



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Chapter 03

EQUIVALENCE CLASSES: SMART TESTING STRATEGIES

Where We Are

What we've learned so far:

-  Exhaustive testing is impossible (32 billion years!)
-  We need smart strategies to test efficiently
-  Clean test code principles (AAA pattern, descriptive names)

Today's question:

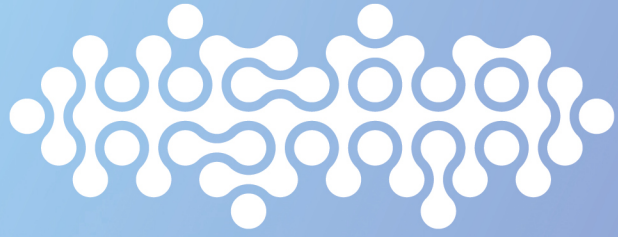
 *If we can't test everything, **how do we choose what to test?***

The Core Problem

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

Questions:

- How many possible inputs? **Infinite!** (all floats)
- How many tests can we write? **Finite** (maybe 5-10)
- How do we choose which 5-10 inputs to test? 🤔



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Equivalence Classes: The Core Idea

1

The Answer: Equivalence Classes

Key Insight:

Not all inputs are equally interesting. Many inputs produce **similar behavior**.

Definition (Accessible):

An **equivalence class** is a group of inputs that:

- Produce similar output behavior
- Test the same code path
- Have the same mathematical properties

Strategy:

Test **one representative** from each equivalence class instead of all possible inputs!

Key Properties

1. Disjoint (no overlap)

Each input belongs to **exactly one** equivalence class

2. Complete (cover everything)

Every possible input belongs to **some** equivalence class

3. Representative testing

Testing **one** value from a class is enough to test the whole class

Result: Testing 3 representatives instead of ∞ inputs! 🎯

Mathematical Foundation (Optional Deep Dive)

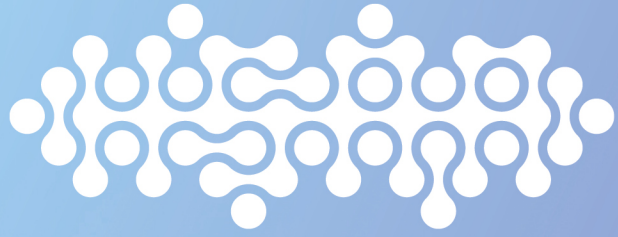
An **equivalence relation** $\sim$ on a set X satisfies:

1. **Reflexive:** $x \sim x$ (every element is equivalent to itself)
2. **Symmetric:** If $x \sim y$, then $y \sim x$
3. **Transitive:** If $x \sim y$ and $y \sim z$, then $x \sim z$

Example: For $\text{reciprocal}(x)$:

- Define: $x \sim y$ if $\text{sign}(x) = \text{sign}(y)$
- This creates 3 equivalence classes: positive, negative, zero

Don't worry if this feels abstract - we'll see concrete examples next!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Numeric Examples

2

Example 1: reciprocal(x) - The Simplest Case

Interactive Question:

🤔 Looking at this code, how many **distinct behaviors** do you see?

Think about:

- What happens when `x > 0` ?
- What happens when `x < 0` ?
- What happens when `x == 0` ?

reciprocal(x): Three Equivalence Classes

Positive numbers	<code>x > 0</code>	<code>x = 5.0</code>	Returns positive reciprocal (<code>0.2</code>)
Negative numbers	<code>x < 0</code>	<code>x = -5.0</code>	Returns negative reciprocal (<code>-0.2</code>)
Zero	<code>x == 0</code>	<code>x = 0.0</code>	Raises <code>ZeroDivisionError</code>

Key insight: All positive numbers behave the same (just different magnitudes)!

reciprocal(x): Test Code

```
def test_reciprocal_positive():  
    """Test with positive number (representative: 5.0)"""  
    assert reciprocal(5.0) == 0.2  
  
def test_reciprocal_negative():  
    """Test with negative number (representative: -5.0)"""  
    assert reciprocal(-5.0) == -0.2  
  
def test_reciprocal_zero():  
    """Test with zero (representative: 0.0)"""  
    with pytest.raises(ZeroDivisionError):  
        reciprocal(0.0)
```

Result: 3 tests instead of ∞ tests! ✓

Common Mistake: Over-Testing

✗ Bad Approach (wasteful):

```
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
# All test the SAME equivalence class (positive numbers)!
```

✓ Good Approach (efficient):

```
def test_reciprocal_positive():
    assert reciprocal(5.0) == 0.2 # One representative is enough!
```

Example 2: Multiple Parameters

```
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
```

Interactive Question:

🤔 *How many equivalence classes do you think this has?*

- *3 parameters, each can be +/-/0, so... $3^3 = 27$ classes?*

The Surprise: Still Only 3 Classes!

Key insight: The **sum** is what matters, not individual parameters!

Positive sum	$x+y+z > 0$	<code>(1.0, 2.0, 3.0)</code>	Returns positive (<code>0.167</code>)
Negative sum	$x+y+z < 0$	<code>(-1.0, -2.0, -3.0)</code>	Returns negative (<code>-0.167</code>)
Zero sum	$x+y+z \approx 0$	<code>(1.0, -0.5, -0.5)</code>	Raises error

Why? Because `reciprocal_sum(1,2,3)` and `reciprocal_sum(2,1,3)` produce the **same behavior** (same sum).

Common Mistake: Testing All Sign Combinations

✗ **Bad Approach (wasteful - 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: These all test the **same 3 behaviors** (positive/negative/zero sum)!

Good Approach: Test Sum Categories

✓ Good Approach (efficient - 3 tests):

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

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

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

Key Principle: Output Categories Determine Input Classes

Wrong thinking:

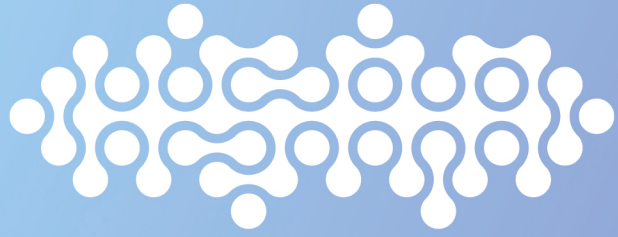
"I have 3 parameters → I need $3^3 = 27$ tests"

Correct thinking:

"What are the distinct **output behaviors**?" → 3 behaviors → 3 tests

General principle:

1. Look at the **code** (white-box testing)
2. Identify **what determines the output**
3. Group inputs by **similar behavior**
4. Choose **one representative** per group



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Choosing Granularity

3

Is the Partition Unique?

Question: Is there only ONE correct way to partition inputs?

Answer: No! **You choose** based on your goals.

Example for `reciprocal(x)` :

Option 1: Coarse (2 classes)

- Non-zero (`x ≠ 0`) → `x = 5.0`
- Zero (`x == 0`) → `x = 0.0`

Option 2: Fine (3 classes) ✨ (Most common)

- Positive (`x > 0`) → `x = 5.0`
- Negative (`x < 0`) → `x = -5.0`
- Zero (`x == 0`) → `x = 0.0`

Choosing Granularity (continued)

Option 3: Very Fine (5 classes)

- Large positive ($x > 1$) → $x = 10.0$
- Small positive ($0 < x \leq 1$) → $x = 0.1$
- Small negative ($-1 \leq x < 0$) → $x = -0.1$
- Large negative ($x < -1$) → $x = -10.0$
- Zero ($x == 0$) → $x = 0.0$

Trade-offs: Coarse vs. Fine Partitions

Coarse (2)	2	Basic	Low	Simple functions, low risk
Fine (3)	3	Good	Medium	Most functions (recommended)
Very Fine (5)	5	Thorough	High	Critical functions, high risk

Guidelines:

- **Coarse:** Quick smoke tests, low-risk functions
- **Fine:** Standard choice for most functions
- **Very Fine:** Security-critical, financial calculations, known bug-prone areas

Guidelines for Choosing Granularity

Consider these factors:

1. **Code paths:** How many distinct branches in the implementation?
2. **Mathematical properties:** Sign changes? Special values (0, 1, -1)?
3. **Domain significance:** Are there natural categories? (e.g., angles: 0°, 90°, 180°)
4. **Bug risk:** Has this area been buggy before?
5. **Cost of failure:** What happens if it breaks? (money lost? safety risk?)

Pragmatic approach: Start with 3-5 classes, add more if bugs are found.

White-Box vs. Black-Box Testing

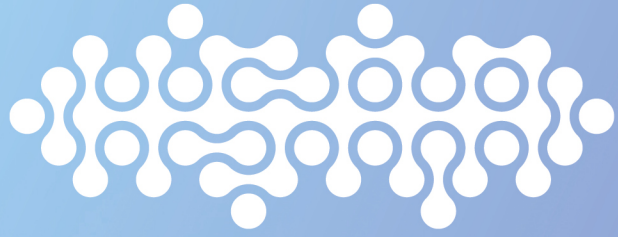
Black-Box Testing:

- You **don't see** the implementation
- Partition based on **specification** and **domain knowledge**
- Example: "reciprocal should work for positive and negative numbers"

White-Box Testing:

- You **can see** the implementation
- Partition based on **code structure** and **logic**
- Example: "I see an `if x == 0` check, so zero is a separate class"

Best practice: Combine both! 🎯



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Angles & Arrays

4

Example 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
```

🤔 *How many equivalence classes? 3 like before, or different?*

calculate_ray_slope: Four Equivalence Classes!

Downward rays	$-90^\circ < \text{angle} < 0^\circ$	$\text{angle} = -45^\circ$	Negative slope (<code>1.0</code>)
Horizontal rays	$\text{angle} = 0^\circ$	$\text{angle} = 0^\circ$	Zero slope (<code>0.0</code>)
Upward rays	$0^\circ < \text{angle} < 90^\circ$	$\text{angle} = 45^\circ$	Positive slope (<code>-1.0</code>)
Vertical rays	$\text{angle} = \pm 90^\circ$	$\text{angle} = 90^\circ$	None (no slope)

Why 4 instead of 3? Zero has **two meanings** here:

- Horizontal ray (`angle = 0°`) → slope is 0
- Vertical ray (`cos = 0`) → no slope (returns `None`)

calculate_ray_slope: Test Code

```
def test_downward_ray():  
    """Downward ray (-45°) → positive slope"""  
    assert calculate_ray_slope(-45.0) == pytest.approx(1.0, abs=0.01)  
  
def test_horizontal_ray():  
    """Horizontal ray (0°) → zero slope"""  
    assert calculate_ray_slope(0.0) == pytest.approx(0.0, abs=0.01)  
  
def test_upward_ray():  
    """Upward ray (45°) → negative slope"""  
    assert calculate_ray_slope(45.0) == pytest.approx(-1.0, abs=0.01)  
  
def test_vertical_ray():  
    """Vertical ray (90°) → None (no slope)"""  
    assert calculate_ray_slope(90.0) is None
```

Meta-Insight: Same Pattern, Different Granularity

<code>reciprocal(x)</code>	3	Sign of input: +, -, 0
<code>reciprocal_sum(x, y, z)</code>	3	Sign of sum : +, -, 0
<code>calculate_ray_slope(angle)</code>	4	Direction + special case (vertical)

Key takeaway:

- Same **pattern** (positive/negative/special)
- Different **granularity** based on context
- **You decide** how fine to partition based on requirements

Why Zero is Special Here

For `reciprocal(x)`:

- `x = 0` → **error** (can't divide by zero)
- Clear mathematical boundary

For `calculate_ray_slope(angle)`:

- `angle = 0°` → **valid result** (horizontal ray, slope = 0)
- `cos(angle) ≈ 0` (i.e., angle $\approx \pm 90^\circ$) → **special case** (vertical ray, no slope)

General principle:

Zero (or any special value) is a separate class when it triggers **different behavior** (error, edge case, special logic).

Arrays: A New Dimension of Complexity

So far: Simple numeric inputs (floats, ints)

Now: Arrays introduce **two dimensions** of equivalence:

1. **Structural equivalence classes:**

- How many elements?
- Empty vs. non-empty?

2. **Value-based equivalence classes:**

- What values do elements have?
- Positive? Negative? Mixed?

Result: Complexity multiplies! 🚀

Example 4: `array_sum(arr)`

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

Structural classes (by length):

- Empty: `[]`
- Single element: `[5.0]`
- Two elements: `[1.0, 2.0]`
- Many elements: `[1.0, 2.0, 3.0, 4.0]`

array_sum: Value-Based Classes

Value classes (by content):

- All positive: `[1.0, 2.0, 3.0]`
- All negative: `[-1.0, -2.0, -3.0]`
- Mixed signs: `[1.0, -2.0, 3.0]`
- Contains zeros: `[1.0, 0.0, 2.0]`
- All zeros: `[0.0, 0.0]`

Note: These can be combined with structural classes!

Combined Complexity: $4 \times 5 = 20$ Tests?

Structural (4) $\times$ Value (5) = 20 potential tests

Empty	N/A	N/A	N/A	N/A	✓
Single	✓	✓	N/A	✓	✓
Two	✓	✓	✓	✓	✓
Many	✓	✓	✓	✓	✓

Do we need all 20? Not necessarily! Strategic sampling is key.

Strategic Testing for array_sum

✓ Practical approach (8-10 tests):

```
# Structural coverage
test_empty_array()          # []
test_single_element()       # [5.0]
test_many_elements()        # [1, 2, 3, 4]

# Value coverage (with "many" structure)
test_all_positive()         # [1, 2, 3]
test_all_negative()         # [-1, -2, -3]
test_mixed_signs()          # [1, -2, 3]
test_contains_zeros()       # [1, 0, 2]

# Edge cases
test_all_zeros()            # [0, 0, 0]
```

Key: Cover each **dimension** with 2-3 representatives, plus edge cases.

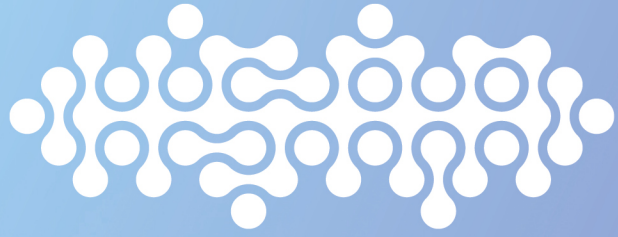
Common Mistake: Focusing on Wrong Dimension

✗ Bad Approach:

```
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]
# All test "many positive elements" - SAME equivalence class!
```

✓ Good Approach:

```
test_many_positive()     # [1, 2, 3, 4]      (many + positive)
test_many_negative()     # [-1, -2, -3, -4] (many + negative)
test_many_mixed()        # [1, -2, 3, -4]   (many + mixed)
# Different VALUE classes with SAME structure
```



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Discrete Functions

5

Good News: Discrete Functions Are Simpler! ✨

All examples so far had continuous outputs:

- `reciprocal(x)` → $\mathbb{R}$ (infinite real numbers)
- `calculate_ray_slope(angle)` → $\mathbb{R} \cup \{\text{None}\}$

We had to *choose* how to partition outputs (3 categories? 4? 5?)

But what about functions with discrete/finite outputs?

- Letter grades: `{"A", "B", "C", "D", "F"}`
- Boolean results: `{True, False}`
- Status codes: `{200, 404, 500}`

The simplification: Output categories are **pre-defined!** 🎯

Continuous vs. Discrete Functions

Aspect	Continuous	Discrete
Output space	Infinite ($\mathbb{R}$)	Finite set
Partitioning	You choose	Pre-determined
Number of classes	Your design choice	Fixed by return type
Main challenge	"How partition outputs?"	"Which inputs → which outputs?"

Key insight shift:

- **Continuous:** Design problem (how to partition?)
- **Discrete:** Analysis task (which inputs produce each output?)

Example 1: Letter Grades

```
def calculate_grade(score: int) -> str:
    """Calculate letter grade (A-F)."""
    if score < 0 or score > 100:
        raise ValueError("Score must be between 0 and 100")

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

🤔 *How many equivalence classes for `score` ?*

Grade Function: 6 Equivalence Classes

Class	Range	Representative	Output
Grade A	$90 \leq \text{score} \leq 100$	score = 95	"A"
Grade B	$80 \leq \text{score} < 90$	score = 85	"B"
Grade C	$70 \leq \text{score} < 80$	score = 75	"C"
Grade D	$60 \leq \text{score} < 70$	score = 65	"D"
Grade F	$0 \leq \text{score} < 60$	score = 30	"F"
Invalid	score < 0 or > 100	score = -10	ValueError

Why 6? 5 valid outputs + 1 error = 6 distinct behaviors

Grade Function: Test Code

```
def test_calculate_grade_A():
    """Equivalence class: Grade A (90-100)"""
    assert calculate_grade(95) == "A"
    assert calculate_grade(90) == "A"    # Boundary
    assert calculate_grade(100) == "A"   # Boundary

def test_calculate_grade_B():
    """Equivalence class: Grade B (80-89)"""
    assert calculate_grade(85) == "B"
    assert calculate_grade(80) == "B"    # Boundary
    assert calculate_grade(89) == "B"    # Boundary

# ... tests for C, D, F, Invalid ...
```

Pattern: One test per class + boundary values

Example 2: Password Validation

```
def validate_password(password: str) -> tuple[bool, str]:  
    """Validate password strength."""  
    if not isinstance(password, str):  
        return (False, "Password must be a string")  
    if len(password) < 8:  
        return (False, "Password too short")  
    if len(password) > 20:  
        return (False, "Password too long")  
    if not any(c.isupper() for c in password):  
        return (False, "Must contain uppercase letter")  
    # ... more checks ...  
    return (True, "Password is valid")
```

🤔 *How many equivalence classes?*

Password Validation: Basic Checks

Class	Representative	Output
Valid	"SecurePwd1!"	(True, "valid")
Not string	12345	(False, "must be string")
Too short	"Abc1!"	(False, "too short")
Too long	"VeryLong..."	(False, "too long")

4 classes so far... what about character requirements? 🤔

Password Validation: Content Requirements

Class	Representative	Output
No uppercase	"password1!"	(False, "uppercase")
No lowercase	"PASSWORD1!"	(False, "lowercase")
No digit	"Password!"	(False, "digit")
No special	"Password1"	(False, "special")

Total: 1 valid + 7 error paths = 8 equivalence classes!

Key insight: Each validation check = one equivalence class!

Password Validation: Order Matters!

Early return pattern:

```
if len(password) < 8:  
    return (False, "Password too short") # First check  
  
if not any(c.isupper() for c in password):  
    return (False, "Must contain uppercase") # Later check
```

What about "abc" (too short AND missing uppercase)?

- Belongs to "too short" class (first failing check wins!)

Testing strategy:

- One test per validation path
- Representatives trigger **exactly one** specific error

Example 3: Shipping Cost

```
def calculate_shipping_cost(weight: float, distance: float,  
                             express: bool = False) -> float:  
  
    # Validation omitted  
  
    # Weight categories  
    if weight <= 5:          base = 5.0  
    elif weight <= 20:       base = 10.0  
    else:                    base = 20.0  
  
    # Distance multiplier  
    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)
```

Shipping Cost: Three Dimensions

Weight (3 classes)	Distance (3 classes)	Express (2 classes)
Light: ≤ 5 kg $\rightarrow$ €5	Local: ≤ 100 km $\rightarrow$ 1.0×	Standard $\rightarrow$ 1.0×
Medium: 5-20 kg $\rightarrow$ €10	Regional: 100-500 km $\rightarrow$ 1.5×	Express $\rightarrow$ 1.5×
Heavy: > 20 kg $\rightarrow$ €20	Long: > 500 km $\rightarrow$ 2.0×	

Shipping Cost: Combinatorial Challenge

Total combinations:

$$3 \times 3 \times 2 = \mathbf{18} \text{ valid classes}$$

Plus 2 error classes (invalid weight, invalid distance)

Total: 20 equivalence classes! 🤖

Do we need to test all 18? Not necessarily!

Strategic Testing: Dimension Coverage

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

# Distance dimension (fix 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 (fix weight=2.5, distance=50)
test_standard_shipping() # €5 × 1.0
test_express_shipping()  # €5 × 1.5

# Boundary tests + error tests
```

Result: ~10 strategic tests instead of 18!

Key Insights: Discrete vs. Continuous

When discrete is EASIER:

- ✓ Output categories are obvious (just list return values!)
- ✓ No granularity debates (function signature defines classes)
- ✓ Complete coverage achievable (6 grades? Write 6 tests!)

When discrete is HARDER:

- ✗ Many possible outputs (8 validation errors = 8 tests minimum)
- ✗ Combinatorial explosion ($3 \times 3 \times 2 = 18$ classes)
- ✗ Hierarchical validation (order matters - early returns)
- ✗ Boundary values still critical (89 vs 90 can hide bugs)

Takeaway: Discrete simplifies output partitioning, not input analysis!

Practical Guidelines

Use discrete analysis when:

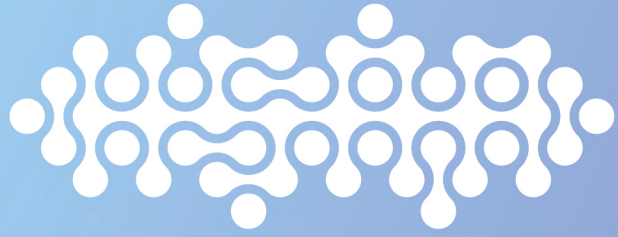
-  Finite return values | Explicit validation
-  Threshold-based logic | Continuous → Discrete mapping

Use continuous analysis when:

-  Math computations on $\mathbb{R}$ | Truly continuous output
-  Granularity must be chosen

Use hybrid analysis when:

-  Continuous inputs → discrete outcomes
-  Geometry/physics | Calculations + conditional logic



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Real-World Complexity: `find_intersection()`

6

Connection: find_intersection() Is Hybrid!

Looks continuous:

- Inputs: floats (angle, position, road points)
- Output: `tuple[float, float] | None` (coordinates)

But outcomes are discrete:

1. **No intersection** → `None`
2. **Ray intersects ascending segment**
3. **Ray intersects descending segment**
4. **Ray intersects horizontal segment**
5. **Ray intersects at vertex** (edge case)

Testing Strategy for Hybrid Functions

Key insight: Even though inputs are continuous (floats), the **outcomes are discrete!**

Testing strategy:

Identify discrete outcome categories, then pick representative inputs for each!

This is exactly what we did with `calculate_shipping_cost` ! 🎯

Pattern:

- Continuous inputs → Discrete outcomes → Test one representative per outcome

The Real Challenge: find_intersection()

From your Road Profile Viewer project:

```
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 ...
```

Think about it: How would you test this? 🤔

Deep Dive Available

★ **Important:** This example is so complex that we've created a **separate detailed walkthrough** with 22 visual illustrations!

See the "**Deep Dive - Testing find_intersection()**" slide deck for the complete analysis.

📖 **Full details:**

[Lecture notes on find_intersection\(\)](#)

How Many Tests Needed?

Interactive Poll:

🤔 How many tests do you think we need for `find_intersection()` ?

A) 5-10 tests

B) 20-30 tests

C) 50-100 tests

D) More than 100 tests

Discuss with your neighbor! ⌚ (30 seconds)

Complexity Dimension 1: Array Structure

x_road and y_road arrays:

1. **Empty** (`len = 0`) → error handling
2. **Single point** (`len = 1`) → degenerate case
3. **Two points** (`len = 2`) → one segment, minimal valid
4. **Three points** (`len = 3`) → two segments
5. **Many points** (`len > 3`) → multiple segments, typical case

Complexity Dimension 2: Angle Classes

angle_degrees:

1. **Downward ray** ($-90^\circ < \text{angle} < 0^\circ$)
2. **Horizontal ray** ($\text{angle} = 0^\circ$)
3. **Upward ray** ($0^\circ < \text{angle} < 90^\circ$)
4. **Vertical ray** ($\text{angle} = \pm 90^\circ$) $\rightarrow$ no slope, special handling
5. **Out of range** ($|\text{angle}| > 90^\circ$) $\rightarrow$ invalid input

Complexity Dimension 3: Camera Position

camera_x and camera_y:

1. **Camera far left** (before first road point)
2. **Camera between first and second point**
3. **Camera in middle of road**
4. **Camera between last two points**
5. **Camera far right** (after last road point)
6. **Camera above/below road** (y position)

Why these matter:

- Position affects which segments are intersectable
- Edge positions test boundary logic

Complexity Dimension 4: Outcome-Based Classes

Based on what the function returns:

1. **Intersection found** → `(x, y)`
2. **No intersection** (ray misses road) → `None`
3. **Multiple possible intersections** (which to choose?)
4. **Ray parallel to segment** (edge case) → `None`

Why these matter:

- Each outcome exercises different code paths
- Edge cases reveal bugs

Complexity Dimension 5: Road Shape

Geometric properties:

1. **Flat road** (all y values same)
2. **Monotonic ascending/descending**
3. **Has peaks and valleys**

Why these matter:

- Flat road simplifies geometry
- Peaks/valleys increase intersection complexity
- Affects which ray angles find intersections

Five Dimensions of Complexity

For `find_intersection()`, we identified:

1. **Array structure:** 5 classes
2. **Angle:** 5 classes
3. **Camera position:** 6 classes
4. **Outcome:** 4 classes
5. **Road shape:** 3 classes (simplified)

Let's do the math... 

The Math: Combinatorial Explosion! 🤯

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

(We simplified road shape to avoid going over 1000!)

Reality check: Can you write and maintain 600 tests? 😊

Spoiler: No! That's where human insight comes in... 🧠

AI Limitations: What LLMs Miss

You might think: "Just ask ChatGPT/Claude to generate all 600 tests!"

Problems with AI-generated tests:

1. **No geometric intuition:** Can't visualize ray-road intersections
2. **No domain knowledge:** Doesn't understand camera-road relationships
3. **Pattern repetition:** Generates similar tests with different numbers
4. **Misses edge cases:** No awareness of parallel rays, degenerate geometry
5. **No prioritization:** Treats all combinations equally (doesn't know what's important)

Bottom line: LLMs can **help**, but can't **replace** human insight! 🧠

What Humans See (That AI Doesn't)

Human insight:

- "Downward ray from above-road camera will **always** miss road" → skip those combinations
- "Camera far right + upward ray → intersection impossible" → skip
- "Two-point road + vertical ray → only works if ray passes through points" → rare edge case
- "Flat road makes angle less important" → reduce angle testing

Result: Humans can prune 600 → **20-30 strategic tests** based on domain knowledge.

This is your value as an engineer! 🎯

Practical Reduction Strategy: 600 → 30 Tests

Strategy: Cover each dimension with 2-3 representatives

1. **Array structure** (5 classes) → test 3: empty, two-point, many-point
2. **Angle** (5 classes) → test 3: downward, horizontal, upward
3. **Camera position** (6 classes) → test 2: middle, edge
4. **Outcome** (4 classes) → ensure all 4 appear in test suite
5. **Road shape** (3 classes) → test 2: flat, varied

The Math: 600 → 30 Tests

Math: $3 + 3 + 2 + 4 + 2 = 14$ base tests

Plus edge cases: Vertical ray, parallel segment, out-of-range → **~10 more**

Total: ~25-30 tests 

 **Want to see the actual tests?** Check out the "**Deep Dive - Testing find_intersection()**" slide deck for all 22 tests with visual diagrams!

Example Test Class Structure (High Level)

```
class TestFindIntersection:
    # Structural dimension
    def test_empty_arrays(self): ...
    def test_two_point_road(self): ...
    def test_many_point_road(self): ...

    # Angle dimension
    def test_downward_ray(self): ...
    def test_horizontal_ray(self): ...
    def test_upward_ray(self): ...
    def test_vertical_ray(self): ... # Edge case

    # Outcome dimension
    def test_intersection_found(self): ...
    def test_no_intersection(self): ...
    def test_ray_parallel_to_segment(self): ...
```

Example Test Class Structure (Combinations)

```
class TestFindIntersection:
    # ... (previous tests)

    # Smart combinations (domain-driven)
    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): ...

    # Edge cases
    def test_ray_through_road_endpoint(self): ...
    def test_angle_out_of_range(self): ...
```

Collaborative Testing: Human + LLM

Human: Identify classes, choose granularity, prioritize risks, review

LLM: Generate boilerplate, create test data, write assertions, suggest edges

Workflow: You design → LLM implements → You review! 🤝

How to Use LLMs Effectively for Testing

✓ Good prompts:

- "Generate a test for `find_intersection` with a two-point flat road and horizontal ray"
- "Write a test that expects `None` when the ray misses the road"

✗ Bad prompts:

- "Generate all possible tests for `find_intersection`" → produces low-quality noise
- "Test this function" → vague, no context

Key: Be **specific** about equivalence classes and expected behavior!

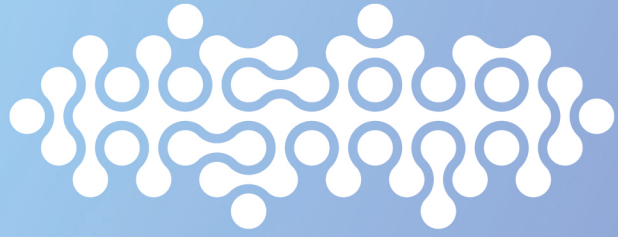
Discussion: Real-World Trade-offs

Scenario: You're testing `find_intersection()` for a self-driving car.

Questions for discussion:

1. Would you test all 600 combinations? Why or why not?
2. How would your answer change if:
 - It's a video game (not safety-critical)?
 - It's medical imaging software (lives at stake)?
 - You have 2 days vs. 2 weeks for testing?
3. Which combinations are **highest risk**?

Think-pair-share: Discuss with your neighbor! 🕒 (2 minutes)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Key Takeaways & Wrap-Up

7

Key Takeaways

1. Equivalence classes reduce infinite inputs to finite tests

- Group inputs by similar behavior, test one per class

2. Output categories determine input classes

- Look at what the function **does**, not what it **takes**
- `reciprocal_sum(x, y, z)`: 3 output behaviors → 3 classes

3. Granularity is a design choice

- Simple/low-risk: 2-3 classes | Complex/high-risk: 5-10 classes
- Consider: code paths, domain, bug risk, failure cost

Key Takeaways (continued)

4. White-box testing reveals hidden equivalence classes

- Looking at code shows what **actually** determines behavior
- Example: `reciprocal_sum` code reveals sum is what matters

5. Real-world functions have combinatorial complexity

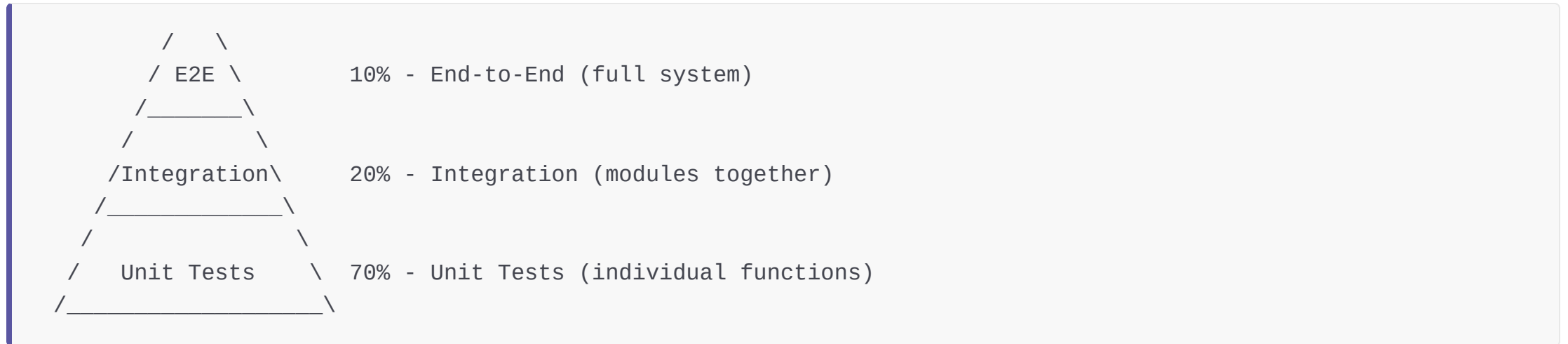
- `find_intersection`: 5 dimensions × many classes = 600 tests
- Reduce via domain knowledge: 600 → 25-30 strategic tests

6. Human insight is irreplaceable

- LLMs can't visualize geometry, understand domain, or prioritize
- You design equivalence classes, LLM helps implement tests

The Big Picture

Testing Pyramid (Reminder):



Today's focus: How to choose **which unit tests** to write efficiently!

Next topic: Boundary value analysis (finding bugs at edges)

Preview: Boundary Value Analysis (Coming Next)

Equivalence class testing is powerful, but it's not enough!

Problem: Bugs often hide at **boundaries** between classes.

Example for `reciprocal(x)` :

- We tested: `x = 5.0` , `x = -5.0` , `x = 0.0` ✓
- But what about: `x = 0.00000001` ? (very close to zero) 🤔
- Or: `x = 1e308` ? (very large, near float max)

Boundary value analysis says: Test at the **edges** of equivalence classes!

Next lecture: How to find and test boundaries systematically.

Hands-On Practice

Exercise: Choose equivalence classes for this function:

```
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"
```

Take 2 minutes: Work individually or with a partner 

Practice Questions

For `classify_temperature(temp_celsius)` :

1. How many equivalence classes?
2. What are they?
3. What representative values would you choose?

Bonus: What boundary values should you test?

Full lecture: [Chapter 03 \(Testing Basics\) - Testing Fundamentals](#)

Summary: The Pattern

For any function, ask:

1. **What are the distinct output behaviors?** (outcomes, return types, errors)
2. **What inputs produce each behavior?** (group by similarity)
3. **How fine should I partition?** (based on risk, complexity, resources)
4. **What's one representative per class?** (typical value that exercises the behavior)

Result: Finite, manageable test suite that covers infinite input space! 🎯

Remember: You're not testing all inputs — you're testing all **behaviors**!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

THANK YOU

Next: Boundary Value Analysis