The AI-Native Refactorer

Duplicated Code

Hands-On: Finding and Consolidating Hidden Duplicates with AI

Overview

This hands-on proves the three core claims from the Topic 2 lecture in real time, using an AI coding assistant and a Python file that contains all three flavors of duplication discussed in the lecture:

- AI can find duplicates humans physically cannot see — including conceptual duplicates with completely different implementations.
- Consolidation reveals drift — copies that have diverged force you to consciously choose canonical behavior.
- The Rule of Three is the right discipline — AI doesn't change when to abstract, just how to find the third instance.

Before You Start

- Have Python 3.10+ installed. Verify with python3 --version.
- Install pytest: pip install pytest.
- Open topic2_before.py from https://github.com/TheAI-Native/AI-NativeRefactorer
- Have an AI coding assistant ready in your editor.
- Have a terminal open in the same directory.

REMEMBER

Duplication is fundamentally a pattern-matching problem, which is exactly what AI does best. Your job in this hands-on is not to find duplicates yourself — it is to drive the AI to find them and to make the right judgments when the AI's suggested consolidation reveals drift in your code.

During Hands-On — Three Steps at a Glance

- STEP 1 — Scan with the AI for all three flavors of duplication. ~4 min
- STEP 2 — Protect with tests, then consolidate the canonical implementations. ~7 min

- STEP 3 — Confront the drift — decide which behaviors survive. ~4 min

Step 1 — Scan With the AI

WHAT TO LEARN

There are three flavors of duplication: identical (same code, different file), structural (same shape, different names), and conceptual (different code, same intent). Identical and structural duplicates can be found with grep or static analysis. Conceptual duplicates can only be found by understanding intent — which is where AI's pattern matching is uniquely valuable.

Run the file first to confirm everything works:

```
python3 topic2_before.py
```

Point your AI assistant at topic2_before.py and ask three increasingly difficult questions:

```
> Find IDENTICAL duplicates in topic2_before.py — functions whose
> bodies are character-for-character the same except for the name.
```

Expected: the AI identifies format_user_phone and format_contact_phone as identical.

```
> Find STRUCTURAL duplicates — functions whose bodies have the same
> shape and intent but differ in names, constants, or error behavior.
```

Expected: the AI identifies validate_user_email, validate_contact_email, and check_email as a structural duplicate cluster. It should point out that they differ in max length, in whether empty input raises, and in stripping behavior.

```
> Find CONCEPTUAL duplicates — functions that perform the same task
> but use entirely different implementations.
```

Expected: the AI identifies user_is_recent (uses a cutoff comparison) and contact_is_recent (iterates day-by-day) as conceptual duplicates. This is the duplicate humans miss. Linters miss it. Static analysis misses it. AI catches it because it understands intent.

WHAT YOU SHOULD GET

A list of seven near-duplicate functions clustered into three groups. The conceptual-duplicate group is the moment to pause for the room. Tell them: "This is the duplicate your team has been ignoring for three years because nobody knew it existed."

Step 2 — Protect and Consolidate

WHAT TO LEARN

Characterization tests for duplicate-code refactors must cover ALL copies, not just one. If the copies have drifted, the tests will document the drift — and that is exactly what you want because the drift is the thing you have to make a decision about.

Generate the test suite:

```
> Write pytest tests that capture the current behavior of every function
> in topic2_before.py. Include tests that document any behavioral DIFFERENCES
> between the near-duplicates — those drift points are the important ones.
```

Save as topic2_tests.py and run:

```
pytest topic2_tests.py -v
```

All tests should pass. Among them, you should see tests with comments like:

- "DRIFT TEST: contact validator has stricter length limit"
- "DRIFT TEST: check_email lowers but does NOT strip whitespace — a quiet bug"

These drift tests are the heart of the exercise. The reference test file has them already if you need it.

Now consolidate. Use the strangler-fig pattern:

```
> Propose a canonical implementation that replaces validate_user_email,
> validate_contact_email, and check_email. Identify the parameters
> needed to capture the legitimate variations. Keep the old functions
> as thin wrappers for backward compatibility.
```

Repeat for the other two clusters:

- format_user_phone and format_contact_phone → format_phone (identical, just rename and delegate).
- user_is_recent and contact_is_recent → is_recent(record, days=30) (pick the cutoff-comparison version).

WHAT YOU SHOULD GET

Three canonical functions: validate_email, format_phone, is_recent. The seven original functions still exist as one-line wrappers calling the canonical version. Run the tests again —

they should still pass against the original behaviors, including the drift behaviors, because the wrappers preserve them.

Step 3 — Confront the Drift

WHAT TO LEARN

Consolidation surfaces every drift in your codebase. Some of those drifts are legitimate variations and should become parameters. Some are bugs that survived because nobody noticed the duplication. You have to decide which is which — the AI cannot make this judgment for you.

Walk through each drift and ask: feature or bug?

Drift 1: contact emails have a 200-character limit, user emails 254.

Decision: probably a feature. Different fields legitimately have different constraints. Keep the parameter. Update the contact-side wrapper to pass max_length=200 explicitly.

Drift 2: validate_contact_email accepts empty input where validate_user_email rejects it.

Decision: probably a bug. Either every email field should reject empty, or every one should treat it as optional. Pick one and apply consistently. The canonical version uses a required=True/False parameter.

Drift 3: check_email doesn't strip whitespace.

Decision: definitely a bug. Nobody wants a leading space in a stored email. The canonical version always strips. This means the corresponding drift test should be UPDATED, not preserved.

Ask the AI to update only the tests for the drifts you've decided are bug fixes:

```
> Update the tests for check_email to expect the new behavior — that
> whitespace IS stripped. Mark this change with a comment explaining
> the bug fix.
```

WHAT YOU SHOULD GET

All tests passing, three canonical implementations, seven wrappers that delegate, and a clear audit trail in the test file showing every drift decision. The codebase is now structurally consistent in a way it has never been before.

Wrap-Up — The Pattern, The Assignment Handoff

REMEMBER

Duplicated code is not a code smell — it is a multiplier on every future change. Consolidation is not just about reducing line count. It is about creating a single source of truth that the next bug fix only has to touch in one place. Every duplicate you consolidate today saves N future engineers from N future bugs.

Key takeaways:

- Three flavors of duplication: identical, structural, conceptual. AI catches all three.
- Characterization tests must cover ALL copies — they document the drift you'll have to resolve.
- Consolidation forces conscious decisions about which behavior is canonical.
- The strangler-fig pattern (canonical implementation + thin wrappers) lets you consolidate without breaking callers.

Source File Reference

The full source for this demo lives at https://github.com/AI-Native-Engineer-Course/AI-Native-Refactorer/tree/main/topic2/. It contains topic2_before.py (the seven near-duplicates), topic2_after.py (the consolidated target), and topic2_tests.py (the characterization tests including the drift markers). Open all three before class so you know where the drift tests live.