



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Kapitel 03

TESTING FUNDAMENTALS

Boundary Analysis und LLM-Assisted Testing

Willkommen zurück!

In **Teil 1** habt ihr gelernt:

- **Testing Pyramid** (70% Unit, 20% Integration, 10% E2E)
- **AAA Pattern** (Arrange-Act-Assert)
- **Äquivalenzklassen** (Gruppierung von Eingaben nach Verhalten)
- **pytest Grundlagen** (Tests schreiben und ausführen)

Aber wir haben euch mit Fragen zurückgelassen...

Die Herausforderung aus Teil 1

Ihr denkt wahrscheinlich:

- 🤔 "Woher weiß ich, ob ich **genug** getestet habe?"
- 🤔 "Was ist mit den kniffligen **Edge Cases** an den Grenzen?"
- 🤔 "20-30 Tests für eine Funktion klingt **mühsam**!"
- 🤔 "Kann ich **KI** nutzen, um Tests schneller zu schreiben?"

Heute lösen wir diese Probleme!

Die heutige Mission

Drei mächtige Techniken

1. **Boundary Value Analysis**

- Wo Bugs sich tatsächlich verstecken (nicht in der Mitte!)

2. **LLM-Assisted Testing**

- Den "Test Cone" mit KI-Unterstützung durchbrechen

3. **Integration Testing & Maintainability**

- Testing zum Teil eures Workflows machen

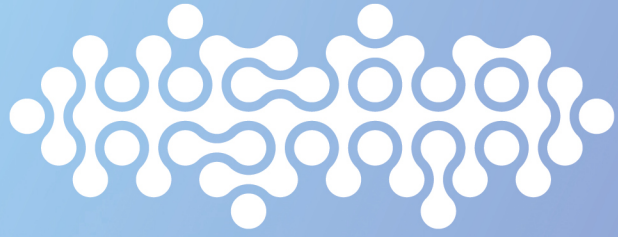
Lernziele

Am Ende dieser Vorlesung werdet ihr:

- ✓ **Grenzwerte systematisch identifizieren** können für jede Funktion
- ✓ **LLMs nutzen** können, um Test-Boilerplate zu generieren (und KI-Fallstricke vermeiden)
- ✓ **Integrationstests schreiben**, die prüfen, ob Module zusammenarbeiten
- ✓ **Feature Branch Workflow anwenden**, um Tests zu eurem Projekt hinzuzufügen
- ✓ **Wissen**, wann ihr "genug" Tests habt (realistische Coverage, keine Perfektion)

Der Weg nach vorne

- ✓ Teil 1: Grundlagen
Warum testen? Wie schreibt man grundlegende Unit Tests?
- Teil 2: Meisterschaft (HEUTE!)
Wo verstecken sich Bugs? Wie testet man effizient?
- Kapitel 03 (TDD und CI): TDD & CI
Tests ZUERST schreiben, alles automatisieren



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Boundary Value Analysis

1

Kernbeobachtung

Bugs lauern oft an den Grenzen zwischen Äquivalenzklassen

Warum Grenzen?

- Off-by-one Fehler
- Fließkomma-Präzisionsprobleme
- Edge Cases in bedingter Logik (`if x < 0` , `if x == 0` , `if x > 0`)

Strategie: An den Grenzen testen, nicht nur in der Mitte!

Beispiel: `reciprocal(x) = 1/x`

Erinnerung aus Teil 1: Wir haben 5 Äquivalenzklassen identifiziert

Klasse	Repräsentant
Positive	<code>x = 5.0</code>
Negative	<code>x = -3.0</code>
Null	<code>x = 0.0</code>
Sehr groß	<code>x = 1e100</code>
Spezialwerte	<code>inf</code> , <code>nan</code>

Frage: Wo sind die **Grenzen** zwischen diesen Klassen?

Grenzen für `reciprocal(x)`

Äquivalenzklasse	Zu testende Grenzwerte
Positive Zahlen	<code>x = 0.0001</code> (nahe Null), <code>x = 1.0</code> , <code>x = 1000000.0</code>
Negative Zahlen	<code>x = -0.0001</code> , <code>x = -1.0</code> , <code>x = -1000000.0</code>
Null	<code>x = 0.0</code> , <code>x = 1e-100</code>
Extreme Grenzen	<code>sys.float_info.max</code> , <code>sys.float_info.min</code> , <code>float('inf')</code> , <code>float('nan')</code>

Kernaussage: Nahe Null, bei Null und bei Extremwerten testen!

Python Float-Grenzen (IEEE 754)

Python Floats sind **64-Bit IEEE 754 Doubles**

Python-Konstante	Typischer Wert	Beschreibung
<code>sys.float_info.max</code>	$\approx 1.8\text{e}308$	Größter endlicher Float
<code>sys.float_info.min</code>	$\approx 2.2\text{e}-308$	Kleinster positiver normalisierter
<code>sys.float_info.epsilon</code>	$\approx 2.2\text{e}-16$	Maschinengenauigkeit
<code>math.inf</code>	<code>inf</code>	Positive Unendlichkeit (nicht-endlich)
<code>math.nan</code>	<code>nan</code>	Not a Number

Kritischer Unterschied

`math.inf` vs `sys.float_info.max`

Sie testen verschiedene Dinge!

`math.inf` - Spezieller nicht-endlicher Wert

- Testet Unendlichkeits-Handling
- `1 / inf = 0.0`

`sys.float_info.max` - Größter endlicher Float

- Testet endliche Bereichsgrenzen
- `1 / sys.float_info.max ≈ 5.6e-309`

Umfassende Boundary Test Suite

```
import sys, math, pytest

# Konstanten für Lesbarkeit definieren
INF = math.inf
FMAX = sys.float_info.max
FMIN_NORM_POS = sys.float_info.min

def test_reciprocal_boundary_near_zero_positive():
    """Grenze: Sehr kleine positive Zahl"""
    result = reciprocal(1e-6)
    assert result == pytest.approx(1e6, rel=1e-9)

def test_reciprocal_boundary_exactly_zero():
    """Grenze: Exakt Null"""
    with pytest.raises(ZeroDivisionError):
        reciprocal(0.0)
```

Testen endlicher Grenzen

```
def test_reciprocal_boundary_max_finite_float():  
    """Grenze: Größter endlicher (sys.float_info.max ≈ 1.8e308)"""  
    result = reciprocal(FMAX)  
    assert result > 0  
    assert result < FMIN_NORM_POS # Underflow zu subnormal  
  
def test_reciprocal_boundary_min_normalized_float():  
    """Grenze: Kleinsten positiver normalisierter"""  
    result = reciprocal(FMIN_NORM_POS)  
    assert result == INF # Overflow zu Unendlichkeit!
```

Wichtig: Sowohl endliche Grenzen ALS AUCH Unendlichkeit testen!

Testen nicht-endlicher Grenzen

```
def test_reciprocal_boundary_positive_infinity():  
    """Grenze: Positive Unendlichkeit"""  
    result = reciprocal(INF)  
    assert result == 0.0 # 1/inf = 0  
  
def test_reciprocal_boundary_nan():  
    """Grenze: Not a Number"""  
    result = reciprocal(math.nan)  
    assert math.isnan(result) # NaN propagiert
```

Merken: `nan == nan` ist immer `False`! Nutzt `math.isnan()`.



Denkt darüber nach

Welcher Test ist wichtiger?

A) Testen von `reciprocal(5.0)` (Mitte der Äquivalenzklasse)

B) Testen von `reciprocal(0.0001)` (Grenze nahe Null)

Diskutiert 1 Minute mit eurem Nachbarn!

Multi-Parameter Grenzen

Funktion: `reciprocal_sum(x, y, z) = 1/(x+y+z)`

Neue Komplexität: Grenzen beinhalten jetzt **Kombinationen!**

Grenzbedingung	Testwerte	Warum wichtig
Summe exakt Null	<code>(1.0, -0.5, -0.5)</code>	Division durch Null
Summe nahe Null (positiv)	<code>(1.0, -0.9999, 0.0)</code>	Großes Ergebnis
Ein Parameter Null	<code>(0.0, 1.0, 1.0)</code>	Beeinflusst Summe nicht
Alle Parameter Null	<code>(0.0, 0.0, 0.0)</code>	Division durch Null

Array-Grenzen

Funktion: `array_sum(arr)` - Summe der Array-Elemente

Zwei Dimensionen von Grenzen!

Strukturelle Grenzen:

- Leeres Array (`len = 0`)
- Einzelnes Element (`len = 1`)
- Zwei Elemente (`len = 2`)

Wert-Grenzen:

- Alle Nullen (`[0.0, 0.0, 0.0]`)
- Enthält eine Null (`[0.0, 1.0, 2.0]`)
- Alternierende Vorzeichen nahe Null (`[1e-10, -1e-10, 1e-10]`)

Was ist mit großen Arrays?

Frage: Sollten wir mit 1.000.000 Elementen testen?

Kurze Antwort: Nein! Nutzt kleine Arrays (10-100 Elemente) in Unit Tests.

Warum keine riesigen Arrays?

- ✗ Langsame Tests (Sekunden statt Millisekunden)
- ✗ Findet nicht mehr Bugs (Logik ändert sich nicht mit Größe)
- ✗ Schwer zu debuggen ("Array-Mismatch bei Index 47.293")
- ✗ Speicherverbrauch (CI-Server könnten ausfallen)

Wann testet man große Arrays? Performance-Tests, nicht Unit Tests!

Komplexes Praxisbeispiel

Funktion: `find_intersection(x_road, y_road, angle_degrees, camera_x, camera_y)`

Zu testende Grenzen:

- **Winkel-Grenzen:** `-90°`, `0°`, `+90°`, `±180°`
- **Strukturell:** Leere Arrays, einzelner Punkt
- **Geometrisch:** Kamera über/unter Straße, Strahl parallel zur Straße
- **Werte:** `angle = 90.0` (vertikal), `angle = 0.0` (horizontal)
- **Edge Cases:** Keine Schnittpunkte, mehrere Schnittpunkte

Hier zählt sich Boundary Analysis wirklich aus!

Diskrete Funktionen: Schwellenwerte

Neuer Grenztyp: Diskrete Schwellenwerte

```
def calculate_grade(score: float) -> str:  
    if score >= 90: return "A"  
    elif score >= 80: return "B"  
    elif score >= 70: return "C"  
    elif score >= 60: return "D"  
    else: return "F"
```

Grenzen sind die Schwellenwerte: 60, 70, 80, 90

Test: 59.99, 60.0, 60.01, 69.99, 70.0, 70.01, ...

Schwellenwert Boundary Tests

```
def test_grade_boundary_just_below_D():  
    """Grenze: Knapp unter D-Schwelle (59.99)"""  
    assert calculate_grade(59.99) == "F"  
  
def test_grade_boundary_exactly_D():  
    """Grenze: Exakt an D-Schwelle (60.0)"""  
    assert calculate_grade(60.0) == "D"  
  
def test_grade_boundary_just_above_D():  
    """Grenze: Knapp über D-Schwelle (60.01)"""  
    assert calculate_grade(60.01) == "D"
```

Warum? Off-by-one Fehler bei `>=` vs `>` Vergleichen!

Mehrdimensionale Schwellenwerte

Funktion: `calculate_shipping_cost(weight, distance, express)`

```
def calculate_shipping_cost(weight: float, distance: int,
                             express: bool) -> float:
    base = 5.0
    if weight > 10: base += 10.0 # Schweres Paket
    if distance > 500: base += 15.0 # Lange Distanz
    if express: base *= 2 # Express-Multiplikator
    return base
```

Grenzen: `weight = 10`, `distance = 500`, `express = True/False`

Kombinationen: `8 Testfälle` (2^3 Grenzbedingungen)

Boundary Testing Strategie Zusammenfassung

1. Äquivalenzklassen identifizieren (Teil 1)

2. Grenzen zwischen Klassen finden

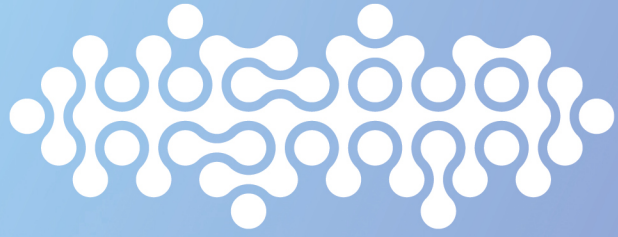
- Null, nahe Null, sehr groß
- Schwellenwerte in Bedingungen
- Strukturelle Grenzen (leer, eins, zwei)

3. An Grenzen testen, nicht nur in der Mitte

- 59.99 , 60.0 , 60.01 (nicht nur 65)

4. Sowohl endliche ALS AUCH nicht-endliche Werte testen

- `sys.float_info.max` UND `math.inf`



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

LLM-Assisted Testing

2

Warum schreiben Entwickler keine Tests?

Forschung zeigt:

1. **"Tests schreiben ist mühsam"** (42%)
 - Boilerplate, repetitive Struktur
2. **"Ich weiß nicht, was ich testen soll"** (38%)
 - Äquivalenzklassen? Grenzen?
3. **"Initiales Setup dauert ewig"** (35%)
 - pytest-Config, Teststruktur

Ergebnis: Prokrastination → Test Cone ⚠

Die Lösung: Mensch-LLM Kollaboration

LLMs für mühsame Teile nutzen, Menschen fokussieren auf Korrektheit

1. Mensch: Identifiziert, was getestet werden muss
2. LLM: Generiert Test-Boilerplate (Imports, Struktur)
3. Mensch: Überprüft und verfeinert (behebt Logik, fügt Edge Cases hinzu)
4. LLM: Generiert weitere Tests basierend auf Feedback
5. Mensch: Verifiziert, dass Assertions korrekt sind
6. Tests ausführen: Bugs im echten Code fangen!

Kernaussage: LLMs glänzen bei Boilerplate, Menschen bei Domänenlogik

Beispiel-Workflow: Schritt 1

LLM prompten:

```
# tests/test_geometry.py
import numpy as np
import pytest
from road_profile_viewer.geometry import find_intersection

# Prompt: "Write pytest unit tests for find_intersection() covering:
# - Equivalence classes: downward, horizontal, upward, vertical
# - Boundary cases: empty arrays, single point, angle=90
# - Use AAA pattern"
```

Schritt 2: LLM generiert Tests

```
def test_find_intersection_downward_angle() -> None:
    """Test mit normalem Abwärtswinkel."""
    x_road = np.array([0, 10, 20, 30], dtype=np.float64)
    y_road = np.array([0, 2, 4, 6], dtype=np.float64)
    x, y, dist = find_intersection(x_road, y_road, -10.0)
    assert x is not None
    assert dist > 0

def test_find_intersection_empty_road() -> None:
    """Test mit leeren Straßen-Arrays."""
    x_road = np.array([], dtype=np.float64)
    y_road = np.array([], dtype=np.float64)
    x, y, dist = find_intersection(x_road, y_road, -10.0)
    assert x is None
```

Nutzen: 80% des Boilerplates sofort geschrieben! ✨

Schritt 3: Menschliche Überprüfung - Fehler finden

✗ Problem 1: Fehlender Edge Case

```
# LLM hat nicht getestet: Was wenn Kamera UNTER der Straße?  
def test_find_intersection_camera_below_road() -> None:  
    x_road = np.array([0, 10, 20], dtype=np.float64)  
    y_road = np.array([5, 5, 5], dtype=np.float64) # Straße bei y=5  
    x, y, dist = find_intersection(x_road, y_road, -10.0,  
                                   camera_x=0.0, camera_y=0.0)  
    assert x is not None # Sollte trotzdem Schnittpunkt finden!
```

Menschliche Überprüfung (Fortsetzung)

✗ Problem 2: Schwache Assertions

```
# LLM: assert x is not None # Zu schwach!  
# Besser: assert 0 <= x <= 30, f"Expected x in [0, 30]"
```

✗ Problem 3: Ungültige Testdaten

```
# Horizontaler Strahl von y=1 schneidet Straße bei y=0 nicht!  
# Fix: Kamera anheben oder Straße neigen
```

Schritt 5: Tests ausführen → Echte Bugs finden!

```
$ uv run pytest tests/test_geometry.py -v
```

Überraschung! Ein Test schlägt fehl:

```
FAILED test_find_intersection_empty_road  
IndexError: index 0 is out of bounds
```

Das ist GUT! Der Test hat einen Bug in eurem echten Code gefunden:

```
def find_intersection(x_road, y_road, ...):  
    # Bug: Prüft nicht, ob Arrays leer sind!  
    for i in range(len(x_road) - 1): # Stürzt ab wenn len = 0  
        ...
```

Bug beheben

```
def find_intersection(x_road, y_road, ...):  
    # Validierung hinzufügen  
    if len(x_road) == 0 or len(y_road) == 0:  
        return None, None, None  
  
    # ... Rest der Funktion
```

Tests erneut ausführen:

```
$ uv run pytest tests/test_geometry.py -v  
===== 8 passed in 0.12s =====
```

✅ **Alle Tests bestehen!** Bug behoben, bevor er Nutzer erreicht.

Worin LLMs gut sind

- ✓ **Boilerplate** (Imports, Teststruktur, AAA-Pattern)
- ✓ **Standardmuster** (Rückgabewerte testen, grundlegende Assertions)
- ✓ **Coverage** (Tests für jede Funktion generieren)

Beispiel: 20 Test-Funktionssignaturen mit Docstrings schreiben

Worin LLMs schlecht sind

- ✗ **Domänenwissen** (Korrektheit des Verhaltens)
- ✗ **Problemspezifische Edge Cases**
- ✗ **Subtile Bugs in Testlogik**
- ✗ **Korrekte erwartete Werte bestimmen**
- ✗ **Unnötige Logik zu Tests hinzufügen**

Warnung: LLMs lieben Mocking

Mocking: Echte Objekte durch Fakes ersetzen, um Funktionsaufrufe zu verifizieren

LLMs schlagen oft das vor:

```
# ✗ LLM-generierter Test mit exzessivem Mocking
from unittest.mock import patch

def test_find_intersection_calls_tan():
    with patch('numpy.tan') as mock_tan:
        mock_tan.return_value = 0.176
        x, y, dist = find_intersection(x_road, y_road, -10.0)
        mock_tan.assert_called_once() # ✗ Testet WIE, nicht WAS
```

Problem: Bricht, wenn ihr die Implementierung ändert, obwohl Verhalten korrekt ist!

Googles Testing-Prinzip

Zustand testen, nicht Interaktionen

Zwei Wege zu prüfen, ob Code funktioniert:

1. **State Testing:** System **nach** Aufruf beobachten (Ergebnisse prüfen)
2. **Interaction Testing:** Funktionsaufrufe verifizieren (mit Mocks)

State Testing ist weniger fragil - fokussiert auf **was** passiert ist, nicht **wie**

Besserer Ansatz: Ergebnisse testen

```
# ✅ ERGEBNIS (Zustand) testen, nicht Methodenaufrufe
def test_find_intersection_returns_correct_intersection():
    """Test dass Schnittpunktposition korrekt ist"""
    x_road = np.array([0, 10, 20], dtype=np.float64)
    y_road = np.array([0, 2, 4], dtype=np.float64)

    x, y, dist = find_intersection(x_road, y_road, -10.0)

    # ERGEBNISSE assertieren
    assert x is not None
    assert 0 <= x <= 20, f"Schnittpunkt sollte in Grenzen sein"
    assert y >= 0
    assert dist > 0
    # Keine Annahmen über aufgerufene numpy-Funktionen!
```

Wann IST Mocking angemessen?

✓ **Mockt:**

- Externe APIs, Datenbanken
- File I/O
- Zeit/Zufall
- Teure Operationen

✗ **Mockt NICHT:**

- Pure Funktionen
- Einfache Datenstrukturen
- Implementierungsdetails
- Math-Bibliotheken

LLM-Fehler: Logik in Tests

```
#  LLM könnte generieren:  
def test_find_intersection_multiple_angles():  
    angles = [-10, -20, -30, -45]  
    for angle in angles: #  Schleife!  
        x, y, dist = find_intersection(x_road, y_road, angle)  
        if x is not None: #  Bedingung!  
            assert dist > 0
```

Probleme: Welcher Winkel ist fehlgeschlagen? Test testet vielleicht nichts!



Besser: Einfache Tests

```
# ✅ pytest parametrize nutzen
@pytest.mark.parametrize("angle", [-10, -20, -30, -45])
def test_find_intersection_downward_angles(angle):
    """Test dass alle Abwärtswinkel positive Distanz zurückgeben"""
    x_road = np.array([0, 10, 20], dtype=np.float64)
    y_road = np.array([0, 2, 4], dtype=np.float64)
    x, y, dist = find_intersection(x_road, y_road, angle)
    assert dist > 0 # Einfach! pytest läuft 4-mal
```

Vorteile: Klar welcher Winkel fehlschlug, keine Logik, leicht zu debuggen

Kernprinzip: Tests sollten geradlinig sein

- ✓ **TUT:** Einfachen, linearen Testcode schreiben
 - ✓ **TUT:** `@pytest.mark.parametrize` für mehrere Eingaben nutzen
 - ✗ **TUT NICHT:** Schleifen in Tests schreiben
 - ✗ **TUT NICHT:** Bedingungen in Tests schreiben
 - ✗ **TUT NICHT:** Erwartete Werte berechnen (hartkodierte Werte nutzen)
- Warum?** Wenn euer Test Logik hat, braucht ihr Tests für eure Tests!

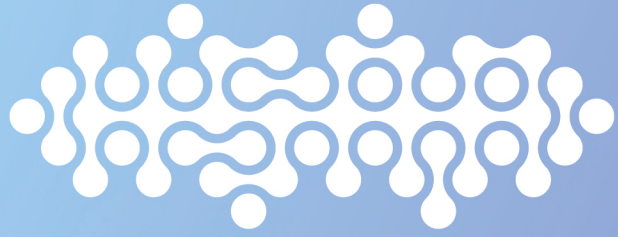


Interaktive Übung

Welcher Test ist besser?

- A) Test mit Schleife über 10 Winkel
- B) 10 separate Tests, einer pro Winkel
- C) Einzelner Test mit `@pytest.mark.parametrize` für 10 Winkel

Diskutiert 2 Minuten mit eurem Nachbarn!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Integration Testing

3

Was sind Integrationstests?

Unit Test: Testet eine Funktion isoliert

Integrationstest: Testet mehrere Module zusammenarbeitend

```
# Unit Test: Testet NUR geometry.find_intersection()
def test_find_intersection():
    x_road = np.array([0, 10])
    y_road = np.array([0, 2])
    x, y, dist = find_intersection(x_road, y_road, -10.0)
    assert x is not None
```

Integrationstest Beispiel

```
# Integrationstest: Testet geometry + road zusammen
def test_road_generation_with_intersection() -> None:
    """Test dass Straßengenerierung Daten erzeugt, die geometry verarbeiten kann."""
    from road_profile_viewer.road import generate_road_profile
    from road_profile_viewer.geometry import find_intersection

    # Straße mit road.py generieren
    x_road, y_road = generate_road_profile(num_points=100)

    # geometry.py nutzen, um Schnittpunkt zu finden
    x, y, dist = find_intersection(x_road, y_road, -10.0)

    # Integration verifizieren
    assert x is not None
    assert 0 <= x <= 80
```

Kernunterschied

Unit Test: Mock/gefälschte Daten (`np.array([0, 10])`)

Integrationstest: Echte Daten von echten Modulen (`generate_road_profile()`)

Warum beides?

- Unit Tests: Schnell, fokussiert, finden Logikfehler
- Integrationstests: Fangen Interface-Mismatches, Datenformatprobleme

Erinnert euch an die Testing Pyramid: 70% Unit, 20% Integration, 10% E2E

Wann Integrationstests schreiben

Fangt diese Probleme:

1. Interface Mismatches

- Funktionssignatur-Änderungen

2. Datenformat-Probleme

- `list` vs `np.array`

3. Annahmen über Daten

- Sortiert vs unsortiert
- Typ-Mismatches

Integrationstest Suite Beispiel

```
# tests/test_integration.py
class TestRoadGeometryIntegration:
    def test_road_data_format_compatible(self) -> None:
        """Verifiziere dass road.py Format zurückgibt, das geometry.py erwartet."""
        x_road, y_road = generate_road_profile()

        # Datentypen prüfen
        assert isinstance(x_road, np.ndarray)
        assert isinstance(y_road, np.ndarray)

        # Sortierung prüfen
        assert np.all(np.diff(x_road) > 0)
```

Warum wir E2E Testing überspringen (Vorerst)

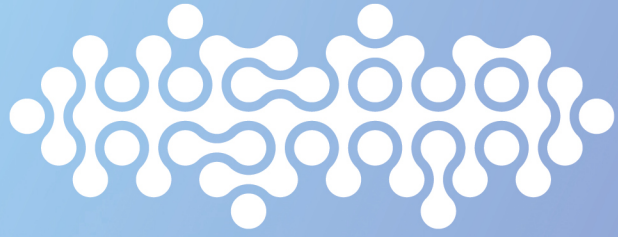
End-to-End Test würde bedeuten:

1. Dash-Anwendung starten
2. Browser öffnen (Selenium/Playwright)
3. Winkel in Eingabefeld eingeben
4. Graph-Update verifizieren

Warum überspringen?

- Komplexes Setup (Selenium, Browser-Treiber)
- Langsam (Sekunden pro Test)
- Fragil (UI-Änderungen brechen Tests)
- Abnehmende Erträge (Unit + Integration fangen 90% der Bugs)

Für diesen Kurs: Meistert zuerst Unit + Integration!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Test Maintainability

4

Das Fragilitätsproblem

Szenario:

Ihr schreibt 20 Unit Tests. Alle bestehen. ✓

Ihr refaktoriert `find_intersection()` für Performance (keine Verhaltensänderung).

Plötzlich schlagen 10 Tests fehl. ✗

Frage: Ist das gut oder schlecht?

Antwort: SCHLECHT!

Fragile Tests schlagen fehl, wenn sich Implementierung ändert, obwohl Verhalten gleich blieb.

Probleme:

- Erzwingen wiederholtes Test-Anpassen
- Verbrauchen Wartungszeit
- Machen Angst vorm Refaktorisieren
- Skalieren schlecht

Googles Einsicht: Große Produktivitätskiller!

Fragiler Test Beispiel

```
#  FRAGIL: Testet WIE die Funktion intern arbeitet
from unittest.mock import patch

def test_find_intersection_uses_tan():
    """Test dass Funktion np.tan nutzt"""
    with patch('numpy.tan') as mock_tan:
        mock_tan.return_value = 0.176
        find_intersection(x_road, y_road, -10.0)
        assert mock_tan.called #  Bricht bei Refaktorisierung!
```

Problem: Wenn ihr zu `atan2` oder Lookup-Tabellen wechselt, schlägt Test fehl, obwohl:

- Schnittpunktposition noch korrekt ist
- Verhalten identisch ist
- Keine Bugs eingeführt wurden



Robuster Test Beispiel

```
# ✓ ROBUST: Testet WAS der Code tut, nicht WIE
def test_find_intersection_returns_correct_position():
    """Test dass Schnittpunktposition geometrisch korrekt ist"""
    # Arrange
    x_road = np.array([0, 10, 20], dtype=np.float64)
    y_road = np.array([0, 2, 4], dtype=np.float64)

    # Act
    x, y, dist = find_intersection(x_road, y_road, -10.0)

    # Assert Verhalten
    assert x is not None
    assert 0 <= x <= 20, f"x sollte in Grenzen sein, ist {x}"
    assert dist > 0
    # Keine Annahmen über WIE es berechnet wurde!
```

Kernprinzip: Via Public APIs testen

Googles Richtlinie:

"Schreibt Tests, die das System auf gleiche Weise aufrufen wie dessen Nutzer."

Public API (was Nutzer sehen):

```
x, y, dist = find_intersection(x_road, y_road, angle)
```

Privat (was Nutzer nicht sehen): numpy-Aufrufe, Variablen, Helfer

Testet die API, nicht die Implementierung!

Googles vier Kategorien von Code-Änderungen

Kategorie	Implementierung ändert?	Verhalten ändert?	Tests brechen?
Pure Refactoring	Ja	Nein	✗ Sollten NICHT brechen
New Feature	Ja	Ja (fügt hinzu)	✓ Neue Tests hinzugefügt
Bug Fix	Ja	Ja (behebt)	✓ Vorher bestandener Test fängt jetzt Bug
Behavior Change	Vielleicht	Ja	✓ Tests aktualisiert

Ziel: Tests sollten nur in Kategorien 2-4 brechen, NICHT Kategorie 1!

Praktische Richtlinien

✓ TUT:

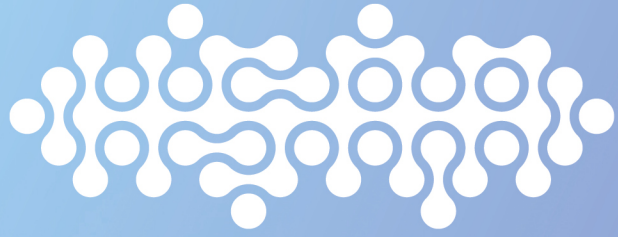
- Via Public APIs testen
- Echte Daten für pure Funktionen nutzen
- Endergebnisse prüfen
- Erwartete Werte hartkodieren

✗ TUT NICHT:

- Implementierungsdetails mocken
- Funktionsaufrufe verifizieren
- Zwischenwerte testen
- Komplexe Logik in Tests nutzen

Wartbare vs fragile Tests

Aspekt	Wartbarer Test	Fragiler Test
Testet	Öffentliches Verhalten	Interne Implementierung
Nutzt	Echte Daten	Exzessive Mocks
Prüft	Endergebnisse	Methodenaufrufe
Überlebt	Refaktorisierung	Nur exakte Implementierung
Schlägt fehl bei	Verhaltensänderung	Implementierungsänderung



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Feature Branch Workflow

5

Workflow: Tests hinzufügen

Tests sind auch ein Feature! Nutzt den gleichen Workflow:

```
# 1. Feature Branch erstellen
git checkout -b add-geometry-tests

# 2. Testdatei erstellen
# tests/test_geometry.py

# 3. Tests schreiben (mit LLM-Unterstützung!)

# 4. Tests ausführen
uv run pytest tests/test_geometry.py -v
```

Workflow (Fortsetzung)

5. Von Tests gefundene Bugs beheben

6. Commit

```
git add tests/test_geometry.py src/road_profile_viewer/geometry.py  
git commit -m "Umfassende Tests für Geometrie-Modul hinzufügen"
```

- 18 Unit Tests für Äquivalenzklassen
- Boundary Tests für Edge Cases
- Integrationstest mit Road-Modul"

7. Pushen und PR erstellen

```
git push -u origin add-geometry-tests  
gh pr create --title "Geometrie-Tests hinzufügen"
```

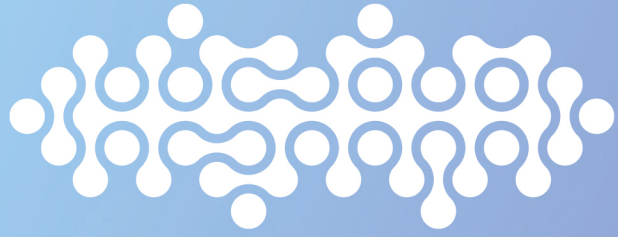
Nächste Vorlesung: CI wird Tests ausführen!

In **Kapitel 03 (TDD und CI)** konfigurieren wir CI für:

- Automatisches Ausführen von Tests bei jedem PR
- Merge blockieren wenn Tests fehlschlagen
- Test-Coverage anzeigen

Vorschau:

```
# .github/workflows/ci.yml
- name: Run tests
  run: uv run pytest tests/ -v
```



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Zusammenfassung

6

Vor dieser Vorlesung

- ✓ Ruff (Style)
- ✓ Pyright (Typen)
- ✓ Grundlegende Unit Testing Konzepte
- ✗ Kein systematisches Boundary Testing
- ✗ Kein LLM-unterstützter Workflow
- ✗ Tests manuell schreiben (mühsam!)

Nach dieser Vorlesung

- ✓ Ruff (Style)
- ✓ Pyright (Typen)
- ✓ Systematische Boundary Value Analysis
- ✓ LLM-unterstützte Testgenerierung
- ✓ Integration Testing
- ✓ Wartbare Test-Praktiken
- ✓ Feature Branch Workflow für Tests

Gemeisterte Kernkonzepte

1. Boundary Value Analysis

- An Grenzen testen (endlich UND nicht-endlich)
- Schwellenwerte in diskreten Funktionen

2. LLM-Assisted Testing

- LLMs für Boilerplate, Menschen verifizieren
- Ergebnisse testen, nicht Methodenaufrufe

3. Integration & Maintainability

- Module zusammen testen
- Verhalten testen, Refaktorisierung überleben

Der Unterschied, den Tests machen

Ohne Tests:

```
Entwickler: *Ändert Code*  
Entwickler: *Klickt manuell UI*  
Entwickler: "Sieht gut aus!" *Pusht*  
Nutzer: "App abgestürzt!" 🌟
```

Mit Tests:

```
Entwickler: *Ändert Code*  
Entwickler: $ pytest tests/  
Entwickler: "❌ Fehlgeschlagen! Bug gefangen"  
Entwickler: *Behebt, Tests bestehen*  
Nutzer: "Alles funktioniert!" ✅
```

Kernaussagen

1. **Qualität $\neq$ Korrektheit** - Style vs Bugs
2. **Bugs verstecken sich an Grenzen** - Systematisch testen
3. **LLMs unterstützen, Menschen verifizieren** - Beste von beiden
4. **Zustand testen, nicht Interaktionen** - Mocking vermeiden
5. **Refaktorisierung überleben** - Verhalten testen
6. **Schnell = häufig** - Millisekunden zählen
7. **Tests geben Vertrauen** - Furchtlos refaktorisieren

Praktische Übung

Eure Aufgabe:

Fügt umfassende Tests zu `geometry.py` in eurem Road Profile Viewer hinzu:

1. LLM nutzen, um Test-Boilerplate zu generieren
2. Boundary Tests für `find_intersection()` hinzufügen
3. Integrationstest mit `road.py` schreiben
4. Tests ausführen: `uv run pytest tests/ -v`
5. Gefundene Bugs beheben
6. Zu Feature Branch pushen

GitHub Classroom Aufgabe wird geteilt!

Vorschau nächste Vorlesung

Kapitel 03 (TDD und CI): Test-Driven Development & CI Integration

- Tests VOR Code schreiben (TDD)
- Red-Green-Refactor Zyklus
- Tests in CI integrieren
- Fehlschlagende Tests blockieren Merges
- Test Coverage Reports

Ihr seid jetzt bereit für professionelle Testing-Praktiken! 🚀

Weiterführende Literatur

Zu Testing:

- Kent Beck's "Test-Driven Development by Example"
- Martin Fowler's "Testing Strategies"
- pytest Docs: <https://docs.pytest.org/>

Zu Test Maintainability:

- Google Testing Blog: "Testing on the Toilet" Serie
- "Software Engineering at Google" (Kapitel 12: Unit Testing)

Zu Boundary Testing:

- Software Testing Fundamentals: Equivalence Partitioning
- IEEE 754 Standard: <https://standards.ieee.org/>



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Fragen?

VIELEN DANK