The AI-Native Smells Detector, Lesson 3: Immobility

Hands-On: Freezing Trapped Business Logic

Overview

This hands-on shows immobility in its most common form: useful business logic welded to a web framework, a database, and a logger so tightly that you cannot lift it out and reuse it. We will:

- Read a Flask-style route that tangles tax calculation with HTTP, SQLite, and logging.
- Try to imagine reusing that tax logic in a CLI batch job - and watch the tangle make it impossible without a rewrite.
- Refactor the same logic into a pure, framework-free domain function plus a thin adapter.
- Reuse the same function from two adapters (web sketch + working CLI) and from a self-test that runs in milliseconds with zero infrastructure.

Before You Start

- Have Python 3.10+ installed. Verify with python3 --version.
- Open two files from the source directory: tax_before.py and tax_after.py.
- Have a terminal open in the same directory, ready to run each file.
- No Flask or SQLite install is needed - the 'before' file uses commented-out imports and stand-ins so the lecture can focus on the SHAPE of the tangle, not on installing packages.

REMEMBER

Immobility is the smell of code that knows too much about its neighbors. The tax rules in the 'before' file are correct and valuable - the problem is they are surrounded by Flask, SQLite, and a logger they did not need. The fix is not to rewrite the rules. The fix is to lift them out into a pure function, and let the framework call into them instead of the other way around.

During Hands-On - Three Steps at a Glance

- **STEP 1** - Read the 'before' file. Identify every concern tangled into the route. Try to imagine reusing the tax rules. ~5 min

- **STEP 2** - Open the 'after' file. Find the pure domain function and the frozen TaxQuote dataclass. Notice what is NOT in the same file. ~5 min
- **STEP 3** - Run the CLI adapter and the self-check. Discuss what it would take to add a second adapter (mobile API, batch job, scheduled report). ~5 min

Step 1: Read the Tangle. Spot the Welds.

WHAT TO LEARN

Open tax_before.py. You are looking at a single function, calculate_tax_route, that lives inside a Flask app. It looks short. It is not. Inside this one function, four completely different concerns are fused together: pulling values out of an HTTP request, querying SQLite, applying tax rules, and writing a log line. The tax rules - the only part that has actual business value - are surrounded and outnumbered. Read it out loud and count the imports the function would need to run.

Open tax_before.py and read through it. Focus on the comments numbered 1 through 4.

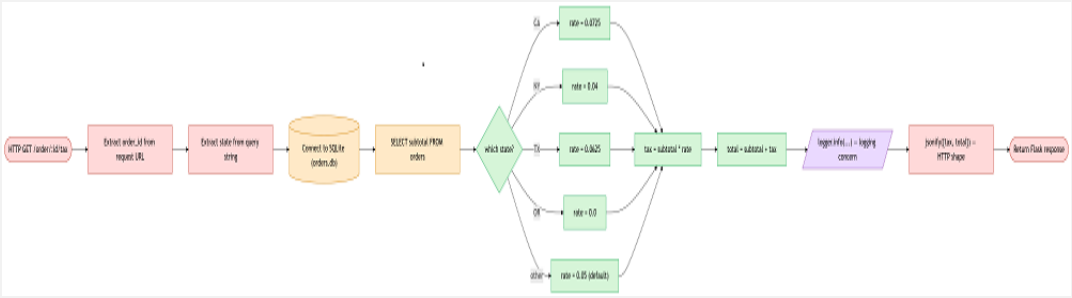
```
# Excerpt from tax_before.py - the tangled route
def calculate_tax_route():
    """
    @app.route("/order/<order_id>/tax", methods=["GET"])
    def calculate_tax_route(order_id):
        ...
    """
    # 1. HTTP concern - pulling values out of the request.
    order_id = "from request URL"
    state = "from request query string"

    # 2. DB concern - talking directly to SQLite.
    # conn = sqlite3.connect("orders.db")
    # cur = conn.cursor()
    # cur.execute("SELECT subtotal FROM orders WHERE id = ?", (order_id,))
    # subtotal = cur.fetchone()[0]
    subtotal = 100.00  # pretend value for demo

    # 3. BUSINESS LOGIC - the only part that has any reuse value.
    if state == "CA":
        rate = 0.0725
    elif state == "TX":
        rate = 0.0625
    elif state == "OR":
        rate = 0.00
    else:
        rate = 0.05
    tax = round(subtotal * rate, 2)
    total = round(subtotal + tax, 2)

    # 4. Logging + HTTP response concern.
```

```
    # logger.info("tax computed", extra={"order": order_id, "tax": tax})
    return {"order_id": order_id, "tax": tax, "total": total}
```



Flowchart for tax_before.py

Run it to confirm it produces a result (the stand-ins fill in for Flask and SQLite):

```
python3 tax_before.py
```

WHAT YOU SHOULD GET

A printed dict that looks like {'order_id': 'from request URL', 'tax': 5.0, 'total': 105.0}. Then ask the class: 'If finance wants to use these exact tax rules in a nightly batch job that has no Flask, no HTTP request, and a different database, what do they do?' The honest answer is: they copy and paste, and now you have two copies that will drift. That is immobility. The rules are trapped.

Step 2: Lift the Logic Into a Pure Function

WHAT TO LEARN

Open tax_after.py. Notice three things. First, the file imports nothing from Flask, sqlite3, or logging - it is pure Python. Second, the tax rules now live in a constant dictionary, TAX_RATES, that anyone can read at a glance. Third, the calculation lives in quote_tax(subtotal, state) - a pure function that takes inputs and returns a TaxQuote dataclass. It has no side effects. It does not know what called it. It will work the same from a Flask route, a CLI, a unit test, or a mobile API. That is mobility.

Open tax_after.py and read through the domain layer:

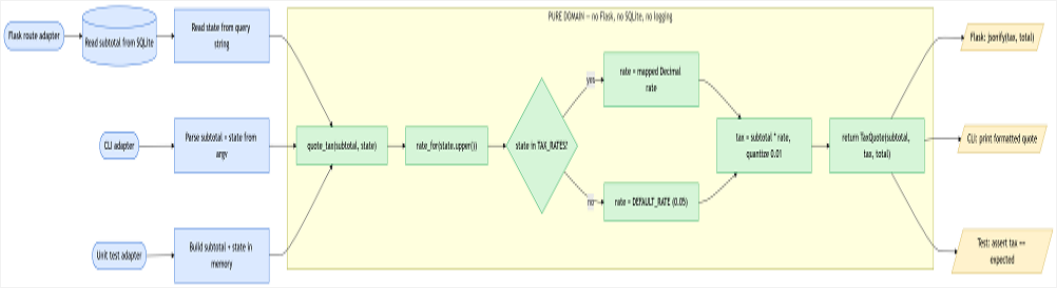
```
# Excerpt from tax_after.py - the pure domain
from dataclasses import dataclass

TAX_RATES = {
    "CA": 0.0725,
    "TX": 0.0625,
```

```
    "OR": 0.00,
}
DEFAULT_RATE = 0.05

@dataclass(frozen=True)
class TaxQuote:
    subtotal: float
    rate: float
    tax: float
    total: float

def quote_tax(subtotal: float, state: str) -> TaxQuote:
    rate = TAX_RATES.get(state.upper(), DEFAULT_RATE)
    tax = round(subtotal * rate, 2)
    total = round(subtotal + tax, 2)
    return TaxQuote(subtotal=subtotal, rate=rate, tax=tax, total=total)
```



Flowchart for tax_after.py

Now look at the adapters lower in the file. The Flask sketch is one short function that calls quote_tax. The CLI adapter is one short function that calls quote_tax. The self-check is one short function that calls quote_tax. Three callers, one rule book.

```
# The web adapter is now THIN.
# @app.route("/order/<order_id>/tax")
# def web_tax(order_id):
#     subtotal = fetch_subtotal_from_db(order_id)
#     state    = request.args.get("state", "")
#     quote    = quote_tax(subtotal, state)   # <-- pure domain call
#     return asdict(quote)
```

WHAT YOU SHOULD GET

An understanding that the SAME tax rules are now callable from anywhere, because they no longer drag a framework, a database, and a logger along with them. Ask the class: 'Where would you put a unit test for this?' The answer is: right next to the function. No fixtures. No mocks. No Flask test client. Just call quote_tax(100, "CA") and assert the result.

Step 3: Reuse Across Adapters. Prove Mobility.

WHAT TO LEARN

Now we exercise the same domain logic from a different adapter - the command line - and from a self-check that asserts the rules behave correctly. Both calls go through quote_tax. Neither knows the other exists. This is what 'mobile' looks like in practice: the same function shows up in multiple deployment shapes without modification.

Run the CLI adapter and the self-check:

```
python3 tax_after.py
```

WHAT YOU SHOULD GET

First line: 'All tax rule tests passed.' Then three lines printed by the CLI showing Subtotal: $250.00, Tax: $10.00, Total: $260.00. The tests ran with no database, no HTTP server, no logger. The CLI ran with no Flask. The web adapter (sketched in comments) would call the SAME quote_tax function, returning the SAME result. One source of truth, three deployment shapes.

Discussion prompt for the room: 'A new request lands - the marketing team wants to send an estimated total in emails. With the BEFORE design, what do you do?' (Answer: copy the rules into the email job, accept drift.) 'With the AFTER design?' (Answer: import quote_tax. Done in two minutes.) That is the dollar value of removing immobility - new use cases land in minutes instead of days.

Wrap-Up

REMEMBER

Immobility is not about how much code you have. It is about how that code is arranged. The same fifteen lines of tax rules went from trapped to reusable without changing a single rule - we just moved them out from under Flask, SQLite, and the logger. The pattern is the same every time: a pure domain in the middle, thin adapters at the edges.

Key takeaways from this hands-on:

- Immobility hides inside functions that import a framework. If your domain logic imports Flask, Django, FastAPI, or a database driver, it is welded in.
- The pure-function-plus-adapters pattern - sometimes called Ports and Adapters or Hexagonal Architecture - is the single most reliable cure.
- Mobility is provable. If you can call the same function from a test, a CLI, and a web route without changing it, you have it. If not, you don't.

- AI coding assistants will happily generate Flask routes that bury logic. Ask them explicitly for 'a pure domain function plus a thin adapter,' and the same model produces the mobile version on the first try.
- When you cure immobility, you usually cure rigidity and fragility at the same time. Pure functions are easier to extend AND harder to break from a distance.

Source File Reference

The two files for this hands-on are tax_before.py and tax_after.py. Both are heavily commented so you can study them outside of class. Run each with python3 to see the behavior described in the WHAT YOU SHOULD GET boxes. In the next lesson we move to Viscosity - the smell where the wrong way to do something is easier than the right way - and we will see how that smell silently undoes the work we just did here.