The AI-Native Refactorer

Topic 4: Tight Coupling

Hands-On: Inverting Dependencies to Unlock Testability

Overview

This hands-on proves the three core claims from the Topic 4 lecture using an AI coding assistant and a 100-line Python file that exhibits the canonical tight-coupling anti-pattern:

- Tight coupling is recognizable from the constructor alone — a class that creates its own dependencies is coupled to all of them.
- Dependency injection converts coupling from a structural problem into a configuration problem — and AI can do the entire conversion in one pass.
- Once dependencies are injected, unit tests collapse from "impossible without three services" to "trivial with three fakes."

Before You Start

- Have Python 3.10+ installed.
- Install pytest: pip install pytest.
- Open topic4_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

Coupling refactors are different from the other three topics in one important way: the very tests that would protect you are usually what motivated the refactor in the first place — because you couldn't write them. Accept that you'll work in smaller, riskier steps than for long methods or duplicate consolidation. The payoff is huge — but the path takes care.

During Hands-On — Three Steps at a Glance

- STEP 1 — Map dependencies and identify what makes the class untestable. ~4 min
- STEP 2 — Extract interfaces, inject dependencies, add a composition root. ~7 min
- STEP 3 — Prove the win by writing unit tests with fakes. ~4 min

Step 1 — Map the Dependencies

WHAT TO LEARN

Before you can fix coupling, you have to see it. The smell shows up most clearly in the constructor: a class that takes zero arguments but creates three dependencies is the canonical pattern. AI can map the dependency graph for you, and — more importantly — point out the implicit ones: environment variable reads, singletons, global state.

Run the file to confirm it works end-to-end:

```
python3 topic4_before.py
```

Ask the AI to map the dependencies:

```
> Read topic4_before.py and answer three questions: (1) what does
> OrderProcessor depend on? (2) where are those dependencies created?
> (3) what hidden dependencies exist (environment variables, globals)?
> List everything I would need in order to test process_order in isolation.
```

Expected: the AI lists PostgresDb, SendgridEmail, and StripeGateway as the explicit dependencies, and the three environment variables DB_URL, SENDGRID_KEY, and STRIPE_KEY as the hidden ones. It should note that OrderProcessor.__init__ creates all three dependencies internally — the anti-pattern is staring you in the face.

WHAT YOU SHOULD GET

A clear inventory of what would need to be replaceable for the class to be testable: a repository, an email client, and a payment gateway. Three interfaces, three implementations.

Step 2 — Invert the Dependencies

WHAT TO LEARN

Dependency inversion has two parts: (1) define interfaces that capture what the dependent class actually needs (not everything the concrete class can do), and (2) move the construction of concrete dependencies out to the application's entry point. In Python, the interfaces are usually typing.Protocol — structural typing means concrete classes don't need to inherit from anything.

Define the interfaces first:

```
> Define three typing.Protocol interfaces that capture exactly what
> OrderProcessor needs: OrderRepository (with save_order), EmailClient
> (with send), and PaymentGateway (with charge). Make them minimal —
> only include the methods OrderProcessor actually calls.
```

Inject the dependencies into OrderProcessor:

```
> Change OrderProcessor.__init__ to accept three parameters: repository,
> email, and payment, typed against the Protocols. Remove the
> internal construction. The process_order method body should stay
> the same — just use self.repository / self.email / self.payment.
> Show me the diff only.
```

Update the concrete classes to take their config explicitly:

```
> Modify PostgresDb, SendgridEmail, and StripeGateway so that their
> __init__ methods take their config (connection string, api key) as
> a parameter instead of reading os.environ directly. The env-var
> reads should happen only at the app entry point.
```

Add a composition root — a single function that wires everything up:

```
> Add a function called wire() that reads the three environment
> variables, constructs PostgresDb / SendgridEmail / StripeGateway
> with their configs, and returns a fully-wired OrderProcessor.
> This is the ONLY place where concrete dependencies should be created.
```

WHAT YOU SHOULD GET

An OrderProcessor whose constructor takes three Protocol parameters, three concrete classes that no longer read environment variables themselves, and a wire() function at the bottom that does all the gluing. Production behavior is unchanged. The class is now testable.

Step 3 — Prove the Win

WHAT TO LEARN

The real proof that a coupling refactor worked is that you can now do something you couldn't before. Writing a unit test with fakes — one that never touches a network, never reads env vars, runs in milliseconds — is that proof.

Ask the AI to write trivially-simple test doubles:

```
> Write three minimal fake classes for testing: FakeRepository,
> FakeEmail, FakePayment. FakePayment should take an `approved` flag
> so we can test both happy path and failure. Each fake should record
> what was called so the tests can assert on it.
```

Then write the unit tests:

```
> Write pytest unit tests that exercise OrderProcessor with the fakes:
> (1) happy path — payment approves, record saved, email sent;
> (2) payment failure — no record saved, no email sent;
> (3) verify the email subject contains the order id.
> Each test should be 5 lines or fewer.
```

Run them:

```
pytest topic4_tests_with_fakes.py -v
```

WHAT YOU SHOULD GET

Two or three unit tests that pass in milliseconds. No network. No credentials. No mocking framework needed. This is the proof that the refactor worked. Production OrderProcessor and test OrderProcessor are the same class — only the wiring differs.

Wrap-Up — The Pattern, The Course Wrap-Up

REMEMBER

Coupling is what determines what you can change, test, and ship independently. Every dependency you inject today saves a future engineer from having to mock the world. The dependency direction rule — high-level policy never depends on low-level details — is the single deepest architectural idea in object-oriented design. AI lets you apply it in minutes instead of weeks.

Key takeaways:

- Tight coupling is recognizable from the constructor — a class that creates its own dependencies is coupled to all of them.
- Dependency injection + Extract Interface + Composition Root is the standard recipe.
- Python's typing.Protocol gives you interfaces without inheritance.

- The win is proven by writing unit tests with fakes — not by counting lines moved.

Source File Reference

The full source for this demo lives at https://github.com/AI-Native-Engineer-Course/AI-Native-Refactorer/tree/main/topic4/. It contains topic4_before.py (the tightly-coupled OrderProcessor), topic4_after.py (with Protocols, DI, and a wire() composition root), and topic4_tests.py (characterization tests for the before version). The repo also includes a sample fakes-based test file that demonstrates the post-refactor testing pattern.

Where to go next

You've now completed all four topics of The AI-Native Refactorer. The full paid course, The AI-Native Engineer, covers another twelve refactoring patterns, advanced AI workflow techniques, and a complete approach to AI-assisted code review and pull-request generation. Visit https://TheAI-NativeEngineer.com to enroll.