisen → 3 Tests

Allgemeines Prinzip:

1. Schaue auf den **Code** (White-Box-Testing)
2. Identifiziere **was die Ausgabe bestimmt**
3. Gruppiere Eingaben nach **ähnlichem Verhalten**
4. Wähle **einen Repräsentanten** pro Gruppe



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Granularität wählen

3

Ist die Partitionierung eindeutig?

Frage: Gibt es nur EINEN richtigen Weg, Eingaben zu partitionieren?

Antwort: Nein! **Du wählst** basierend auf deinen Zielen.

Beispiel für `reciprocal(x)` :

Option 1: Grob (2 Klassen)

- Nicht-Null (`x ≠ 0`) → `x = 5.0`
- Null (`x == 0`) → `x = 0.0`

Option 2: Fein (3 Klassen) ✨ (Am häufigsten)

- Positiv (`x > 0`) → `x = 5.0`
- Negativ (`x < 0`) → `x = -5.0`
- Null (`x == 0`) → `x = 0.0`

Granularität wählen (fortgesetzt)

Option 3: Sehr fein (5 Klassen)

- Groß positiv ($x > 1$) $\rightarrow x = 10.0$
- Klein positiv ($0 < x \leq 1$) $\rightarrow x = 0.1$
- Klein negativ ($-1 \leq x < 0$) $\rightarrow x = -0.1$
- Groß negativ ($x < -1$) $\rightarrow x = -10.0$
- Null ($x == 0$) $\rightarrow x = 0.0$

Abwägungen: Grobe vs. feine Partitionen

Grob (2)	2	Basis	Niedrig	Einfache Funktionen, niedriges Risiko
Fein (3)	3	Gut	Mittel	Die meisten Funktionen (empfohlen)
Sehr fein (5)	5	Gründlich	Hoch	Kritische Funktionen, hohes Risiko

Richtlinien:

- **Grob:** Schnelle Smoke-Tests, risikoarme Funktionen
- **Fein:** Standardwahl für die meisten Funktionen
- **Sehr fein:** Sicherheitskritisch, Finanzberechnungen, bekannt fehleranfällige Bereiche

Richtlinien zur Wahl der Granularität

Berücksichtige diese Faktoren:

1. **Code-Pfade:** Wie viele verschiedene Verzweigungen in der Implementierung?
2. **Mathematische Eigenschaften:** Vorzeichenwechsel? Spezialwerte (0, 1, -1)?
3. **Domain-Bedeutung:** Gibt es natürliche Kategorien? (z.B. Winkel: 0°, 90°, 180°)
4. **Bug-Risiko:** War dieser Bereich vorher schon fehlerhaft?
5. **Kosten des Versagens:** Was passiert, wenn es kaputt geht? (Geldverlust? Sicherheitsrisiko?)

Pragmatischer Ansatz: Starte mit 3-5 Klassen, füge mehr hinzu, wenn Bugs gefunden werden.

White-Box vs. Black-Box Testing

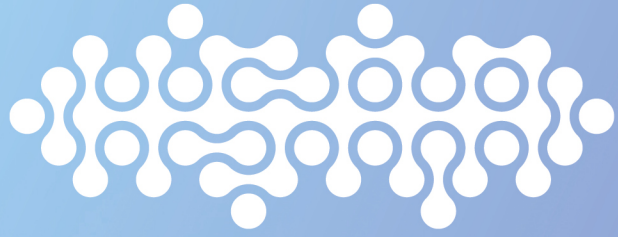
Black-Box Testing:

- Du siehst die Implementierung **nicht**
- Partitionierung basiert auf **Spezifikation** und **Domänenwissen**
- Beispiel: "reciprocal sollte für positive und negative Zahlen funktionieren"

White-Box Testing:

- Du **kannst** die Implementierung sehen
- Partitionierung basiert auf **Code-Struktur** und **Logik**
- Beispiel: "Ich sehe eine `if x == 0` Prüfung, also ist Null eine separate Klasse"

Best Practice: Kombiniere beides! 🎯



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Winkel & Arrays

4

Beispiel 3: calculate_ray_slope(angle_degrees)

```
def calculate_ray_slope(angle_degrees: float) -> float | None:
    """Calculate ray slope from angle. Returns None for vertical rays."""
    angle_rad = -np.deg2rad(angle_degrees)

    # Vertical ray detection (cos ≈ 0)
    if np.abs(np.cos(angle_rad)) < 1e-10:
        return None

    slope = np.tan(angle_rad)
    return slope
```

🤔 *Wie viele Äquivalenzklassen? 3 wie vorher, oder anders?*

calculate_ray_slope: Vier Äquivalenzklassen!

Abwärts-Strahlen	$-90^\circ < \text{angle} < 0^\circ$	$\text{angle} = -45^\circ$	Negative Steigung (1.0)
Horizontale Strahlen	$\text{angle} = 0^\circ$	$\text{angle} = 0^\circ$	Null-Steigung (0.0)
Aufwärts-Strahlen	$0^\circ < \text{angle} < 90^\circ$	$\text{angle} = 45^\circ$	Positive Steigung (-1.0)
Vertikale Strahlen	$\text{angle} = \pm 90^\circ$	$\text{angle} = 90^\circ$	None (keine Steigung)

Warum 4 statt 3? Null hat hier **zwei Bedeutungen**:

- Horizontaler Strahl ($\text{angle} = 0^\circ$) → Steigung ist 0
- Vertikaler Strahl ($\cos = 0$) → keine Steigung (gibt **None** zurück)

calculate_ray_slope: Test-Code

```
def test_downward_ray():  
    """Abwärts-Strahl (-45°) → positive Steigung"""  
    assert calculate_ray_slope(-45.0) == pytest.approx(1.0, abs=0.01)  
  
def test_horizontal_ray():  
    """Horizontaler Strahl (0°) → Null-Steigung"""  
    assert calculate_ray_slope(0.0) == pytest.approx(0.0, abs=0.01)  
  
def test_upward_ray():  
    """Aufwärts-Strahl (45°) → negative Steigung"""  
    assert calculate_ray_slope(45.0) == pytest.approx(-1.0, abs=0.01)  
  
def test_vertical_ray():  
    """Vertikaler Strahl (90°) → None (keine Steigung)"""  
    assert calculate_ray_slope(90.0) is None
```

Meta-Erkenntnis: Gleiches Muster, unterschiedliche Granularität

<code>reciprocal(x)</code>	3	Vorzeichen der Eingabe: +, -, 0
<code>reciprocal_sum(x,y,z)</code>	3	Vorzeichen der Summe : +, -, 0
<code>calculate_ray_slope(angle)</code>	4	Richtung + Sonderfall (vertikal)

Zentrale Erkenntnis:

- Gleiches **Muster** (positiv/negativ/spezial)
- Unterschiedliche **Granularität** basierend auf Kontext
- **Du entscheidest** wie fein partitioniert wird basierend auf Anforderungen

Warum Null hier besonders ist

Für `reciprocal(x)` :

- `x = 0` → **Fehler** (kann nicht durch Null teilen)
- Klare mathematische Grenze

Für `calculate_ray_slope(angle)` :

- `angle = 0°` → **gültiges Ergebnis** (horizontaler Strahl, Steigung = 0)
- `cos(angle) ≈ 0` (d.h. $\text{angle} \approx \pm 90^\circ$) → **Sonderfall** (vertikaler Strahl, keine Steigung)

Allgemeines Prinzip:

Null (oder jeder Spezialwert) ist eine separate Klasse, wenn es **unterschiedliches Verhalten** auslöst (Fehler, Sonderfall, spezielle Logik).

Arrays: Eine neue Dimension von Komplexität

Bisher: Einfache numerische Eingaben (Floats, Ints)

Jetzt: Arrays führen **zwei Dimensionen** von Äquivalenz ein:

1. Strukturelle Äquivalenzklassen:

- Wie viele Elemente?
- Leer vs. nicht-leer?

2. Wertbasierte Äquivalenzklassen:

- Welche Werte haben die Elemente?
- Positiv? Negativ? Gemischt?

Ergebnis: Komplexität multipliziert sich! 🚀

Beispiel 4: `array_sum(arr)`

```
def array_sum(arr: list[float]) -> float:  
    """Returns sum of array elements"""  
    return sum(arr)
```

Strukturelle Klassen (nach Länge):

- Leer: `[]`
- Ein Element: `[5.0]`
- Zwei Elemente: `[1.0, 2.0]`
- Viele Elemente: `[1.0, 2.0, 3.0, 4.0]`

array_sum: Wert-basierte Klassen

Wert-Klassen (nach Inhalt):

- Alle positiv: [1.0, 2.0, 3.0]
- Alle negativ: [-1.0, -2.0, -3.0]
- Gemischte Vorzeichen: [1.0, -2.0, 3.0]
- Enthält Nullen: [1.0, 0.0, 2.0]
- Alle Nullen: [0.0, 0.0]

Kombination: Strukturelle × Wert-Dimensionen = erhöhte Komplexität!

Kombinierte Komplexität: $4 \times 5 = 20$ Tests?

Strukturell (4) $\times$ Wert (5) = 20 potenzielle Tests

Leer	N/A	N/A	N/A	N/A	✓
Eins	✓	✓	N/A	✓	✓
Zwei	✓	✓	✓	✓	✓
Viele	✓	✓	✓	✓	✓

Brauchen wir alle 20? Nicht unbedingt! Strategisches Sampling ist der Schlüssel.

Strategisches Testen für array_sum

✓ Praktischer Ansatz (8-10 Tests):

```
# Strukturelle Abdeckung
test_empty_array()          # []
test_single_element()       # [5.0]
test_many_elements()        # [1, 2, 3, 4]

# Wert-Abdeckung (mit "viele" Struktur)
test_all_positive()         # [1, 2, 3]
test_all_negative()         # [-1, -2, -3]
test_mixed_signs()          # [1, -2, 3]
test_contains_zeros()       # [1, 0, 2]

# Sonderfälle
test_all_zeros()            # [0, 0, 0]
```

Schlüssel: Decke jede **Dimension** mit 2-3 Repräsentanten ab, plus Sonderfälle.

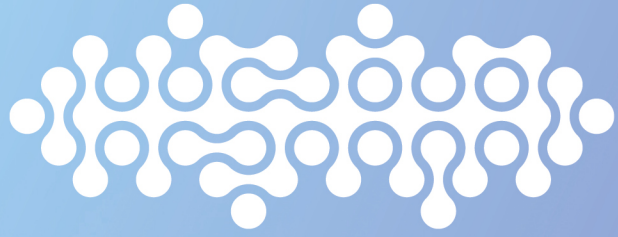
Häufiger Fehler: Fokus auf falsche Dimension

✗ Schlechter Ansatz:

```
test_three_elements()    # [1, 2, 3]
test_four_elements()     # [1, 2, 3, 4]
test_five_elements()     # [1, 2, 3, 4, 5]
test_six_elements()      # [1, 2, 3, 4, 5, 6]
# Alle testen "viele positive Elemente" - GLEICHE Äquivalenzklasse!
```

✓ Guter Ansatz:

```
test_many_positive()     # [1, 2, 3, 4]      (viele + positiv)
test_many_negative()     # [-1, -2, -3, -4] (viele + negativ)
test_many_mixed()        # [1, -2, 3, -4]   (viele + gemischt)
# Unterschiedliche WERT-Klassen mit GLEICHER Struktur
```



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Diskrete Funktionen

5

Gute Nachricht: Diskrete Funktionen sind einfacher! ✨

Alle bisherigen Beispiele hatten kontinuierliche Ausgaben:

- `reciprocal(x)` → $\mathbb{R}$ (unendlich viele reelle Zahlen)
- `calculate_ray_slope(angle)` → $\mathbb{R} \cup \{\text{None}\}$

Wir mussten *wählen*, wie wir die Ausgaben partitionieren (3 Kategorien? 4? 5?)

Aber was ist mit Funktionen mit diskreten/endlichen Ausgaben?

- Notenstufen: `{"A", "B", "C", "D", "F"}`
- Boolesche Ergebnisse: `{True, False}`
- Status-Codes: `{200, 404, 500}`

Die Vereinfachung: Ausgabekategorien sind **vordefiniert!** 🎯

Kontinuierliche vs. Diskrete Funktionen

Aspekt	Kontinuierlich	Diskret
Ausgaberaum	Unendlich ($\mathbb{R}$)	Endliche Menge
Partitionierung	Du wählst	Vordefiniert
Anzahl Klassen	Deine Designentscheidung	Fest durch Rückgabebetyp
Haupt Herausforderung	"Wie partitioniere ich Ausgaben?"	"Welche Eingaben → welche Ausgaben?"

- **Kontinuierlich:** Designproblem (wie partitionieren?)
- **Diskret:** Analyseaufgabe (welche Eingaben erzeugen welche Ausgabe?)

Beispiel 1: Notenstufen

```
def calculate_grade(score: int) -> str:
    """Berechne Notenstufe (A-F)."""
    if score < 0 or score > 100:
        raise ValueError("Punktzahl muss zwischen 0 und 100 liegen")

    if score >= 90:    return "A"
    elif score >= 80: return "B"
    elif score >= 70: return "C"
    elif score >= 60: return "D"
    else:             return "F"
```

🤔 *Wie viele Äquivalenzklassen für `score` ?*

Notenfunktion: 6 Äquivalenzklassen

Klasse	Bereich	Repräsentant	Ausgabe
Note A	$90 \leq \text{score} \leq 100$	<code>score = 95</code>	<code>"A"</code>
Note B	$80 \leq \text{score} < 90$	<code>score = 85</code>	<code>"B"</code>
Note C	$70 \leq \text{score} < 80$	<code>score = 75</code>	<code>"C"</code>
Note D	$60 \leq \text{score} < 70$	<code>score = 65</code>	<code>"D"</code>
Note F	$0 \leq \text{score} < 60$	<code>score = 30</code>	<code>"F"</code>
Ungültig	<code>score < 0 or > 100</code>	<code>score = -10</code>	<code>ValueError</code>

Warum 6? 5 gültige Ausgaben + 1 Fehler = 6 verschiedene Verhaltensweisen

Notenfunktion: Testcode

```
def test_calculate_grade_A():
    """Äquivalenzklasse: Note A (90-100)"""
    assert calculate_grade(95) == "A"
    assert calculate_grade(90) == "A"    # Grenzwert
    assert calculate_grade(100) == "A"   # Grenzwert

def test_calculate_grade_B():
    """Äquivalenzklasse: Note B (80-89)"""
    assert calculate_grade(85) == "B"
    assert calculate_grade(80) == "B"    # Grenzwert
    assert calculate_grade(89) == "B"    # Grenzwert

# ... Tests für C, D, F, Ungültig ...
```

Muster: Ein Test pro Klasse + Grenzwerte

Beispiel 2: Passwortvalidierung

```
def validate_password(password: str) -> tuple[bool, str]:  
    """Validiere Passwortstärke."""  
    if not isinstance(password, str):  
        return (False, "Passwort muss ein String sein")  
    if len(password) < 8:  
        return (False, "Passwort zu kurz")  
    if len(password) > 20:  
        return (False, "Passwort zu lang")  
    if not any(c.isupper() for c in password):  
        return (False, "Muss Großbuchstaben enthalten")  
    # ... weitere Prüfungen ...  
    return (True, "Passwort ist gültig")
```

🤔 *Wie viele Äquivalenzklassen?*

Passwortvalidierung: Basis-Prüfungen

Klasse	Repräsentant	Ausgabe
Gültig	"SecurePwd1!"	(True, "gültig")
Kein String	12345	(False, "muss String sein")
Zu kurz	"Abc1!"	(False, "zu kurz")
Zu lang	"VeryLong..."	(False, "zu lang")

4 Klassen bisher... was ist mit Zeichenanforderungen? 🤔

Passwortvalidierung: Inhaltsanforderungen

Klasse	Repräsentant	Ausgabe
Keine Großbuchstaben	"password1!"	(False, "Großbuchstaben")
Keine Kleinbuchstaben	"PASSWORD1!"	(False, "Kleinbuchstaben")
Keine Ziffer	"Password!"	(False, "Ziffer")
Kein Sonderzeichen	"Password1"	(False, "Sonderzeichen")

Gesamt: 1 gültig + 7 Fehlerpfade = 8 Äquivalenzklassen!

Wichtige Erkenntnis: Jede Validierungsprüfung = eine Äquivalenzklasse!

Passwortvalidierung: Reihenfolge ist wichtig!

Early-Return-Muster:

```
if len(password) < 8:  
    return (False, "Passwort zu kurz") # Erste Prüfung  
  
if not any(c.isupper() for c in password):  
    return (False, "Muss Großbuchstaben enthalten") # Spätere Prüfung
```

Was ist mit "abc" (zu kurz UND fehlende Großbuchstaben)?

- Gehört zur "zu kurz" Klasse (erste fehlgeschlagene Prüfung gewinnt!)

Teststrategie:

- Ein Test pro Validierungspfad. Repräsentanten lösen **einen** spezifischen Fehler aus.

Beispiel 3: Versandkosten

```
def calculate_shipping_cost(weight: float, distance: float,
                             express: bool = False) -> float:
    # Validierung ausgelassen

    # Gewichtskategorien
    if weight <= 5:         base = 5.0
    elif weight <= 20:      base = 10.0
    else:                   base = 20.0

    # Entfernungsmultiplikator
    if distance <= 100:     mult = 1.0
    elif distance <= 500:   mult = 1.5
    else:                   mult = 2.0

    return base * mult * (1.5 if express else 1.0)
```

Versandkosten: Drei Dimensionen

(3 Klassen)	(3 Klassen)	(2 Klassen)
Leicht: ≤ 5 kg $\rightarrow$ €5	Lokal: ≤ 100 km $\rightarrow$ 1.0×	Standard $\rightarrow$ 1.0×
Mittel: 5-20 kg $\rightarrow$ €10	Regional: 100-500 km $\rightarrow$ 1.5×	Express $\rightarrow$ 1.5×
Schwer: > 20 kg $\rightarrow$ €20	Fernstrecke: > 500 km $\rightarrow$ 2.0×	

Versandkosten: Kombinatorische Herausforderung

Gesamtkombinationen:

$$3 \times 3 \times 2 = \mathbf{18} \text{ gültige Klassen}$$

Plus 2 Fehlerklassen (ungültiges Gewicht, ungültige Entfernung)

Gesamt: 20 Äquivalenzklassen! 🤯

Müssen wir alle 18 testen? Nicht unbedingt!

Strategisches Testen: Dimensionsabdeckung

Strategie: Jede Dimension abdecken, nicht alle Kombinationen

```
# Gewichtsdimension (fixiere distance=50, express=False)
test_light_weight()    # 2.5 kg → €5
test_medium_weight()  # 10 kg → €10
test_heavy_weight()    # 30 kg → €20

# Entfernungsdimension (fixiere weight=2.5, express=False)
test_local_distance()  # 50 km → €5 × 1.0
test_regional_distance() # 250 km → €5 × 1.5
test_long_distance()   # 1000 km → €5 × 2.0

# Express-Dimension (fixiere weight=2.5, distance=50)
test_standard_shipping() # €5 × 1.0
test_express_shipping()  # €5 × 1.5

# Grenzwerttests + Fehlertests
```

Ergebnis: ~10 strategische Tests statt 18!

Wichtige Erkenntnisse: Diskret vs. Kontinuierlich

Wann diskret EINFACHER ist:

- ✓ Ausgabekategorien sind offensichtlich (einfach Rückgabewerte auflisten!)
- ✓ Keine Granularitätsdebatten (Funktionssignatur definiert Klassen)
- ✓ Vollständige Abdeckung erreichbar (6 Noten? Schreibe 6 Tests!)

Wann diskret SCHWIERIGER ist:

- ✗ Viele mögliche Ausgaben (8 Validierungsfehler = 8 Tests minimum)
- ✗ Kombinatorische Explosion ($3 \times 3 \times 2 = 18$ Klassen)
- ✗ Hierarchische Validierung (Reihenfolge ist wichtig - Early Returns)
- ✗ Grenzwerte bleiben kritisch (89 vs 90 kann Bugs verbergen)

Fazit: Diskret vereinfacht Ausgabepartitionierung, nicht Eingabeanalyse!

Praktische Richtlinien

Verwende diskrete Analyse wenn:

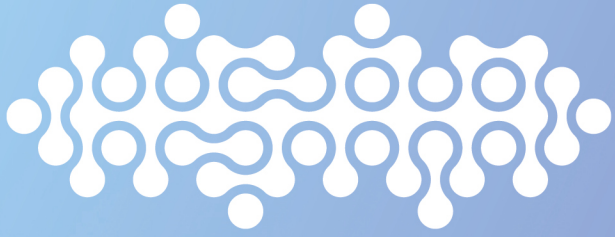
- ✓ Endliche Rückgabewerte | Explizite Validierung
- ✓ Schwellenwert-Logik | Kontinuierlich → Diskret

Verwende kontinuierliche Analyse wenn:

- ✓ Math. Berechnungen auf $\mathbb{R}$ | Wirklich kontinuierlich
- ✓ Granularität muss gewählt werden

Verwende hybride Analyse wenn:

- ✓ Kontinuierliche Eingaben → diskrete Ergebnisse
- ✓ Geometrie/Physik | Berechnungen + bedingte Logik



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Reale Komplexität: `find_intersection()`

6

Verbindung: find_intersection() ist Hybrid!

Sieht kontinuierlich aus:

- Eingaben: Floats (Winkel, Position, Straßenpunkte)
- Ausgabe: `tuple[float, float] | None` (Koordinaten)

Aber Ergebnisse sind diskret:

1. **Keine Schnittmenge** → `None`
2. **Strahl schneidet aufsteigendes Segment**
3. **Strahl schneidet absteigendes Segment**
4. **Strahl schneidet horizontales Segment**
5. **Strahl schneidet an Scheitelpunkt** (Randfall)

Teststrategie für hybride Funktionen

Wichtige Erkenntnis: Obwohl Eingaben kontinuierlich sind (Floats), sind die **Ergebnisse diskret!**

Teststrategie:

Diskrete Ergebniskategorien identifizieren, dann repräsentative Eingaben für jede wählen!

Genau das haben wir mit `calculate_shipping_cost` gemacht! 🎯

Muster:

- Kontinuierliche Eingaben → Diskrete Ergebnisse → Einen Repräsentanten pro Ergebnis testen

Die echte Herausforderung: find_intersection()

Aus deinem Road Profile Viewer Projekt:

```
def find_intersection(
    x_road: np.ndarray,      # Road profile arrays
    y_road: np.ndarray,
    angle_degrees: float,    # Ray direction
    camera_x: float = 0,     # Camera position
    camera_y: float = 1.5
) -> tuple[float, float] | None:
    # ... 70 lines of geometric logic ...
```

Denk darüber nach: Wie würdest du das testen? 🤔

Vertiefung verfügbar

★ **Wichtig:** Dieses Beispiel ist so komplex, dass wir eine **separate detaillierte Anleitung** mit 22 visuellen Illustrationen erstellt haben!

Siehe das "**Vertiefung - Testen von `find_intersection()`**" Foliendeck für die vollständige Analyse.

📖 **Vollständige Details:**

[Vorlesungsnotizen zu `find\_intersection\(\)`](#)

Wie viele Tests werden gebraucht?

Interaktive Umfrage:

🤔 Wie viele Tests brauchen wir für `find_intersection()` ?

- A) 5-10 Tests
- B) 20-30 Tests
- C) 50-100 Tests
- D) Mehr als 100 Tests

Diskutiere mit deinem Nachbarn! ⌚ (30 Sekunden)

Komplexitätsdimension 1: Array-Struktur

x_road und y_road Arrays:

1. **Leer** (`len = 0`) → Fehlerbehandlung
2. **Ein Punkt** (`len = 1`) → degenerierter Fall
3. **Zwei Punkte** (`len = 2`) → ein Segment, minimal gültig
4. **Drei Punkte** (`len = 3`) → zwei Segmente
5. **Viele Punkte** (`len > 3`) → mehrere Segmente, typischer Fall

Komplexitätsdimension 2: Winkel-Klassen

angle_degrees:

1. **Abwärts-Strahl** ($-90^\circ < \text{angle} < 0^\circ$)
2. **Horizontaler Strahl** ($\text{angle} = 0^\circ$)
3. **Aufwärts-Strahl** ($0^\circ < \text{angle} < 90^\circ$)
4. **Vertikaler Strahl** ($\text{angle} = \pm 90^\circ$) → keine Steigung, spezielle Behandlung
5. **Außerhalb des Bereichs** ($|\text{angle}| > 90^\circ$) → ungültige Eingabe

Komplexitätsdimension 3: Kamera-Position

1. **Kamera weit links** (vor dem ersten Straßenpunkt)
2. **Kamera zwischen erstem und zweitem Punkt**
3. **Kamera in Straßenmitte**
4. **Kamera zwischen letzten zwei Punkten**
5. **Kamera weit rechts** (nach dem letzten Straßenpunkt)
6. **Kamera über/unter Straße** (y-Position)

Warum das wichtig ist:

- Position beeinflusst welche Segmente schneidbar sind
- Randpositionen testen Grenzlogik

Komplexitätsdimension 4: Ergebnis-basierte Klassen

Basierend darauf, was die Funktion zurückgibt:

1. **Schnittpunkt gefunden** → `(x, y)`
2. **Kein Schnittpunkt** (Strahl verfehlt Straße) → `None`
3. **Mehrere mögliche Schnittpunkte** (welchen wählen?)
4. **Strahl parallel zu Segment** (Sonderfall) → `None`

Warum das wichtig ist:

- Jedes Ergebnis durchläuft unterschiedliche Code-Pfade
- Sonderfälle offenbaren Bugs

Komplexitätsdimension 5: Straßenform

Geometrische Eigenschaften:

1. **Flache Straße** (alle y-Werte gleich)
2. **Monoton aufsteigend/absteigend**
3. **Hat Gipfel und Täler**

Warum das wichtig ist:

- Flache Straße vereinfacht Geometrie
- Gipfel/Täler erhöhen Schnittpunkt-Komplexität
- Beeinflusst welche Strahlwinkel Schnittpunkte finden

Fünf Dimensionen der Komplexität

Für `find_intersection()` haben wir identifiziert:

1. **Array-Struktur:** 5 Klassen
2. **Winkel:** 5 Klassen
3. **Kamera-Position:** 6 Klassen
4. **Ergebnis:** 4 Klassen
5. **Straßenform:** 3 Klassen (vereinfacht)

Rechnen wir mal... 

Die Mathematik: Kombinatorische Explosion! 🤯

$$5 \times 5 \times 6 \times 4 = \mathbf{600} \text{ Tests!}$$

(Wir haben die Straßenform vereinfacht, um über 1000 zu vermeiden!)

Realitätscheck: Kannst du 600 Tests schreiben und pflegen? 😊

Spoiler: Nein! Hier kommt menschliche Einsicht ins Spiel... 🧠

KI-Einschränkungen: Was LLMs übersehen

Du denkst vielleicht: "Frag einfach ChatGPT/Claude, alle 600 Tests zu generieren!"

Probleme mit KI-generierten Tests:

1. **Keine geometrische Intuition:** Kann Strahl-Straßen-Schnittpunkte nicht visualisieren
2. **Kein Domänenwissen:** Versteht Kamera-Straßen-Beziehungen nicht
3. **Musterwiederholung:** Generiert ähnliche Tests mit unterschiedlichen Zahlen
4. **Übersieht Sonderfälle:** Keine Kenntnis von parallelen Strahlen, degenerierter Geometrie
5. **Keine Priorisierung:** Behandelt alle Kombinationen gleich (weiß nicht was wichtig ist)

Fazit: LLMs können **helfen**, aber **nicht ersetzen** menschliche Einsicht! 🧠

Was Menschen sehen (was KI nicht sieht)

Menschliche Einsicht:

- "Abwärts-Strahl von Kamera über Straße wird **immer** Straße verfehlen" → diese Kombinationen überspringen
- "Kamera weit rechts + Aufwärts-Strahl → Schnittpunkt unmöglich" → überspringen
- "Zwei-Punkt-Straße + vertikaler Strahl → funktioniert nur wenn Strahl durch Punkte geht" → seltener Sonderfall
- "Flache Straße macht Winkel weniger wichtig" → Winkel-Tests reduzieren

Ergebnis: Menschen können 600 → **20-30 strategische Tests** reduzieren basierend auf Domänenwissen.

Das ist dein Wert als Softwareentwickler:in! 🎯

Praktische Reduktionsstrategie: 600 → 30 Tests

Strategie: Decke jede Dimension mit 2-3 Repräsentanten ab

1. **Array-Struktur** (5 Klassen) → teste 3: leer, zwei-Punkte, viele-Punkte
2. **Winkel** (5 Klassen) → teste 3: abwärts, horizontal, aufwärts
3. **Kamera-Position** (6 Klassen) → teste 2: Mitte, Rand
4. **Ergebnis** (4 Klassen) → stelle sicher alle 4 erscheinen in Test-Suite
5. **Straßenform** (3 Klassen) → teste 2: flach, variiert

Die Rechnung: 600 → 30 Tests

Rechnung: $3 + 3 + 2 + 4 + 2 = 14$ Basis-Tests

Plus Sonderfälle: Vertikaler Strahl, paralleles Segment, außerhalb Bereich → **~10 mehr**

Gesamt: **~25-30 Tests** ✓

💡 **Willst du die tatsächlichen Tests sehen?** Sieh dir das "**Vertiefung - Testen von find_intersection()**" Foliendeck für alle 22 Tests mit visuellen Diagrammen an!

Beispiel Test-Klassen-Struktur (Überblick)

```
class TestFindIntersection:
    # Strukturelle Dimension
    def test_empty_arrays(self): ...
    def test_two_point_road(self): ...
    def test_many_point_road(self): ...

    # Winkel-Dimension
    def test_downward_ray(self): ...
    def test_horizontal_ray(self): ...
    def test_upward_ray(self): ...
    def test_vertical_ray(self): ... # Sonderfall

    # Ergebnis-Dimension
    def test_intersection_found(self): ...
    def test_no_intersection(self): ...
    def test_ray_parallel_to_segment(self): ...
```

Beispiel Test-Klassen-Struktur (Kombinationen)

```
class TestFindIntersection:
    # ... (vorherige Tests)

    # Intelligente Kombinationen (Domain-getrieben)
    def test_camera_on_road_horizontal_ray(self): ...
    def test_camera_above_road_downward_ray_hits(self): ...
    def test_camera_below_road_upward_ray_hits(self): ...
    def test_camera_far_right_upward_ray_misses(self): ...
    def test_flat_road_any_angle(self): ...
    def test_peaked_road_multiple_intersections(self): ...

    # Sonderfälle
    def test_ray_through_road_endpoint(self): ...
    def test_angle_out_of_range(self): ...
```

Kollaboratives Testen: Mensch + LLM

Mensch: Klassen identifizieren, Granularität wählen, Risiken priorisieren, überprüfen

LLM: Boilerplate generieren, Testdaten erstellen, Assertions schreiben, Sonderfälle vorschlagen

Workflow: Du entwirfst → LLM implementiert → Du überprüfst! 🤝

LLMs effektiv zum Testen nutzen

✓ Gute Prompts:

- "Generiere einen Test für `find_intersection` mit flacher Zwei-Punkt-Straße und horizontalem Strahl"
- "Schreibe einen Test der `None` erwartet wenn der Strahl die Straße verfehlt"

✗ Schlechte Prompts:

- "Generiere alle möglichen Tests für `find_intersection`" → produziert minderwertiges Rauschen
- "Teste diese Funktion" → vage, kein Kontext

Schlüssel: Sei **spezifisch** über Äquivalenzklassen und erwartetes Verhalten!

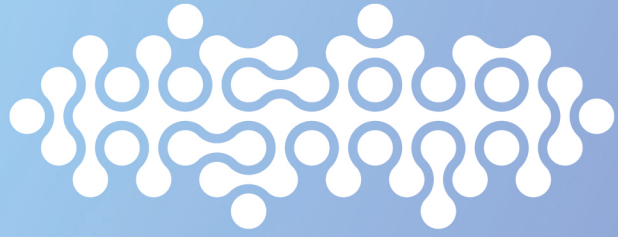
Diskussion: Reale Abwägungen

Szenario: Du testest `find_intersection()` für ein selbstfahrendes Auto.

Diskussionsfragen:

1. Würdest du alle 600 Kombinationen testen? Warum oder warum nicht?
2. Wie würde sich deine Antwort ändern wenn:
 - Es ein Videospiel wäre (nicht sicherheitskritisch)?
 - Es medizinische Bildgebungssoftware wäre (Leben stehen auf dem Spiel)?
 - Du 2 Tage vs. 2 Wochen fürs Testen hättest?
3. Welche Kombinationen haben das **höchste Risiko**?

Think-Pair-Share: Diskutiere mit deinem Nachbarn! 🕒 (2 Minuten)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Zentrale Erkenntnisse & Abschluss

7

Zentrale Erkenntnisse

1. Äquivalenzklassen reduzieren unendliche Eingaben auf endliche Tests

- Gruppiere Eingaben nach ähnlichem Verhalten, teste einen pro Klasse

2. Ausgabe-Kategorien bestimmen Eingabe-Klassen

- Schaue was die Funktion **macht**, nicht was sie **nimmt**
- `reciprocal_sum(x, y, z)`: 3 Ausgabeverhalten → 3 Klassen

3. Granularität ist eine Design-Entscheidung

- Einfach/risikoarm: 2-3 Klassen | Komplex/hochriskant: 5-10 Klassen
- Berücksichtige: Code-Pfade, Domain, Bug-Risiko, Fehlerkosten

Zentrale Erkenntnisse (fortgesetzt)

4. White-Box-Testing offenbart verborgene Äquivalenzklassen

- Code anschauen zeigt was **tatsächlich** Verhalten bestimmt
- Beispiel: `reciprocal_sum` Code zeigt Summe ist entscheidend

5. Reale Funktionen haben kombinatorische Komplexität

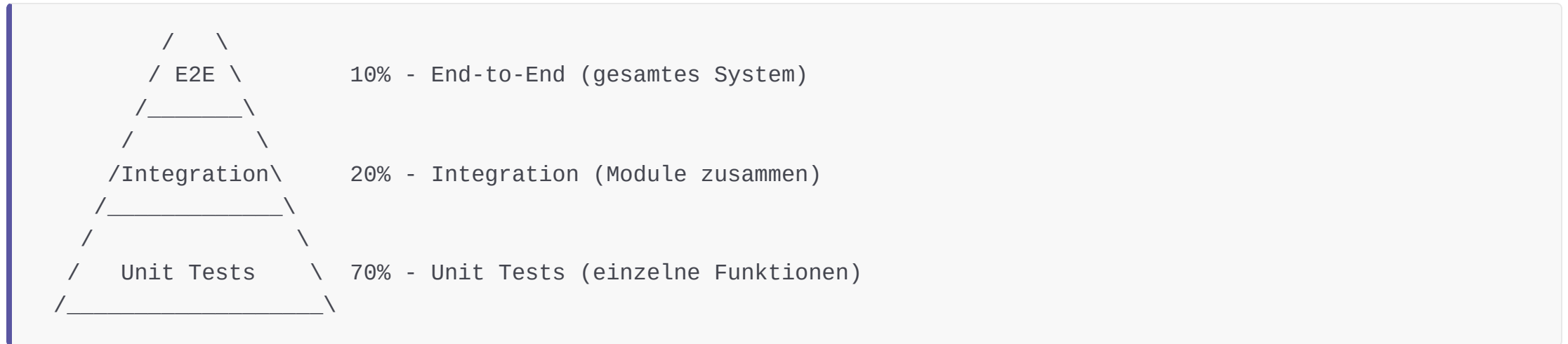
- `find_intersection`: 5 Dimensionen $\times$ viele Klassen = 600 Tests
- Reduziere via Domänenwissen: 600 $\rightarrow$ 25-30 strategische Tests

6. Menschliche Einsicht ist unersetzlich

- LLMs können Geometrie nicht visualisieren, Domain nicht verstehen, nicht priorisieren
- Du entwirfst Äquivalenzklassen, LLM hilft Tests zu implementieren

Das große Bild

Testing-Pyramide (Erinnerung):



Heutiger Fokus: Wie wählt man **welche Unit-Tests** effizient schreibt!

Nächstes Thema: Grenzwertanalyse (Bugs an Rändern finden)

Vorschau: Grenzwertanalyse (Kommt als Nächstes)

Äquivalenzklassen-Testen ist mächtig, aber nicht genug!

Problem: Bugs verstecken sich oft an **Grenzen** zwischen Klassen.

Beispiel für `reciprocal(x)`:

- Wir haben getestet: `x = 5.0`, `x = -5.0`, `x = 0.0` ✓
- Aber was ist mit: `x = 0.00000001` ? (sehr nah an Null) 🤔
- Oder: `x = 1e308` ? (sehr groß, nah am Float-Maximum)

Grenzwertanalyse sagt: Teste an den **Rändern** der Äquivalenzklassen!

Nächste Vorlesung: Wie man Grenzen systematisch findet und testet.

Praktische Übung

Aufgabe: Wähle Äquivalenzklassen für diese Funktion:

```
def classify_temperature(temp_celsius: float) -> str:
    """Classify temperature for weather app"""
    if temp_celsius < 0:
        return "freezing"
    elif temp_celsius < 10:
        return "cold"
    elif temp_celsius < 20:
        return "mild"
    elif temp_celsius < 30:
        return "warm"
    else:
        return "hot"
```

Nimm dir 2 Minuten: Arbeite allein oder mit einem Partner 

Übungsfragen

Für `classify_temperature(temp_celsius)` :

1. Wie viele Äquivalenzklassen?
2. Welche sind es?
3. Welche repräsentativen Werte würdest du wählen?

Bonus: Welche Grenzwerte solltest du testen?

Vollständige Vorlesung: [Kapitel 03 \(Testing-Grundlagen\) - Testing-Grundlagen](#)

Zusammenfassung: Das Muster

Für jede Funktion, frage:

1. **Was sind die verschiedenen Ausgabe-Verhaltensweisen?** (Ergebnisse, Rückgabetypen, Fehler)
2. **Welche Eingaben produzieren jedes Verhalten?** (gruppiere nach Ähnlichkeit)
3. **Wie fein soll ich partitionieren?** (basierend auf Risiko, Komplexität, Ressourcen)
4. **Was ist ein Repräsentant pro Klasse?** (typischer Wert der das Verhalten ausübt)

Ergebnis: Endliche, handhabbare Test-Suite die unendlichen Eingaberaum abdeckt! 🎯

Denk dran: Du testest nicht alle Eingaben — du testest alle **Verhaltensweisen**!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Fragen?

DANKE