



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Chapter 03

TESTING FUNDAMENTALS

Boundary Analysis and LLM-Assisted Testing

Welcome Back!

In **Part 1** you learned:

- **Testing Pyramid** (70% unit, 20% integration, 10% E2E)
- **AAA Pattern** (Arrange-Act-Assert)
- **Equivalence Classes** (grouping inputs by behavior)
- **pytest basics** (writing and running tests)

But we left you with questions...

The Challenge From Part 1

You're probably thinking:

- 🤔 "How do I know if I've tested **enough**?"
- 🤔 "What about those tricky **edge cases** at boundaries?"
- 🤔 "Writing 20-30 tests for one function sounds **tedious**!"
- 🤔 "Can I use **AI** to help me write tests faster?"

Today we solve these problems!

Today's Mission

Three Powerful Techniques

1. **Boundary Value Analysis**

- Where bugs actually hide (not in the middle!)

2. **LLM-Assisted Testing**

- Break the "test cone" with AI assistance

3. **Integration Testing & Maintainability**

- Make testing part of your workflow

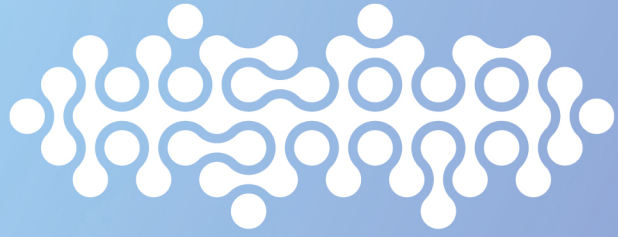
Learning Objectives

By the end of this lecture, you will:

- ✓ **Identify** boundary values systematically for any function
- ✓ **Use** LLMs to generate test boilerplate (while avoiding AI pitfalls)
- ✓ **Write** integration tests that verify modules work together
- ✓ **Apply** feature branch workflow to add tests to your project
- ✓ **Know** when you have "enough" tests (realistic coverage, not perfection)

The Road Ahead

- ✓ Part 1: Foundation
Why test? How to write basic unit tests?
- Part 2: Mastery (TODAY!)
Where do bugs hide? How to test efficiently?
- Chapter 03 (TDD and CI): TDD & CI
Write tests FIRST, automate everything



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Boundary Value Analysis

1

Key Observation

Bugs often lurk at the boundaries between equivalence classes

Why boundaries?

- Off-by-one errors
- Floating-point precision issues
- Edge cases in conditional logic (`if x < 0` , `if x == 0` , `if x > 0`)

Strategy: Test at the boundaries, not just in the middle!

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

Recall from Part 1: We identified 5 equivalence classes

Class	Representative
Positive	<code>x = 5.0</code>
Negative	<code>x = -3.0</code>
Zero	<code>x = 0.0</code>
Very large	<code>x = 1e100</code>
Special values	<code>inf</code> , <code>nan</code>

Question: Where are the **boundaries** between these classes?

Boundaries for `reciprocal(x)`

Equivalence Class	Boundary Values to Test
Positive numbers	<code>x = 0.0001</code> (near zero), <code>x = 1.0</code> , <code>x = 1000000.0</code>
Negative numbers	<code>x = -0.0001</code> , <code>x = -1.0</code> , <code>x = -1000000.0</code>
Zero	<code>x = 0.0</code> , <code>x = 1e-100</code>
Extreme boundaries	<code>sys.float_info.max</code> , <code>sys.float_info.min</code> , <code>float('inf')</code> , <code>float('nan')</code>

Key insight: Test near zero, at zero, and at extreme values!

Python Float Boundaries (IEEE 754)

Python floats are **64-bit IEEE 754 doubles**

Python Constant	Typical Value	Description
<code>sys.float_info.max</code>	$\approx 1.8e308$	Largest finite float
<code>sys.float_info.min</code>	$\approx 2.2e-308$	Smallest positive normalized
<code>sys.float_info.epsilon</code>	$\approx 2.2e-16$	Machine epsilon
<code>math.inf</code>	<code>inf</code>	Positive infinity (non-finite)
<code>math.nan</code>	<code>nan</code>	Not a Number

Critical Distinction

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

They test different things!

`math.inf` - Special non-finite value

- Tests infinity handling
- `1 / inf = 0.0`

`sys.float_info.max` - Largest finite float

- Tests finite range limits
- `1 / sys.float_info.max ≈ 5.6e-309`

Comprehensive Boundary Test Suite

```
import sys, math, pytest

# Define constants for readability
INF = math.inf
FMAX = sys.float_info.max
FMIN_NORM_POS = sys.float_info.min

def test_reciprocal_boundary_near_zero_positive():
    """Boundary: Very small positive"""
    result = reciprocal(1e-6)
    assert result == pytest.approx(1e6, rel=1e-9)

def test_reciprocal_boundary_exactly_zero():
    """Boundary: Exactly zero"""
    with pytest.raises(ZeroDivisionError):
        reciprocal(0.0)
```

Testing Finite Boundaries

```
def test_reciprocal_boundary_max_finite_float():
    """Boundary: Largest finite (sys.float_info.max ≈ 1.8e308)"""
    result = reciprocal(FMAX)
    assert result > 0
    assert result < FMIN_NORM_POS # Underflows to subnormal

def test_reciprocal_boundary_min_normalized_float():
    """Boundary: Smallest positive normalized"""
    result = reciprocal(FMIN_NORM_POS)
    assert result == INF # Overflows to infinity!
```

Key: Test both finite limits AND infinity!

Testing Non-Finite Boundaries

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

Remember: `nan == nan` is always `False`! Use `math.isnan()`.



Think About It

Which test is more important?

A) Testing `reciprocal(5.0)` (middle of equivalence class)

B) Testing `reciprocal(0.0001)` (near-zero boundary)

Discuss with your neighbor for 1 minute!

Multi-Parameter Boundaries

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

New complexity: Boundaries now include **combinations!**

Boundary Condition	Test Values	Why Important
Sum exactly zero	(1.0, -0.5, -0.5)	Division by zero
Sum near zero (positive)	(1.0, -0.9999, 0.0)	Large result
One parameter zero	(0.0, 1.0, 1.0)	Doesn't affect sum
All parameters zero	(0.0, 0.0, 0.0)	Division by zero

Array Boundaries

Function: `array_sum(arr)` - sum of array elements

Two dimensions of boundaries!

Structural boundaries:

- Empty array (`len = 0`)
- Single element (`len = 1`)
- Two elements (`len = 2`)

Value boundaries:

- All zeros (`[0.0, 0.0, 0.0]`)
- Contains one zero (`[0.0, 1.0, 2.0]`)
- Alternating signs near zero (`[1e-10, -1e-10, 1e-10]`)

What About Large Arrays?

Question: Should we test with 1,000,000 elements?

Short Answer: No! Use small arrays (10-100 elements) in unit tests.

Why not huge arrays?

-  Slow tests (seconds instead of milliseconds)
-  Doesn't find more bugs (logic doesn't change with size)
-  Hard to debug ("Array mismatch at index 47,293")
-  Memory usage (CI servers might run out)

When DO you test large arrays? Performance tests, not unit tests!

Complex Real-World Example

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

Boundaries to test:

- **Angle boundaries:** `-90°`, `0°`, `+90°`, `±180°`
- **Structural:** Empty arrays, single point
- **Geometric:** Camera above/below road, ray parallel to road
- **Value:** `angle = 90.0` (vertical), `angle = 0.0` (horizontal)
- **Edge cases:** No intersection, multiple intersections

This is where boundary analysis really pays off!

Discrete Functions: Thresholds

New type of boundary: Discrete thresholds

```
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"
```

Boundaries are the thresholds: 60, 70, 80, 90

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

Threshold Boundary Tests

```
def test_grade_boundary_just_below_D():  
    """Boundary: Just below D threshold (59.99)"""  
    assert calculate_grade(59.99) == "F"  
  
def test_grade_boundary_exactly_D():  
    """Boundary: Exactly at D threshold (60.0)"""  
    assert calculate_grade(60.0) == "D"  
  
def test_grade_boundary_just_above_D():  
    """Boundary: Just above D threshold (60.01)"""  
    assert calculate_grade(60.01) == "D"
```

Why? Off-by-one errors in `>=` vs `>` comparisons!

Multi-Dimensional Thresholds

Function: `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 # Heavy package
    if distance > 500: base += 15.0 # Long distance
    if express: base *= 2 # Express multiplier
    return base
```

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

Combinations: 8 test cases (2^3 boundary conditions)

Boundary Testing Strategy Summary

1. Identify equivalence classes (Part 1)

2. Find boundaries between classes

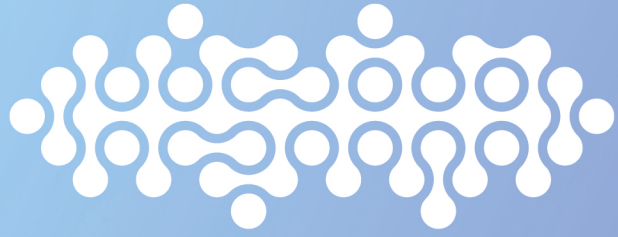
- Zero, near-zero, very large
- Thresholds in conditionals
- Structural boundaries (empty, single, two)

3. Test at boundaries, not just in middle

- `59.99` , `60.0` , `60.01` (not just `65`)

4. Test both finite AND non-finite values

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



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

LLM-Assisted Testing

2

Why Don't Developers Write Tests?

Research shows:

1. **"Writing tests is tedious"** (42%)
 - Boilerplate, repetitive structure
2. **"I don't know what to test"** (38%)
 - Equivalence classes? Boundaries?
3. **"Initial setup takes forever"** (35%)
 - pytest config, test structure

Result: Procrastination → Test cone ⚠

The Solution: Human-LLM Collaboration

Use LLMs to handle tedious parts, humans focus on correctness

1. Human: Identifies what needs testing
2. LLM: Generates test boilerplate (imports, structure)
3. Human: Reviews and refines (fixes logic, adds edge cases)
4. LLM: Generates more tests based on feedback
5. Human: Verifies assertions are correct
6. Run tests: Catch bugs in actual code!

Key insight: LLMs excel at boilerplate, humans excel at domain logic

Example Workflow: Step 1

Prompt the LLM:

```
# 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"
```

Step 2: LLM Generates Tests

```
def test_find_intersection_downward_angle() -> None:
    """Test with normal downward angle."""
    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 with empty road 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
```

Value: 80% of boilerplate written instantly! ✨

Step 3: Human Review - Find Flaws

✗ Problem 1: Missing edge case

```
# LLM didn't test: What if camera is BELOW the road?
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) # Road at 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 # Should still find intersection!
```

Human Review (continued)

✗ Problem 2: Weak assertions

```
# LLM: assert x is not None # Too weak!  
# Better: assert 0 <= x <= 30, f"Expected x in [0, 30]"
```

✗ Problem 3: Invalid test data

```
# Horizontal ray from y=1 won't intersect road at y=0!  
# Fix: Raise camera or tilt road
```

Step 5: Run Tests → Find Real Bugs!

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

Surprise! One test fails:

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

This is **GOOD!** Test found a bug in your actual code:

```
def find_intersection(x_road, y_road, ...):  
    # Bug: Doesn't check if arrays are empty!  
    for i in range(len(x_road) - 1): # Crashes if len = 0  
        ...
```

Fix the Bug

```
def find_intersection(x_road, y_road, ...):  
    # Add validation  
    if len(x_road) == 0 or len(y_road) == 0:  
        return None, None, None  
  
    # ... rest of function
```

Run tests again:

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

✓ **All tests pass!** Bug fixed before it reached users.

What LLMs Are Good At

- ✓ **Boilerplate** (imports, test structure, AAA pattern)
- ✓ **Standard patterns** (testing return values, basic assertions)
- ✓ **Coverage** (generating tests for each function)

Example: Writing 20 test function signatures with docstrings

What LLMs Are Bad At

- ✗ **Domain knowledge** (correctness of behavior)
- ✗ **Problem-specific edge cases**
- ✗ **Subtle bugs in test logic**
- ✗ **Determining correct expected values**
- ✗ **Adding unnecessary logic to tests**

Warning: LLMs Love Mocking

Mocking: Replacing real objects with fakes to verify function calls

LLMs often suggest this:

```
# ❌ LLM-generated test with excessive 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() # ❌ Tests HOW, not WHAT
```

Problem: Breaks if you change implementation, even though behavior is correct!

Google's Testing Principle

Test State, Not Interactions

Two ways to verify code works:

1. **State Testing:** Observe system **after** invoking it (check results)
2. **Interaction Testing:** Verify function calls (using mocks)

State testing is less brittle - focuses on **what** results occurred, not **how**

Better Approach: Test Results

```
# ✅ Test the RESULT (state), not method calls
def test_find_intersection_returns_correct_intersection():
    """Test that intersection position is correct"""
    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)

    # Assert OUTCOMES
    assert x is not None
    assert 0 <= x <= 20, f"Intersection should be in bounds"
    assert y >= 0
    assert dist > 0
    # No assumptions about which numpy functions were called!
```

When IS Mocking Appropriate?

✓ **DO mock:**

- External APIs, databases
- File I/O
- Time/Randomness
- Expensive operations

✗ **DON'T mock:**

- Pure functions
- Simple data structures
- Implementation details
- Math libraries

LLM Failure: Logic in Tests

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

Problems: Which angle failed? Test might not test anything!



Better: Simple Tests

```
# ✓ Use pytest parametrize
@pytest.mark.parametrize("angle", [-10, -20, -30, -45])
def test_find_intersection_downward_angles(angle):
    """Test that all downward angles return positive distance"""
    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 # Simple! pytest runs 4 times
```

Benefits: Clear which angle failed, no logic, easy to debug

Key Principle: Tests Should Be Straight-Line

- ✓ **DO:** Write simple, linear test code
- ✓ **DO:** Use `@pytest.mark.parametrize` for multiple inputs
- ✗ **DON'T:** Write loops in tests
- ✗ **DON'T:** Write conditionals in tests
- ✗ **DON'T:** Calculate expected values (use hardcoded values)

Why? If your test has logic, you need tests for your tests!

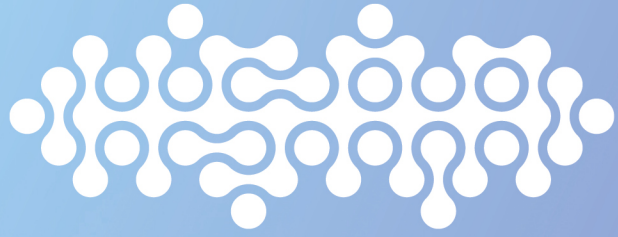


Interactive Exercise

Which test is better?

- A)** Test with loop iterating over 10 angles
- B)** 10 separate tests, one per angle
- C)** Single test with `@pytest.mark.parametrize` for 10 angles

Discuss with your neighbor for 2 minutes!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Integration Testing

3

What Are Integration Tests?

Unit test: Tests one function in isolation

Integration test: Tests multiple modules working together

```
# Unit test: Test ONLY 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
```

Integration Test Example

```
# Integration test: Test geometry + road together
def test_road_generation_with_intersection() -> None:
    """Test road generation produces data geometry can process."""
    from road_profile_viewer.road import generate_road_profile
    from road_profile_viewer.geometry import find_intersection

    # Generate road using road.py
    x_road, y_road = generate_road_profile(num_points=100)

    # Use geometry.py to find intersection
    x, y, dist = find_intersection(x_road, y_road, -10.0)

    # Verify integration works
    assert x is not None
    assert 0 <= x <= 80
```

Key Difference

Unit test: Mock/fake data (`np.array([0, 10])`)

Integration test: Real data from actual modules (`generate_road_profile()`)

Why both?

- Unit tests: Fast, focused, find logic bugs
- Integration tests: Catch interface mismatches, data format issues

Remember the testing pyramid: 70% unit, 20% integration, 10% E2E

When to Write Integration Tests

Catch these issues:

1. Interface mismatches

- Function signature changes

2. Data format issues

- `list` vs `np.array`

3. Assumptions about data

- Sorted vs unsorted
- Type mismatches

Integration Test Suite Example

```
# tests/test_integration.py
class TestRoadGeometryIntegration:
    def test_road_data_format_compatible(self) -> None:
        """Verify road.py returns format geometry.py expects."""
        x_road, y_road = generate_road_profile()

        # Check data types
        assert isinstance(x_road, np.ndarray)
        assert isinstance(y_road, np.ndarray)

        # Check sorted
        assert np.all(np.diff(x_road) > 0)
```

Why We Skip E2E Testing (For Now)

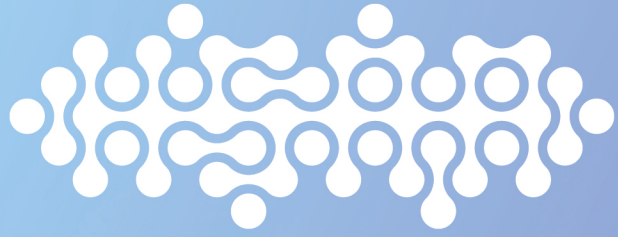
End-to-End test would mean:

1. Start Dash application
2. Open browser (Selenium/Playwright)
3. Enter angle in input field
4. Verify graph updates

Why skip?

- Complex setup (Selenium, browser drivers)
- Slow (seconds per test)
- Brittle (UI changes break tests)
- Diminishing returns (unit + integration catch 90% of bugs)

For this course: Master unit + integration first!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Test Maintainability

4

The Brittleness Problem

Scenario:

You write 20 unit tests. They all pass. ✓

You refactor `find_intersection()` for performance (no behavior change).

Suddenly 10 tests fail. ✗

Question: Is this good or bad?

Answer: BAD!

Brittle tests fail when implementation changes, even though behavior stayed the same.

Problems:

- Force repeated test tweaking
- Consume maintenance time
- Make you afraid to refactor
- Scale poorly

Google's insight: Major productivity killer!

Brittle Test Example

```
#  BRITTLE: Tests HOW the function works internally
from unittest.mock import patch

def test_find_intersection_uses_tan():
    """Test that function uses np.tan"""
    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 #  Breaks on refactoring!
```

Problem: If you switch to `atan2` or lookup tables, test fails even though:

- Intersection position is still correct
- Behavior is identical
- No bugs were introduced



Robust Test Example

```
# ✓ ROBUST: Tests WHAT the code does, not HOW
def test_find_intersection_returns_correct_position():
    """Test intersection position is geometrically correct"""
    # 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 behavior
    assert x is not None
    assert 0 <= x <= 20, f"x should be in bounds, got {x}"
    assert dist > 0
    # No assumptions about HOW it calculated this!
```

Key Principle: Test Via Public APIs

Google's guideline:

"Write tests that invoke the system the same way its users would."

Public API (what users see):

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

Private (what users don't see): numpy calls, variables, helpers

Test the API, not implementation!

Google's Four Categories of Code Changes

Category	Implementation Changes?	Behavior Changes?	Tests Break?
Pure Refactoring	Yes	No	✗ Should NOT break
New Feature	Yes	Yes (adds)	✓ New tests added
Bug Fix	Yes	Yes (fixes)	✓ Previously passing test now catches bug
Behavior Change	Maybe	Yes	✓ Tests updated

Goal: Tests should only break in categories 2-4, NOT category 1!

Practical Guidelines

✓ DO:

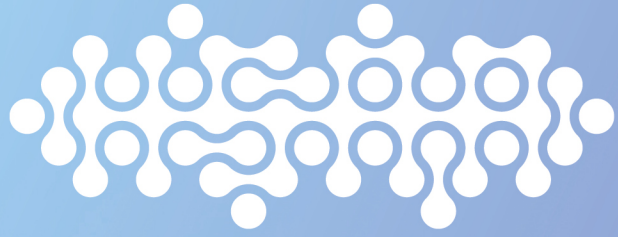
- Test via public APIs
- Use real data for pure functions
- Check final outcomes
- Hardcode expected values

✗ DON'T:

- Mock implementation details
- Verify function calls
- Test intermediate values
- Use complex logic in tests

Maintainable vs Brittle Tests

Aspect	Maintainable Test	Brittle Test
Tests	Public behavior	Internal implementation
Uses	Real data	Excessive mocks
Checks	Final outcomes	Method calls
Survives	Refactoring	Only exact implementation
Fails when	Behavior changes	Implementation changes



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Feature Branch Workflow

5

Workflow: Adding Tests

Tests are a feature too! Use the same workflow:

```
# 1. Create feature branch
git checkout -b add-geometry-tests

# 2. Create test file
# tests/test_geometry.py

# 3. Write tests (with LLM assistance!)

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

Workflow (continued)

5. Fix bugs found by tests

6. Commit

```
git add tests/test_geometry.py src/road_profile_viewer/geometry.py  
git commit -m "Add comprehensive tests for geometry module"
```

- 18 unit tests covering equivalence classes
- Boundary tests for edge cases
- Integration test with road module"

7. Push and create PR

```
git push -u origin add-geometry-tests  
gh pr create --title "Add geometry tests"
```

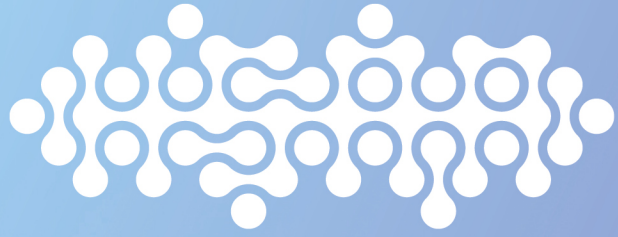
Next Lecture: CI Will Run Tests!

In **Chapter 03 (TDD and CI)**, we'll configure CI to:

- Run tests automatically on every PR
- Block merge if tests fail
- Show test coverage

Preview:

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



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Summary

6

Before This Lecture

- ✓ Ruff (style)
- ✓ Pyright (types)
- ✓ Basic unit testing concepts
- ✗ No systematic boundary testing
- ✗ No LLM-assisted workflow
- ✗ Writing tests manually (tedious!)

After This Lecture

- ✓ Ruff (style)
- ✓ Pyright (types)
- ✓ Systematic boundary value analysis
- ✓ LLM-assisted test generation
- ✓ Integration testing
- ✓ Maintainable test practices
- ✓ Feature branch workflow for tests

Key Concepts Mastered

1. Boundary Value Analysis

- Test at boundaries (finite AND non-finite)
- Thresholds in discrete functions

2. LLM-Assisted Testing

- LLMs for boilerplate, humans verify
- Test results, not method calls

3. Integration & Maintainability

- Test modules together
- Test behavior, survive refactoring

The Difference Tests Make

Without tests:

```
Developer: *Changes code*  
Developer: *Manually clicks UI*  
Developer: "Looks good!" *Pushes*  
User: "App crashed!" 🌟
```

With tests:

```
Developer: *Changes code*  
Developer: $ pytest tests/  
Developer: "❌ Failed! Bug caught"  
Developer: *Fixes, tests pass*  
User: "Everything works!" ✅
```

Key Takeaways

1. **Quality $\neq$ Correctness** - Style vs bugs
2. **Bugs hide at boundaries** - Test systematically
3. **LLMs assist, humans verify** - Best of both
4. **Test state, not interactions** - Avoid mocking
5. **Survive refactoring** - Test behavior
6. **Fast = frequent** - Milliseconds matter
7. **Tests give confidence** - Refactor fearlessly

Hands-On Exercise

Your Task:

Add comprehensive tests to `geometry.py` in your Road Profile Viewer:

1. Use LLM to generate test boilerplate
2. Add boundary tests for `find_intersection()`
3. Write integration test with `road.py`
4. Run tests: `uv run pytest tests/ -v`
5. Fix any bugs found
6. Push to feature branch

GitHub Classroom assignment will be shared!

Next Lecture Preview

Chapter 03 (TDD and CI): Test-Driven Development & CI Integration

- Write tests BEFORE code (TDD)
- Red-Green-Refactor cycle
- Integrate tests into CI
- Make failing tests block merges
- Test coverage reports

You're now ready for professional testing practices! 🚀

Further Reading

On Testing:

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

On Test Maintainability:

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

On Boundary Testing:

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



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

THANK YOU