Module 4: Intelligent Refactoring and CI-CD, Lesson 2: Prompting for Clean Architecture

Hands-On: Strangling the God Class — Live Prompt-Chained Refactor

---

Overview

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

- Refactoring is structural change with preserved behavior — and AI can do it safely when you constrain the prompt
- Prompt chaining turns one scary God-Class refactor into four small, reviewable, reversible steps
- Characterization tests are the cheap, professional safety net that lets you merge with confidence

You will decompose a small OrderManager God Class into focused services using exactly four prompts in one chat session, then generate a characterization test suite that proves behavior was preserved. The student lab extends this to a second extracted responsibility and a circular dependency fix.

Before You Start

- Open your AI tool of choice: Claude.ai, Cursor, or GitHub Copilot Chat
- Have a terminal ready with: pip install pytest pytest-mock
- Create a working directory with one file: order_manager.py (starter code shown in the box below)
- Keep the five-part refactor framework visible on a second monitor throughout
- Do NOT start a new chat between prompts — context continuity is what makes the chain work

> **Remember**
> Refactoring is not rewriting. Every prompt in this chain constrains the AI to a single, specific change with the original signatures preserved. If you cannot describe the change in one sentence, you are about to ask for a rewrite — stop and narrow the scope.

During Hands-On — Keep Visible if Possible

Every refactor prompt in this lecture follows the same five parts:

```
1. SCOPE        — exactly which class / method / smell, never "rewrite this"
2. CONSTRAINTS  — public signatures preserved, dependencies injected
3. PATTERN      — what refactoring (extract class, dep inversion, etc.)
4. OUTPUT       — diff with inline comments, only changed methods
5. VALIDATION   — tests assert consistency with original, not correctness
```

Setup — Paste the God Class into the Chat

> **What to Learn**
> Before any refactor prompt, the AI needs the starting code in context. Paste it once at the top of the session — every subsequent prompt then references “the OrderManager class above” without re-pasting. That is session continuity working for you.

Paste this into a fresh chat (and save it as order_manager.py):

```
class OrderManager:
    """Handles the entire order lifecycle. Yes, all of it."""

    def __init__(self, db, smtp_client, payment_gateway):
        self.db = db
        self.smtp = smtp_client
        self.gateway = payment_gateway
        self.audit_log = []

    def place_order(self, user_id, items, card_token):
        # 1. price the order
        total = sum(i["price"] * i["qty"] for i in items)

        # 2. charge the card
        charge_id = self.gateway.charge(card_token, total)
        if not charge_id:
            return {"ok": False, "reason": "payment_failed"}

        # 3. decrement inventory
        for item in items:
            row = self.db.fetch("products", item["sku"])
            if row["stock"] < item["qty"]:
                self.gateway.refund(charge_id)
                return {"ok": False, "reason": "out_of_stock"}
            self.db.update("products", item["sku"],
                           {"stock": row["stock"] - item["qty"]})

        # 4. persist the order
        order_id = self.db.insert("orders", {
            "user_id": user_id, "items": items,
            "total": total, "charge_id": charge_id,
        })

        # 5. send confirmation email
        user = self.db.fetch("users", user_id)
```

```
        body = f"Hi {user['name']}, order {order_id} confirmed. Total: ${total}."
        self.smtp.send(to=user["email"],
                       subject=f"Order {order_id} confirmed", body=body)

        # 6. write to audit log
        self.audit_log.append({"type": "order_placed",
                               "order_id": order_id, "user_id": user_id})

        return {"ok": True, "order_id": order_id, "total": total}
```

> **What to point out**
> Six numbered comments inside one method. That is the smell. Each comment is a hint at a separate responsibility — pricing, payment, inventory, persistence, notifications, audit. We are going to make the AI confirm that map for us in the very next prompt.

Prompt 1 — Inventory the Responsibilities

> **What to Learn**
> Step one of the chain is always the same: ask the AI to read the class and list the distinct responsibilities. No code changes yet. You are building a map you can refer to for the rest of the session — and you are letting the AI commit to its own analysis before you ask it to act on it.

Paste this into the AI chat:

```
SCOPE: The OrderManager class above. Read-only analysis — do not
modify any code yet.

CONSTRAINTS: Treat the public method `place_order` as a fixed
contract. Its signature and return shape must not change.

PATTERN: Single Responsibility analysis. I am preparing to
decompose this God Class via prompt chaining.

OUTPUT: A numbered list of the distinct business responsibilities
currently mixed inside this class. For each one, give:
  - a short name (e.g. "NotificationService")
  - the lines / comment-numbers it covers
  - the dependencies it would need if extracted

VALIDATION: Do not propose code. Do not collapse two
responsibilities into one. If you are unsure whether something
is its own responsibility, list it separately and flag it.
```

> **What you should get**

> A list of five or six responsibilities — pricing, payment, inventory, order persistence, notifications, audit logging — each with the comment number(s) it covers and the collaborator it depends on. That list is now your refactor backlog.

You should ask:

"Why didn't we just ask it to extract everything in one prompt?"

Answer: because a single prompt that does six extractions at once is unreviewable, untestable, and unrevertable. Four small diffs is what professional refactoring looks like — and prompt chaining is how you get there.

Prompt 2 — Extract One Responsibility (Strangler Fig in Miniature)

> **Remember**
> Same chat session. The AI already has the code and the responsibility list in context — we don't repeat them. We pick one item from the list — Notifications is the cleanest first cut — and tell the AI to build the new module alongside the old one. The original is not touched yet. That is the Strangler Fig idea applied at class scale.

Continue in the same chat — do NOT start a new one:

```
SCOPE: From the responsibility list above, extract item
"NotificationService" only. Do NOT extract any other
responsibility in this prompt.

CONSTRAINTS:
  - Create a new class NotificationService in a new file
    notification_service.py.
  - Use constructor injection for the smtp_client and a db
    accessor for user lookups (no globals).
  - Public method signature: send_order_confirmation(order_id,
    user_id, total) -> None.
  - Do NOT modify order_manager.py in this prompt.

PATTERN: Extract Class. New module lives alongside the old
code; the old code is untouched until Prompt 3.

OUTPUT: The full contents of notification_service.py only.
Add a docstring to the class and the public method explaining
the boundary the class enforces.

VALIDATION: The new class must produce byte-identical email
subject and body strings to what the current OrderManager
produces. Do not "improve" the wording.
```

> **What you should get**

A small, focused NotificationService class with two injected dependencies and one public method. The body and subject strings must match the originals exactly — that is what makes the next step safe. Save the file. Do not run anything yet.

Watch for these failure modes:

- Scope creep — AI tries to also extract PaymentService. Reply: “Notifications only. Show me only notification_service.py.”
- Interface drift — AI changes the email body wording “for clarity.” Reject and re-prompt with the byte-identical constraint.
- Hallucinated dependencies — AI imports jinja2 or a templating library that wasn’t in the original. Reject; we are preserving behavior, not modernizing it.

Prompt 3 — Generate the Reviewable Delegation Diff

What to Learn

Now we modify the original class — but only the parts that change. We ask for a unified diff with inline comments, which is exactly what your code reviewer wants to see. The reviewer reads architectural intent, not implementation details. Review time drops; review quality goes up.

Third prompt — still the same session:

```
SCOPE: order_manager.py only. The public method place_order
and its return shape must remain identical.

CONSTRAINTS:
  - Inject NotificationService via the constructor.
  - Replace the inline email-sending block (comment 5) with a
    single call to the injected service.
  - Do NOT change pricing, payment, inventory, persistence, or
    audit code.
  - Do NOT delete the smtp_client parameter from __init__ yet —
    a later refactor will remove it. Mark it with a comment:
    # TODO: remove after all notification call sites migrated.

PATTERN: Delegate-then-strangle. The old method now routes to
the new class. The old smtp dependency stays wired until every
caller is migrated; only then does the legacy parameter die.

OUTPUT: A unified diff of order_manager.py only. Above each
changed block, add a one-line `# why:` comment explaining the
architectural reason for that change. Show only changed
methods, not the whole file.

VALIDATION: After the diff is applied, place_order must
produce identical observable behavior for any input.
```

What you should get

A small unified diff — typically eight to fifteen lines — with `# why:` comments on each changed block. Paste that diff straight into your pull request description. The reviewer can now evaluate whether your reasoning is sound rather than reverse-engineering it from the code.

Apply the diff and run a smoke check:

```
python -c "from order_manager import OrderManager; print('imports ok')"
```

Prompt 4 — Characterization Tests for Behavioral Equivalence

What to Learn

This is the safety net. Characterization tests do not assert that the current behavior is correct — they assert that it is consistent. You run them before the refactor and they pass (they are recording reality). You run them again after, and if any fail, you changed behavior. That is the entire professional standard, in two test runs.

Fourth prompt — still the same session:

```
SCOPE: Generate a pytest characterization test suite for the
place_order method as it exists in the current
(post-refactor) order_manager.py.

CONSTRAINTS:
  - Use pytest-mock to stub db, smtp_client, payment_gateway,
    and the new NotificationService.
  - Capture every observable side effect: payment charges,
    refunds, inventory writes, order inserts, notification
    calls, audit-log entries, and the returned dict.

PATTERN: Characterization testing. Tests document current
behavior — they are NOT correctness assertions.

OUTPUT: A single test file test_order_manager_characterization.py
with one test per observable behavior (one file with five to
seven test functions is the right size). Above each test,
add a one-line comment of the form:
    # captures: <observable behavior being pinned>

VALIDATION: Do NOT assert on what the behavior should be.
Assert on what it currently IS. If a value seems wrong to
you, lock it in as-is and add a `# suspect:` comment.
```

What you should get

> A test file you can run immediately. At least one test may need a small fix on the first run — that is fine and expected. The point is that you now have an executable specification of the current behavior, and any future refactor that breaks it will fail loudly.

Run the tests:

```
pytest test_order_manager_characterization.py -v
```

If a test fails because the AI guessed wrong about the current behavior, do NOT fix the production code — fix the test to match what the code actually does. That is the discipline. The test is meant to mirror reality, even reality you don't love.

***Tip:** Commit at every step in the chain. Four prompts, four commits. If Prompt 3 produces a diff you don't love, you `git reset` to the end of Prompt 2 and re-prompt — you have not lost the rest of your work.*

Wrap-Up — The Pattern, The Assignment Handoff

> **Remember**
> Four prompts. One session. We took a 50-line God Class, surfaced its responsibilities, extracted one of them into a focused service, generated a reviewable delegation diff, and pinned the new behavior down with characterization tests. The next responsibility is one prompt away. That is the rhythm of AI-driven refactoring done right.

Key takeaways:

- Refactor, don't rewrite — every prompt in the chain has a single, sentence-sized scope
- Strangler Fig at class scale — the new module lives alongside the old until every caller is migrated
- Five-part refactor prompt: scope, constraints, pattern, output, validation
- Reviewable diffs with `# why:` comments turn implementation review into architectural review
- Characterization tests pin current behavior — they are the cheap, professional way to merge with confidence

Student lab: extend the chain. Run the same four-prompt sequence to extract a second responsibility (PaymentService is a good candidate), then write a fifth prompt that breaks a deliberate circular dependency between OrderManager and your new services using dependency inversion.