The AI-Native Refactorer

Topic 1: Long Methods

Hands-On: Decomposing a 90-Line Function with AI in 15 Minutes

Overview

This hands-on proves the three core claims from the Topic 1 lecture in real time, using an AI coding assistant (such as Claude, Copilot, or Cursor) and about ninety lines of Python:

- A long method is a structural problem you can fix without changing behavior — refactoring is about names and boundaries, not logic.
- Characterization tests are non-negotiable — they are what let you change structure aggressively while staying safe.
- The right prompts give clean, scoped diffs — vague prompts give you rewrites you don't want.

You will run three small Python programs in sequence and drive an AI assistant through the five-step refactoring workflow. The student assignment at the end of this hands-on extends the work by tackling a second long method on the student's own.

Before You Start

- Have Python 3.10+ installed. Verify with python3 --version.
- Install pytest: pip install pytest.
- Open the file topic1_before.py from https://github.com/TheAI-Native/AI-NativeRefactorer
- Have an AI coding assistant ready in your editor (Claude Code, Cursor, Copilot Chat, or similar).
- Have a terminal open in the same directory, ready to run the file step by step.
- Have the lecture slides visible on a second screen if possible — the five-step workflow diagram from Slide 10 makes the structure of this exercise land harder.

REMEMBER

Every refactoring topic in this course follows the same five-step workflow: SCAN, PRIORITIZE, PROTECT, EXTRACT, VERIFY. You will see this loop again in Topics 2, 3, and 4. The order matters. Skipping PROTECT is how people break production with refactors.

During Hands-On — Three Steps at a Glance

- STEP 1 — Scan and protect. Find the long method and generate characterization tests. ~5 min
- STEP 2 — Extract helpers. Drive the AI through a sequence of named refactorings. ~7 min
- STEP 3 — Verify. Run the tests against the new structure and read the diff together. ~3 min

Step 1 — Scan, Protect, and Prepare to Refactor

WHAT TO LEARN

The refactoring itself is the easy part. The hard part is being sure you didn't change behavior. Characterization tests are the safety net — they capture what the function currently does, including any quirks, so that the refactor cannot silently change behavior. Generating them with AI takes about two minutes and saves you hours of debugging later.

Open topic1_before.py and run it once to confirm the smoke test works:

```
python3 topic1_before.py
```

You should see a printed order record. Now point your AI assistant at the file and ask:

```
> Scan topic1_before.py. List every function longer than 30 lines,
> ranked by number of distinct responsibilities. For the worst offender,
> name each responsibility in 4 words or fewer.
```

The AI should identify process_order as the long method and list its responsibilities: validation, inventory, pricing, discounting, tax, payment, notification, persistence. Point to the section header comments inside the function — the function is literally telling the room it wants to be split.

Now generate the characterization tests. The prompt language matters:

```
> Write pytest tests that capture the current behavior of process_order.
> Do not change the function. Cover the happy path, the gold-tier discount,
> the SAVE5 and SAVE10 coupons, the empty-order error, the negative-quantity
> error, the insufficient-stock error, and verify the record is saved to the
database.
```

Save the output as topic1_tests.py and run it:

```
pytest topic1_tests.py -v
```

WHAT YOU SHOULD GET

All tests pass. If any test fails, the AI's understanding of the function is wrong — review and correct before continuing. This is the moment to catch misunderstandings, not after you've refactored. The reference test file is in the source repo if you need it.

Step 2 — Extract Named Helpers, One at a Time

WHAT TO LEARN

The temptation is to ask the AI to 'refactor process_order.' Resist it. Vague prompts produce vague rewrites. Instead, walk through the section header comments in the function and extract one responsibility at a time, naming each helper as you go. This gives you a sequence of small, reviewable diffs instead of one giant rewrite you have to trust.

Use this prompt pattern for each section. Notice the scope-limiting phrases:

```
> Apply Extract Function to the validation block (lines 70-77) in
> topic1_before.py. Name the new function validate_order. It should
> take the order and raise the same exceptions in the same cases.
> Show me the diff only — do not rewrite the whole file.
```

Repeat the same pattern for each of the remaining responsibilities. After each extraction, run the test suite again:

```
pytest topic1_tests.py -v
```

The full sequence to walk through:

- Extract validate_order — the input validation block.
- Extract reserve_inventory — the stock-check and reservation loops.
- Extract calculate_totals — subtotal, discount, tax, and total math (this one can itself decompose further).
- Extract charge_customer — the payment-gateway stub.
- Extract send_confirmation — the email construction.
- Extract save_order — the database-record assembly and save call.

WHAT YOU SHOULD GET

After all six extractions, process_order itself is six lines that call the six new helpers in sequence. The test suite still passes — exactly. If a test starts failing, the AI changed behavior during one of the extractions. Revert that step and try again with a tighter prompt.

Step 3 — Verify and Read the Diff

WHAT TO LEARN

A refactor isn't done when the tests pass — it's done when a human has read the new code and agreed that it's clearer than the old code. AI can write the diff. Only you can decide whether it's an improvement. Use the AI as a second pair of eyes to read the diff back to you.

Compare the file to topic1_after.py in the source repo:

```
diff topic1_before.py topic1_after.py | less
```

Then ask the AI a final, evaluative question:

```
> Compare the original process_order to the refactored version.
> Are the names accurate? Is the top-level orchestrator at one level
> of abstraction? Where would a new contributor add a new discount type?
```

The third question is the real test. If your refactor was good, the answer is obvious: add a new entry to COUPON_VALUES, or extend calculate_discount. If the AI can't find where to make the change, the refactor isn't done.

WHAT YOU SHOULD GET

A six-line orchestrator, seven small named helpers, all nine characterization tests passing, and a clear answer to 'where would I add a new feature.' That last answer is the deepest payoff of refactoring — the cost of future change just dropped.

Wrap-Up — The Pattern, The Assignment Handoff

REMEMBER

Refactoring is not optimization. It is not modernization. It is not making code prettier. It is changing structure without changing behavior, so that the next person who reads the code (possibly you in six months) can understand it faster and change it more safely. Every refactoring topic in this course is a variation on that one idea.

Key takeaways:

- The five-step workflow — SCAN, PRIORITIZE, PROTECT, EXTRACT, VERIFY — applies to every refactor in this course.
- Generating characterization tests BEFORE refactoring is what makes aggressive structural change safe.
- Scope-limiting prompts ('show me the diff only', 'do not rewrite the whole file') give clean, reviewable changes.
- Extract one responsibility at a time, run the tests between extractions, and stop when the top-level function reads like a recipe.

Source File Reference

The full source for this demo lives at https://github.com/AI-Native-Engineer-Course/AI-Native-Refactorer/tree/main/topic1/ in the course repository. It contains topic1_before.py (the 90-line function), topic1_after.py (the refactored target state), and topic1_tests.py (the reference characterization tests). Open all three files once before class and walk through the three steps so you know where to pause for questions.