The AI-Native Smells Detector, Lesson 1: Rigidity

Hands-On: Refactoring a Rigid NotificationService

Overview

This hands-on proves the three core claims from the lecture in real time, using nothing but Python and about a hundred and twenty lines of code:

- A rigid notification service forces every caller and every new channel to edit the same central if/elif chain - the cascade is visible in source.
- The cure is mechanical: a one-method interface, one class per channel, and dependency injection at the composition root.
- Once refactored, adding a brand new channel - Slack - takes one line in the wiring and zero changes to existing code. The Open/Closed Principle stops being theoretical.

You will run two small Python programs side by side. The student assignment at the end extends the refactored version by adding a new channel without touching the existing ones.

Before You Start

- Have Python 3.10+ installed. Verify with python3 --version.
- Open both files from the source directory: notification_service_before.py and notification_service_after.py.
- Have a terminal open in the same directory, ready to run each file step by step.
- Have the lecture slides visible on a second screen if possible - the if/elif chain on Slide 6 is the moment of recognition for the room.

REMEMBER

Rigidity is measured in calendar time, not lines of code. The signal is the GAP between the size of a requested change and the size of the resulting pull request. Every demo step below is a way to make that gap shrink. Watch the pattern repeat in every smell that follows in this course.

During Hands-On - Three Steps at a Glance

- **STEP 1** - Read and run the rigid version. Find the if/elif chain. Show how callers are forced to know channel strings. ~4 min
- **STEP 2** - Walk through the refactor. Introduce NotificationChannel, one class per channel, and dependency injection. ~6 min

- **STEP 3** - Add a SlackChannel WITHOUT touching existing code. Prove the Open/Closed Principle by re-running the test. ~5 min

Step 1 - Reading the Rigid NotificationService

WHAT TO LEARN

A rigid service exposes itself through three signals: (a) a giant if/elif chain dispatching on a string code, (b) hard-coded configuration buried inside the class, and (c) callers throughout the codebase that have to know the magic string. Each new feature multiplies the number of edits required. This is what 'a simple change forces a cascade of changes' looks like in concrete code.

Open notification_service_before.py and run it from the terminal:

```
python3 notification_service_before.py
```

Read the central method with the room. The smell is unmistakable:

```
def send(self, channel, recipient, subject, body):
    if channel == "email":
        # ... 8 lines of SMTP code ...
    elif channel == "sms":
        # ... SMS-specific formatting ...
    elif channel == "console":
        # ... console print ...
    else:
        raise ValueError(f"Unknown channel: {channel}")
```

I used this prompt to ask Claude what it found:
I am a senior software engineer. I have python code that I am afraid might have a bit of Rigidity that I have added to this chat. Can you help me identify the Rigidity and suggest ways to fix it?
Output as a Word document and use a table to organize your findings.

Now scroll down to the callers. Notice that every caller in the codebase has to know the literal string for the channel. on_order_placed calls send("email", ...). on_payment_failed calls send("sms", ...). The string is scattered across the entire system.

WHAT YOU SHOULD GET

Two printed lines from the SMS and console paths. The output is uninteresting; the SHAPE of the code is the point. Pause and ask the room: "If marketing comes in tomorrow and wants Slack notifications, how many files do we touch?" Count them out loud. Add one elif here. Add the new magic string in every caller. Add Slack config to the constructor. Three places minimum, more if the system is older. That is the cascade.

Step 2 - The Refactor

WHAT TO LEARN

There are three moves, and they always go together. First, define a one-method interface - NotificationChannel with a single deliver method. Second, lift each branch of the if/elif into its own class that implements the interface. Third, stop hard-coding configuration: each channel takes its config in __init__, and the service receives the channels through dependency injection. The result is a service that doesn't know what an email is - which is exactly the point.

Open notification_service_after.py. Walk through the four blocks in order:

Block 1 - The interface:

```
class NotificationChannel(ABC):
    @abstractmethod
    def deliver(self, message: Message) -> None: ...
```

Block 2 - One class per channel:

```
class EmailChannel(NotificationChannel):
    def __init__(self, smtp_host, smtp_port, smtp_user, smtp_password): ...
    def deliver(self, message): ...

class SmsChannel(NotificationChannel): ...
class ConsoleChannel(NotificationChannel): ...
```

Block 3 - The service itself is now tiny:

```
class NotificationService:
    def __init__(self, routes: dict[str, NotificationChannel]):
        self._routes = routes

    def notify(self, purpose, message):
        self._routes[purpose].deliver(message)
```

Block 4 - The composition root, where wiring lives:

```
def build_service():
    routes = {
        "order_confirmation": EmailChannel(...),
        "payment_alert":      SmsChannel(...),
        "system_error":       ConsoleChannel(),
    }
    return NotificationService(routes)
```

Run it:

```
python3 notification_service_after.py
```

WHAT YOU SHOULD GET

The same output as before, but produced by a service that no longer knows the channels exist. Notice that callers now ask for a PURPOSE - "order_confirmation" - not a transport. Marketing can swap the channel behind that purpose by editing one line in build_service, without touching a single caller. The cascade is gone.

Step 3 - Add Slack Without Changing Existing Code

WHAT TO LEARN

The Open/Closed Principle says modules should be open for extension but closed for modification. In a rigid codebase that sentence is meaningless because extension always requires modification. After the refactor we can finally cash that check. Adding a new channel is purely additive.

Notice that notification_service_after.py already includes SlackChannel and uses it. Compare the diff in your head: what existed BEFORE we added Slack? Three channels. What changed when we added it? One new class - SlackChannel - and one line in build_service putting it in the routes map. EmailChannel did not change. SmsChannel did not change. ConsoleChannel did not change. NotificationService did not change. NO caller changed.

To make the point even more vividly, ask the room to add a sixth channel - say, a Webhook channel - in real time:

```
class WebhookChannel(NotificationChannel):
    def __init__(self, url):
        self.url = url
    def deliver(self, message):
        print(f"[WEBHOOK {self.url}] {message.subject}")

# In build_service, add ONE line:
routes["audit_event"] = WebhookChannel("https://hooks.example.com/audit")
```

Run it. The new channel works. Nothing else changed.

WHAT YOU SHOULD GET

A working Webhook channel after roughly 30 seconds of edits, all of which were additive. Stop here for the room. This is the moment that justifies the entire refactor. Tell them: this is what your codebase feels like every day when you have removed rigidity. Adding things is fast. Adding things is safe.

Wrap-Up - The Pattern, The Assignment Handoff

REMEMBER

Rigidity is a smell of dispatch. Wherever you find an if/elif chain on a type code, you have found a rigidity hotspot. The fix is mechanical: one interface, one class per variant, one composition root. Once you have done this refactor twice, your hands will do it automatically the next time. Every language has slightly different syntax for the moves. The shape is identical in all of them.

Key takeaways:

- A rigid module is identified by its blast radius - one small ask in plain English produces a large pull request in source code.
- The three classic sources of rigidity are direct coupling, conditional sprawl, and hidden configuration. Every codebase suffers from all three at different spots.
- Polymorphism, dependency injection, and externalized configuration are the three patterns that remove rigidity. They are most powerful when used together.
- After the refactor, adding a new variant is purely additive. That additive-ness is the practical meaning of the Open/Closed Principle.
- AI coding assistants amplify whatever shape your code already has. A well-factored codebase becomes a force multiplier; a rigid codebase becomes a rigidity multiplier. Refactor first, then accelerate.

Source File Reference

The two files for this demo are notification_service_before.py and notification_service_after.py, each roughly a hundred lines of standard Python with no external dependencies. Open both files once before class and walk through the three steps so you know where to pause for questions. The BEFORE file is intentionally a little ugly - that is the point. The AFTER file is intentionally a little verbose - the verbosity buys you a permanent ability to extend the system without touching it.