Module 4: Intelligent Refactoring and CI-CD, Lesson 1: The AI-Refactoring Engine

Hands-On: Diagnosing user_account_service.py — From Health Report to Refactor Backlog

Overview

This demo proves three claims from the lecture in real time:

- AI pattern-matches code smells the way a 20-year veteran does — when you give it the right context window
- Subjective smell findings get teeth only when you cross-check them with objective metrics: cyclomatic complexity and git change frequency
- The output of a diagnosis session is not opinions — it is a prioritized backlog and a safety-net test plan

You will run three prompts against `user_account_service.py` (~200 lines, deliberately messy) and produce three artifacts: a health report, a priority matrix, and a characterization test list. The student lab extends the same workflow to a multi-file directory.

Before You Start

- Open your AI tool of choice: Claude.ai, Cursor, or GitHub Copilot Chat
- Have a terminal ready with: `pip install radon` (Python complexity scoring)
- Pull the course repo and `cd` into `module4/lecture1/`. Confirm `user_account_service.py` is present
- Confirm git history exists: `git log --follow user_account_service.py` should return commits
- Keep all three prompts in the same chat session — context continuity is the whole point

> **Remember**
> Diagnosis comes before treatment. We are not refactoring anything in this lecture. We are building three artifacts that tell us what to refactor, in what order, and with what safety net. Resist the urge to fix anything you find — write it down and move on.

During Hands-On — Keep Visible if Possible

The Health Report prompt has four required outputs:

```
1. SMELL   — name of each code smell detected
```

```
2. LINES    — specific line numbers affected
3. RISK     — low / medium / high regression risk
4. FIX      — one-sentence recommended remediation
```

Plus one structural requirement — format everything as a single markdown table.

Prompt 1 — The Health Report

> **What to Learn**
>
> The role assignment is what changes the answer. “You are a senior software architect” puts the AI in pattern-matching mode against millions of codebases. The structured output requirement is what makes the result actionable instead of conversational — a table you can sort, paste into a backlog, and act on.

Paste this into the AI chat:

```
ROLE: You are a senior software architect with 20 years of
experience reviewing legacy production code.

CODE: <paste full contents of user_account_service.py here>

TASK: Diagnose this file for code smells. Identify every
instance of the six canonical smells:
  1. Long Method
  2. God Class
  3. Shotgun Surgery
  4. Feature Envy
  5. Primitive Obsession
  6. Dead Code

For each instance found, return exactly four fields:
  - SMELL:  the named smell from the list above
  - LINES:  specific line numbers (e.g., 47–93)
  - RISK:   low / medium / high regression risk
  - FIX:    a one-sentence remediation

OUTPUT: Format the entire response as a single markdown
table with columns SMELL | LINES | RISK | FIX. No prose
before or after the table. No commentary. Just the table.
```

> **What you should get**

> A markdown table with 6–10 rows. Long Method and God Class will dominate. The risk column will not be uniform — some smells are high-risk because the code touches authentication or money, others are low-risk because they live in dead branches. That variance is the signal you want.

Follow-up prompt — same session, do NOT start a new chat:

```
Of the smells you just listed, which poses the highest
regression risk and why? Answer in two sentences.
```

You should ask:

"Why ask the AI to rank when the table already has a risk column?"

Answer: Risk in the table is a single-cell judgment per smell, in isolation. Asking for the *highest-risk* item forces the AI to reason about *interactions* — a Long Method that lives inside a God Class that handles authentication is more dangerous than the sum of its individual smells. The table tells you what is wrong; the follow-up tells you what to fix first.

Save the table: Copy the table into `health_report.docx` in the repo. This is artifact one of three.

Prompt 2 — Cross-Check with Numbers

> **Remember**
> The smells from Prompt 1 are pattern-based. They tell you what feels wrong. Cyclomatic complexity and git change frequency tell you how badly and how often it matters. Without numbers, the health report is an opinion. With numbers, it is a triage decision.

Run these in your terminal first:

```
# Cyclomatic complexity per function
radon cc user_account_service.py -s -a
or on windows: radon cc user_account_service.py -s -a

# Change frequency over project history
git log --oneline --follow -- user_account_service.py | wc -l
```

Note any function rated C, D, E, or F by radon (complexity > 10). Note the commit count.

Continue in the same chat session:

```
Two new inputs for you:

1. Radon cyclomatic complexity output:
   <paste radon output>

2. Git change frequency for this file:
   <paste commit count, e.g., "47 commits in last 18 months">

TASK: Combine these numbers with your earlier health report
to produce a priority matrix. Use these axes:
  - X axis: change frequency (low / medium / high)
  - Y axis: complexity + smell severity (low / medium / high)

Place each function from the radon output on the matrix.
Highest priority is top-right (changes often AND is complex).
Lowest priority is bottom-left (rarely touched, simple).

OUTPUT: A markdown table with columns
  FUNCTION | COMPLEXITY | FREQUENCY | QUADRANT | PRIORITY (1-5)
sorted by PRIORITY ascending (1 = fix first).
```

> **What you should get**
> A ranked list. The top is almost always one or two functions that are both touched constantly AND score above 15 on cyclomatic complexity. Those are the functions silently generating production risk. Anything in priority 4 or 5 — leave it alone for now.

Save the matrix: This is artifact two of three. Save as `priority_matrix.md`.

***Tip:** If your radon scores and the AI's earlier subjective risk levels disagree by more than one band, the disagreement is itself a finding — investigate that function manually. Numbers and patterns should agree on truly broken code; when they don't, one of them is wrong.*

Prompt 3 — The Safety Net

> **What to Learn**
> We do not refactor without a safety net. The next lecture will rewrite the priority-1 function. Before we touch a single line, we need characterization tests — tests that pin the current behavior, bugs included. This last prompt produces the test plan.

Same session, third and final prompt:

```
Take the priority-1 function from the matrix above.

TASK: List the minimum set of characterization tests I
need to write before I refactor this function. The goal
of these tests is NOT to verify correctness — it is to
capture current behavior so that a refactor cannot
silently change it.

Include:
  - Happy-path inputs and expected outputs
  - Edge cases the function currently handles
  - Edge cases the function currently MIS-handles
    (preserve the bug — characterization tests document
    behavior, not correctness)
  - Side effects to assert against (DB writes, external
    calls, log statements, exceptions raised)

OUTPUT: A numbered list of test cases. For each, give:
  - test name
  - input
  - expected output OR expected side effect
  - one-line note on what behavior this test pins
```

> **What you should get**
> Six to twelve numbered test cases. At least one of them should explicitly preserve a known-buggy behavior — that is the AI flagging that the function currently does something you may not have intended, and warning you not to silently "fix" it during refactor. That note is gold. Read it carefully before you continue.

Save the test plan: Artifact three of three. Save as `characterization_tests.md`. In the next lecture, each numbered case becomes an actual pytest function — written before any refactoring begins.

Wrap-Up — Three Prompts, Three Artifacts, Zero Refactors

> **Remember**
> Three prompts. One session. Three artifacts: `health_report.md`, `priority_matrix.md`, `characterization_tests.md`. Together they answer *what is broken*, *what to fix first*, and *what tests must exist before we touch anything*. That is a complete diagnosis. We have not refactored a single line — and that is the point.

Key takeaways:

- Diagnosis before treatment — always
- Pattern-based findings (smells) and number-based findings (complexity, frequency) must agree before you trust either
- The output of diagnosis is not a report — it is a prioritized backlog plus a test plan
- Session continuity matters — the AI carries the file, the smells, and the matrix forward into the test prompt for free
- Characterization tests pin behavior, not correctness — the bugs come along for the ride until the refactor is intentional

In the next lecture, we open `priority_matrix.md`, pick the priority-1 function, write the characterization tests as real pytest, and apply the Strangler Fig pattern to rewrite it safely. See you there.