



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Chapter 02

REFACTORING — FROM MONOLITH TO MODULES

Where We Are Now

Congratulations! You've achieved:

- ✅ **Chapter 02 (Code Quality in Practice):** PEP8 compliance with Ruff
- ✅ **Chapter 02 (Feature Branch Development):** Feature branches & Pull Requests
- ✅ **Chapter 02 (Automation and CI/CD):** Automated CI/CD checks

Your CI is green! 🎉

But... your entire app lives in a single 390-line `main.py` file.

The Problem: Real Pain Points

Scenario 1: The Bug Hunt

```
You: "There's a bug in intersection calculation."  
You: *scrolls through 390 lines*  
You: *passes generate_road_profile(), create_dash_app()...*  
You: "Found it at line 267! Wait, what does this depend on?"
```

Scenario 2: The Merge Conflict

```
Student A: Edits lines 50-120 (road generation)  
Student B: Edits lines 200-350 (UI styling)  
Result: MERGE CONFLICT (despite unrelated changes!)
```

The Problem: Testing Nightmare

Scenario 3: Testing (Preview of Chapter 03 (Testing Fundamentals))

```
# You want to test find_intersection()
from main import find_intersection

# What actually happens:
# - Imports entire Dash framework
# - Starts web server
# - Takes 5 seconds per test
# - 20 functions to test = 100 seconds!
```

Result: You give up on testing. 😞

Today's Learning Objectives

By the end of this lecture, you will:

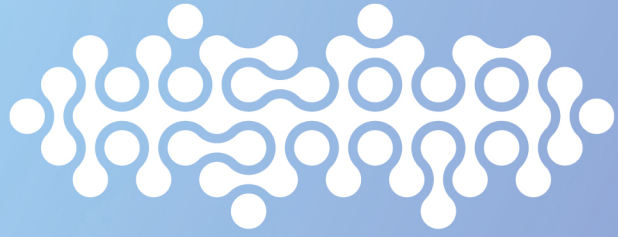
1.  **Understand** why monolithic code becomes unmaintainable
2.  **Identify** natural boundaries in your code
3.  **Refactor** a monolith into focused modules
4.  **Organize** imports to avoid circular dependencies
5.  **Experience** the benefits of modular code
6.  **Prepare** your codebase for testing (Chapter 03 (Testing Fundamentals))

What You WON'T Learn Today

Not covered (yet):

- ✗ Design patterns by name (emerge naturally from practice)
- ✗ Advanced architecture patterns (come later)
- ✗ Testing (that's Chapter 03 (Testing Fundamentals)!)

Today's Focus: Practical refactoring with real code.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Workflow & Plan

1

Applying the Feature Development Workflow

Remember the workflow from Chapter 02 (Feature Development)?

1. Create a feature branch
2. Implement your changes
3. Push to GitHub
4. Open a Pull Request
5. Automated CI checks run
6. Code review
7. Merge when approved
8. Delete the branch

Today's refactoring IS a feature! We'll use this workflow.

Step 1: Create Your Feature Branch

```
# Make sure you're on main and up to date
$ git checkout main
$ git pull origin main

# Create a new feature branch
$ git checkout -b feature/refactor-to-modules
Switched to a new branch 'feature/refactor-to-modules'

# Verify you're on the new branch
$ git branch
  main
* feature/refactor-to-modules
```

Why? Main stays stable while we experiment safely.

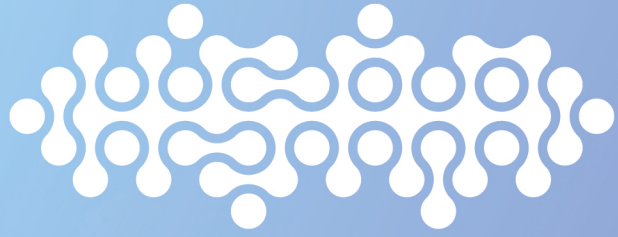
The Plan: Iterative Refactoring

Small, safe steps with commits after each:

```
Step 1: Extract geometry.py → Commit → Test  
Step 2: Extract road.py → Commit → Test  
Step 3: Extract visualization.py → Commit → Test  
Step 4: Simplify main.py → Commit → Test  
Final: Push branch → Open PR → CI validates → Merge
```

Key Principle: After each step, the application still works perfectly.

Why small steps? Easy to identify what broke, easy to undo.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Understanding the Monolith

2

What is a Monolith?

Code where everything lives in one place, tightly coupled together.

Your current `main.py` (390 lines):

- Road profile generation (lines 20-60)
- Geometry calculations (lines 60-180)
- Dash application UI (lines 190-390)
- Application entry point

Everything is mixed together: Math, UI, data generation, app logic.

Why Monoliths Become Problems

1. Hard to Navigate

- Finding specific functionality requires scrolling through unrelated code
- Line 111? Was that geometry or road generation?

2. Hard to Test

- Must import everything to test one function
- Dash starts, slow imports, side effects

3. Hard to Collaborate

- Two developers editing same file = merge conflicts
- Even when changes are completely independent!

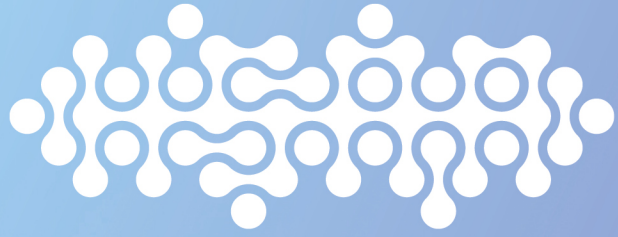
Why Monoliths Become Problems (2)

4. High Coupling

```
def find_intersection(...):  
    # Accidentally uses a global variable from UI section  
    camera_x = CAMERA_POSITION # Where is this defined??  
    # Now geometry depends on UI - hard to test!
```

5. Violates "One File, One Purpose"

- What does `main.py` do?
- Current answer: "Everything!"
- Better answer: "Starts the application. That's it."



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Natural Boundaries

3

The "3-Word Description" Test

If you can describe a module in 3 words or less, it's focused enough.

✓ Good Examples:

- `geometry.py` → "Ray intersection math"
- `road.py` → "Road profile generation"
- `visualization.py` → "Dash UI components"
- `main.py` → "Application entry point"

The "3-Word Test" (Continued)

✗ Bad Examples:

- `utilities.py` → "Miscellaneous useful stuff" (too vague!)
- `helpers.py` → "Helper functions" (which kind??)
- `functions.py` → "Various functions" (meaningless!)

Key Principle: Specific, focused naming reveals clear purpose.

Analyzing Your Current Code

Category 1: Pure Math Functions

- `calculate_ray_line()` - Only depends on numpy
- `find_intersection()` - Only depends on numpy
- **Characteristic:** Same inputs = same outputs (pure)
- **Destination:** `geometry.py`

Category 2: Data Generation

- `generate_road_profile()` - Only depends on numpy
- **Characteristic:** Generates data, doesn't display it
- **Destination:** `road.py`

Analyzing Your Current Code (2)

Category 3: UI and Application Logic

- `create_dash_app()` - Depends on Dash, geometry, road
- `update_graph()` callback - Orchestrates everything
- **Characteristic:** Creates UI, handles user input
- **Destination:** `visualization.py`

Category 4: Entry Point

- `main()` - Just starts the application
- **Characteristic:** Minimal logic
- **Destination:** `main.py` (keep simple!)

The Dependency Hierarchy

Natural dependency flow:

```
main.py
  ↓ depends on
visualization.py
  ↓ depends on
geometry.py, road.py
  ↓ depends on
numpy (external library)
```

Key Insight: Dependencies flow **ONE DIRECTION ONLY**.

Why? Prevents circular dependencies!

✓ `visualization.py` can import from `geometry.py`

✗ `geometry.py` should NOT import from `visualization.py`

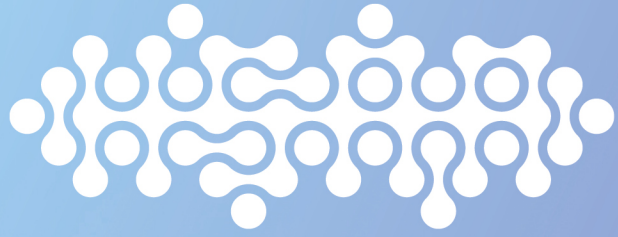
Target Structure

After refactoring:

```
road-profile-viewer/  
├─ main.py           # Entry point only (~15 lines)  
├─ geometry.py       # Pure math functions (~80 lines)  
├─ road.py           # Road generation (~60 lines)  
├─ visualization.py  # Dash app and UI (~200 lines)  
└─ (other files remain the same)
```

From 1 file (390 lines) → 4 focused modules (465 lines total)

Extra lines are documentation - that's good! 



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Hands-On Refactoring

4

Step 1: Create the Module Files

```
# In your road-profile-viewer directory
$ touch geometry.py
$ touch road.py
$ touch visualization.py

# Verify the structure
$ ls *.py
geometry.py  main.py  road.py  visualization.py
```

Now we have our module files ready!

Step 2: Extract Geometry

Create `geometry.py` with:

- `calculate_ray_line()` function
- `find_intersection()` function
- Comprehensive docstrings
- Type hints for IDE support

Update `main.py`:

```
from geometry import calculate_ray_line, find_intersection
```

Delete the geometry function definitions from `main.py`.

Step 2: Test & Commit

Test that it works:

```
$ uv run road-profile-viewer  
# Application should start normally  
# Verify ray intersection still works
```

✓ **Checkpoint:** Application works!

Commit your progress:

```
$ git add geometry.py main.py  
$ git commit -m "Extract geometry functions to geometry.py"
```

- Move `calculate_ray_line()` and `find_intersection()`
- Add comprehensive docstrings and type hints
- Update `main.py` imports
- Application still works identically"

Step 3: Extract Road

Create `road.py` with:

- `generate_road_profile()` function
- Docstrings explaining the clothoid approximation

Update `main.py` :

```
from road import generate_road_profile
```

Delete the road generation function from `main.py` .

Step 3: Test & Commit

Test again:

```
$ uv run road-profile-viewer  
# Verify road generation still works
```

✓ **Checkpoint:** Still working!

Commit this step:

```
$ git add road.py main.py  
$ git commit -m "Extract road generation to road.py"
```

- Move `generate_road_profile()`
- Add docstrings
- Update imports
- Application still works identically"

Step 4: Extract Visualization

Create `visualization.py` with:

- `create_dash_app()` function
- All UI layout code
- All callbacks
- Imports from `geometry` and `road` modules

Simplify `main.py`:

```
from visualization import create_dash_app

def main():
    app = create_dash_app()
    print("Starting Road Profile Viewer...")
    app.run(debug=True)
```

Step 4: Test & Commit

Test the complete refactoring:

```
$ uv run road-profile-viewer  
# Verify EVERYTHING still works:  
# - Road renders correctly  
# - Ray angle adjusts  
# - Intersection point updates  
# - Distance displays
```

✓ **Checkpoint:** Everything works! 🎉

Step 4: Final Commit

```
$ git add visualization.py main.py  
$ git commit -m "Extract UI layer to visualization.py"
```

- Move create_dash_app() and all UI code
- Simplify main.py to just entry point (~20 lines)
- visualization.py imports from geometry and road
- Application works identically to before

Completes modular refactoring:
main → visualization → geometry/road"

Now we have a clean commit history showing the refactoring progression!

Verifying the Refactoring

Check your module structure:

```
$ ls *.py
geometry.py  main.py  road.py  visualization.py
```

Check line counts:

```
$ wc -l *.py
 175 geometry.py      # Pure geometry calculations
   67 road.py         # Road generation
 203 visualization.py # All UI code
   20 main.py         # Just the entry point
  ---
 465 total
```

main.py went from 390 lines to 20 lines! 📉

Verify the Dependency Flow

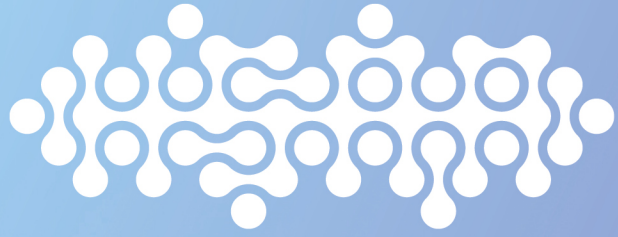
```
# main.py imports:
from visualization import create_dash_app

# visualization.py imports:
from geometry import calculate_ray_line, find_intersection
from road import generate_road_profile

# geometry.py imports:
import numpy as np # Only external dependency

# road.py imports:
import numpy as np # Only external dependency
```

✓ **One-directional dependencies — no cycles!**



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Benefits Gained

5

Easy Navigation

Before:

```
# "Where's the intersection calculation?"  
# *Scrolls through 390 lines*  
# *Gets lost in unrelated code*
```

After:

```
# "Where's the intersection calculation?"  
# Opens geometry.py → find_intersection() is right there!
```

Win: Find code in seconds instead of minutes.

Focused Changes

Before:

```
# Changing road generation algorithm  
# File: main.py (390 lines)  
# Risk: Might accidentally break UI code
```

After:

```
# Changing road generation algorithm  
# File: road.py (67 lines)  
# Safety: Can't accidentally touch UI code!
```

Win: Changes are safer and more focused.

Better Collaboration

Before:

Student A: Editing main.py lines 50-120 (road generation)
Student B: Editing main.py lines 200-350 (UI styling)
Result: MERGE CONFLICT! 🤬
Time wasted: 30 minutes resolving conflicts

After:

Student A: Editing road.py
Student B: Editing visualization.py
Result: No conflicts! 😊
Time wasted: 0 minutes

Win: Parallel development without conflicts!

Testability

Before:

```
# To test find_intersection()
from main import find_intersection
# Imports EVERYTHING (Dash, UI, etc.)
# Slow, heavy, side effects
```

After:

```
# To test find_intersection()
from geometry import find_intersection
# Only imports geometry module
# Fast, lightweight, no side effects
```

Win: Fast, focused tests (Chapter 03 (Testing Fundamentals)!)

Reusability

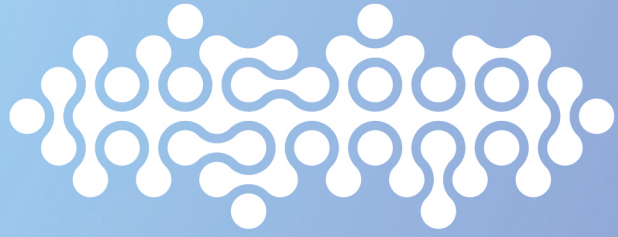
Before:

```
# Want to use find_intersection() in another project?  
# Copy-paste from main.py  
# → Brings along unnecessary Dash code 😞
```

After:

```
# Want to use find_intersection() in another project?  
# Just copy geometry.py  
# → Clean, standalone module! 😊
```

Win: Pure functions are reusable everywhere.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Common Pitfalls

6

Pitfall #1: Circular Imports

The Error:

```
ImportError: cannot import name 'generate_road_profile'  
from partially initialized module 'road'  
(most likely due to a circular import)
```

The Cause:

```
# visualization.py  
from road import generate_road_profile  
  
# road.py (BAD!)  
from visualization import create_dash_app  
# Why would road.py need UI code??
```

Pitfall #1: Solution

Rule: Lower-level modules (geometry, road) should **NEVER** import from higher-level modules (visualization, main).

Dependencies flow one direction:

```
main → visualization → geometry/road
```

Not:

```
geometry ← visualization ← main (✗ WRONG!)
```

Pitfall #2: Unclear Boundaries

The Confusion:

```
# Where do camera_x and camera_y belong?  
# - geometry.py? (used in calculations)  
# - visualization.py? (it's a UI parameter)  
# - A new config.py file?
```

The Solution:

- **If hardcoded:** Keep as parameter default in function signature
- **If user-configurable:** Keep in visualization.py (UI decides)
- **Don't create "config.py" prematurely!** You'll know when needed.

Pitfall #3: Over-Refactoring

The Temptation:

Should I create:

- geometry_helpers.py?
- geometry_utils.py?
- ray_calculations.py?
- intersection_math.py?
- angle_converter.py?

The Solution: YAGNI Principle

- **"You Ain't Gonna Need It"**
- Start with 3-4 focused modules
- Only split when a module becomes too large (>300 lines)
- **Avoid "utility/helper" modules** — they become dumping grounds!

Pitfall #4: Forgetting Imports

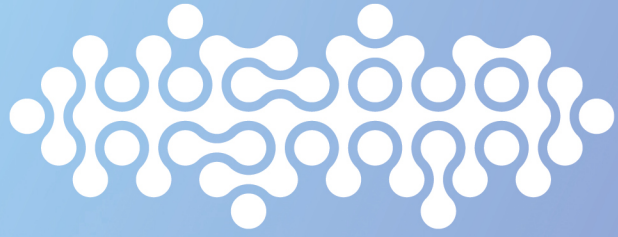
The Error:

```
# In main.py
def main():
    x, y = generate_road_profile()
# NameError: name 'generate_road_profile' is not defined
```

The Fix:

```
# Add the import at the top
from road import generate_road_profile
```

Pro Tip: Use IDE auto-import (Ctrl+. in VS Code).



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

CI & PR Workflow

7

Running CI Checks

Your refactoring should pass all CI checks:

```
# Run Ruff (style check)
```

```
$ uv run ruff check .
```

```
All checks passed! ✓
```

```
# Run Ruff format check
```

```
$ uv run ruff format --check .
```

```
All files formatted correctly! ✓
```

```
# Run Pyright (type check)
```

```
$ uv run pyright
```

```
0 errors, 0 warnings ✓
```

If you get errors: Fix them before pushing!

Push Your Feature Branch

Review your commits:

```
$ git log --oneline  
7h8i9j0 Extract UI layer to visualization.py  
d4e5f6g Extract road generation to road.py  
a1b2c3d Extract geometry functions to geometry.py
```

Clean, logical progression! ✨

Push to GitHub:

```
$ git push -u origin feature/refactor-to-modules
```

Create Pull Request

```
$ gh pr create \  
  --title "Refactor: Split monolithic main.py into focused modules" \  
  --body "Refactors monolithic main.py (390 lines) into 4 modules:"
```

- geometry.py: Pure math functions
- road.py: Road generation logic
- visualization.py: Dash UI layer
- main.py: Entry point only (20 lines)

Benefits: Easier navigation, testing, reduced merge conflicts

- ✓ All Ruff/Pyright checks pass
- ✓ Application works identically"

Or use the GitHub web interface!

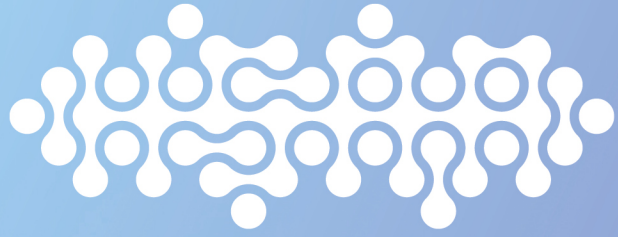
Wait for CI Checks

GitHub Actions will automatically run:

- ✓ Ruff check (style)
- ✓ Ruff format check
- ✓ Pyright (types)

All should pass! (You've been running them locally.)

Once green: Ready to merge! 🚀



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

AI as a Tool

8

Leveraging AI Coding Assistants

Important Disclaimer:

Before using AI for refactoring, you MUST understand:

-  What makes good module boundaries
-  How to avoid circular dependencies
-  The "3-word description" test
-  One-directional dependency flow

Why? Without these principles, you cannot:

- Write effective prompts
- Evaluate AI's suggestions
- Catch subtle mistakes

AI is a tool, not a replacement for your knowledge!

What AI Can and Cannot Do

AI can help with:

-  Mechanical code movement (copy/paste between files)
-  Updating import statements
-  Writing boilerplate (docstrings, type hints)
-  Maintaining consistency

AI cannot:

-  Decide WHERE to split your code (requires judgment)
-  Understand your project's architecture needs
-  Evaluate trade-offs
-  Ensure alignment with team conventions

Example Prompt

You've planned the refactoring (manually!), now AI helps execute.

You will refactor monolithic main.py (390 lines) into:

- geometry.py - Pure math functions
- road.py - Road profile generation
- visualization.py - Dash UI layer
- main.py - Entry point (~20 lines)

Create feature branch feature/refactor-to-modules

Make 4 incremental commits (one per step).

Step 1: Extract Geometry Module

- Copy `calculate_ray_line()` and `find_intersection()`
- Add imports: `import numpy as np`
- Keep all docstrings and type hints
- Commit: "refactor: extract geometry to separate module"

[Continue with Steps 2, 3, 4...]

Why This Prompt Works

Clear Structure:

-  Explicit file names and locations
-  Step-by-step breakdown
-  Specific requirements
-  Commit message format

Includes Context:

-  Explains WHY (absolute imports, etc.)
-  What to preserve (docstrings)
-  Shows the desired end state

Enables Verification:

-  Each step independently testable

Clear success criteria

Responsibilities After Code Generation

Never blindly accept AI-generated code!

Always:

1. ✓ Read every line — Understand what changed
2. ✓ Test after each step — `uv run road-profile-viewer`
3. ✓ Verify imports — Check for circular dependencies
4. ✓ Run CI checks — Make sure Ruff and Pyright pass
5. ✓ Review commits — Check git diff
6. ✓ Validate structure — Does it match your plan?

Red Flags to Watch For

Common AI Mistakes:

- ✗ Circular imports (geometry imports from visualization)
- ✗ Missing or wrong import paths
- ✗ Lost functionality (deleted instead of moved)
- ✗ Changed logic (AI "improved" something unnecessarily)
- ✗ Inconsistent naming or style

You are the quality gate!

Human-AI Collaboration Model

Human: Strategic Thinking

- |— Identify monolith problems
- |— Decide module boundaries
- |— Plan dependency hierarchy
- |— Define success criteria

↓

AI: Tactical Execution

- |— Move code between files
- |— Write boilerplate code
- |— Format commit messages

↓

Human: Verification & Quality

- |— Test functionality
- |— Review architecture
- |— Look for subtle bugs
- |— Validate against CI

You are the architect. AI is the construction crew.

Learning Exercise Recommendation

For students:

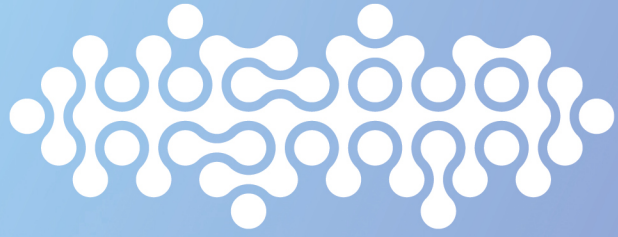
1. First time: Do it manually (today's exercise)

- Deeply understand the challenges
- Build intuition for problems
- Learn to spot circular dependencies

2. Second time: Use AI assistance (future projects)

- Now you can evaluate AI's suggestions
- You'll catch its mistakes
- You'll write better prompts

Why this order? Learn principles first, then leverage automation.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Summary

9

Before vs. After

Before refactoring:

```
main.py (390 lines)
└─ Everything mixed:
    Math, UI, data, startup
```

- ✗ Hard to find code
- ✗ Hard to test
- ✗ Merge conflicts
- ✗ Tight coupling

After refactoring:

```
└─ geometry.py (175 lines)
└─ road.py (67 lines)
└─ visualization.py (203 lines)
└─ main.py (20 lines)
```

- ✓ Clear module boundaries
- ✓ Testable
- ✓ Editable in parallel
- ✓ Loose coupling

Benefits

- ✓ **Easy navigation** — Clear module names
- ✓ **Testable** — Import geometry without Dash
- ✓ **Collaborative** — Edit different files simultaneously
- ✓ **Reusable** — Pure functions in standalone modules
- ✓ **Maintainable** — Single-responsibility modules

Key Principles Learned

1. **Separation of Concerns**

- Each module does one thing well

2. **Dependency Direction**

- Lower-level modules don't import from higher-level

3. **The "3-Word Description" Test**

- Can you describe the module concisely?

4. **YAGNI ("You Ain't Gonna Need It")**

- Don't create abstractions until they're needed

Key Takeaways

1. **Monoliths grow organically** — fine initially, problems later
2. **Refactoring $\neq$ rewriting** — same functionality, better structure
3. **Modules need clear boundaries** — pure functions vs. UI
4. **Dependencies flow one direction** — prevents circular imports
5. **Testability emerges from modularity** — can't test what you can't import
6. **The "3-Word Test"** — describe module purpose concisely
7. **YAGNI** — start simple, refactor when needed
8. **CI should pass** — refactoring doesn't change functionality

Further Reading

On Refactoring:

- Martin Fowler, *Refactoring: Improving the Design of Existing Code*
- Quote: "Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

On Module Design:

- Robert C. Martin, *Clean Code*
- SOLID Principles (you're applying Single Responsibility!)

On Python Modules:

- Real Python: Python Modules and Packages
- PEP 8: Package and Module Names

Practice Time: Hands-On Exercise!

Now it's your turn to apply what you've learned!

What you'll do:

-  Refactor a real monolithic codebase (390 lines → 4 modules)
-  Follow the exact steps from today's lecture
-  Get automated feedback on your refactoring
-  See your score improve as you complete each step

Experience the full workflow: Feature branch → Commits → Tests → CI/CD



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

THANK YOU