The AI-Native Smell Detector, Lesson 5: Needless Complexity

Hands-On: From 100 Lines to 3

Overview

This is the final hands-on of the course. We will take a perfectly correct, perfectly tested, six-class implementation of a function that computes an average, and reduce it to three lines that do exactly the same thing. We will:

- Read average_before.py and identify every layer of scaffolding: strategy interface, concrete strategy, factory, config, result type, service, builder.
- Ask the diagnostic questions from slide 9: what does each abstraction protect us from? When has that thing happened? How many subtypes today?
- Open average_after.py and see the same behavior in three lines using the standard library.
- Discuss how this exact transformation applies to YOUR codebase, and how to prompt AI assistants to produce the 'after' version on the first try.

Before You Start

- Have Python 3.10+ installed. Verify with python3 --version.
- Open two files from the source directory: average_before.py and average_after.py.
- Have a terminal open in the same directory, ready to run each file.
- Have the lecture slides visible on a second screen - especially slide 9 (the diagnostic questions) and slide 14 (the AI angle). Refer back as you walk.

REMEMBER

The 'before' file is not bad code. It is correctly written. It has tests. It follows naming conventions. The variables are sensible. The smell is not in any LINE of the file - it is in the GAP between how much machinery the file contains and how much behavior it actually produces. A six-class hierarchy that returns the mean of a list of numbers is the gap. Senior engineers see the gap immediately. The skill of this lesson is teaching the room to see it too.

During Hands-On - Three Steps at a Glance

- **STEP 1** - Read the 'before' file. Count the classes. Trace the call chain from caller to actual arithmetic. ~5 min
- **STEP 2** - Apply the three diagnostic questions from slide 9 to each class. Discover that none of them earn their existence. ~5 min
- **STEP 3** - Open the 'after' file. Three lines. Same behavior. Discuss the prompt that would generate the 'after' version directly. ~5 min

Step 1: Read the Scaffolding. Count the Classes.

WHAT TO LEARN

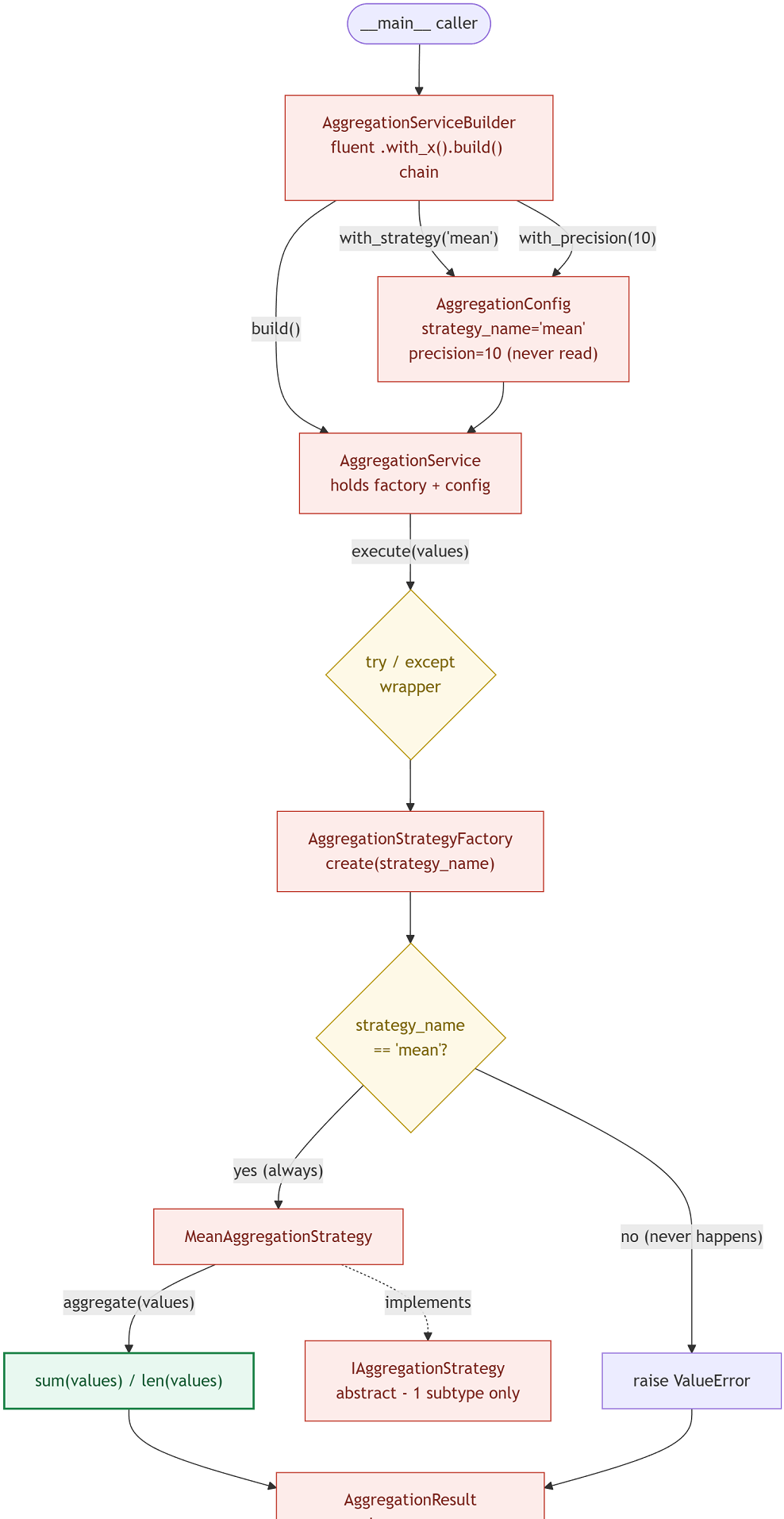
Open average_before.py. The file is around 100 lines. It contains seven named types: IAggregationStrategy (abstract), MeanAggregationStrategy (concrete), AggregationStrategyFactory, AggregationConfig, AggregationResult, AggregationService, and AggregationServiceBuilder. Read each one out loud. Notice how each looks reasonable in isolation. Notice how, taken together, they form a stack that turns a one-line operation into a ceremony.

Skim the file. Here is the call chain you have to follow to compute an average:

```
# Excerpt from average_before.py - the ceremony
# To compute the mean of [10, 20, 30, 40], the caller writes:

service = (AggregationServiceBuilder()
           .with_config(AggregationConfig(precision=2))
           .with_strategy("mean")
           .build())
result = service.aggregate([10, 20, 30, 40])
print(result.value)

# That call chain passes through:
#   1. AggregationServiceBuilder.build()
#   2. AggregationStrategyFactory.create("mean")
#   3. new MeanAggregationStrategy()
#   4. new AggregationService(strategy, config)
#   5. service.aggregate(values)
#   6. strategy.compute(values)
#   7. ...which finally calls sum(values) / len(values)
```


__main__ caller
AggregationServiceBuilder
fluent .with_x().build()
chain
with_strategy('mean')
with_precision(10)
build()
AggregationConfig
strategy_name='mean'
precision=10 (never read)
AggregationService
holds factory + config
execute(values)
try / except
wrapper
AggregationStrategyFactory
create(strategy_name)
strategy_name
== 'mean'?
yes (always)
no (never happens)
MeanAggregationStrategy
aggregate(values)
implements
sum(values) / len(values)
IAggregationStrategy
abstract - 1 subtype only
raise ValueError
AggregationResult

Flowchart for averge_before.py

Run the 'before' file to confirm it works:

```
python3 average_before.py
```

WHAT YOU SHOULD GET

The output: 'Average: 25.0'. The behavior is correct. The implementation is around 100 lines. Ask the room: 'How many of you have written code that looks like this?' Almost every hand will go up. Then ask: 'How many of you have written code that looks like this because you genuinely expected to swap the aggregation strategy?' Almost no hands. That is the smell.

Step 2: Run the Diagnostic. Score Each Class.

WHAT TO LEARN

Apply the three questions from slide 9 to each class. Question one: what does this protect me from? Question two: when has that thing happened? Question three: how many concrete subtypes today? If the answers are vague, never, and one - the class is needless.

Walk the room through the scoring. Read each one aloud:

- **IAggregationStrategy** - Q1: protects against having to change the aggregation algorithm. Q2: never happened. Q3: one implementer. VERDICT: needless.
- **MeanAggregationStrategy** - Wraps a one-line calculation. Exists only because the interface exists. VERDICT: needless.
- **AggregationStrategyFactory** - Q1: protects against having to instantiate the strategy directly. Q2: instantiating directly is fine. Q3: one strategy. VERDICT: needless.
- **AggregationConfig** - Holds a 'precision' parameter that is never actually varied. VERDICT: needless.
- **AggregationResult** - Wraps a single number in a typed object so the caller can call .value. VERDICT: needless.
- **AggregationService** - Orchestrator with one operation, one strategy, one config. VERDICT: needless.
- **AggregationServiceBuilder** - Builds a service whose only required input is the strategy name. VERDICT: needless.

WHAT YOU SHOULD GET

Seven verdicts of 'needless'. Then ask the room: 'If we delete all of this, what behavior do we lose?' Long pause. The answer is: nothing. The behavior is 'return the mean of a list of numbers,' and Python's standard library already does that. Every class in this file is scaffolding around a one-line operation.

Step 3: The 'After' File. Three Lines.

WHAT TO LEARN

Open average_after.py. The file has a self-check test block and a CLI demo, so it is about thirty lines total - but the actual function is three lines. Read it. Notice what is gone: no interface, no factory, no builder, no config, no service, no result type. Notice what is there: a function that takes a list and returns a number. That is the whole behavior. That is all that was ever needed.

The 'after' implementation:

```
# Excerpt from average_after.py - the entire implementation
from statistics import mean

def average(values: list[float]) -> float:
    return mean(values)
```

Run it:

```
python3 average_after.py
```

WHAT YOU SHOULD GET

Output that shows the same answer the 'before' file produced - the average of the same list - plus passing self-check assertions. The behavior is identical. The reading cost is one-thirtieth. There is nothing for a new engineer to learn before they can use this function. There is nothing for a debugger to step through. There is no abstraction to defend in code review. That is the dollar value of removing this smell.

Write a small Python module that computes the average of a list of numbers in a method called average. Call the method from main with [10, 20, 30, 40]

Keep it as simple as the problem is: there's only one operation today, so don't introduce interfaces, factories, config objects, result wrappers, or

classes of any kind. A single function is the whole design.

Guidelines:
- Prefer Python's standard library over hand-rolled arithmetic.
- Use idiomatic exceptions for error cases (e.g. empty input); don't invent a result/status object to carry errors.
- Add a brief comment noting the simplest next step if a second aggregation (like a median) were ever genuinely needed — i.e. add a second small function then, not an abstraction now.
- Include a tiny __main__ example that prints the average of a sample list.

Discussion prompt for the room: 'Suppose tomorrow we genuinely need a MEDIAN. With the BEFORE design, what do we do?' (Answer: add a MedianStrategy, register it with the factory, possibly extend the config.) 'With the AFTER design?' (Answer: import median from statistics. Or write a one-line median function. Two minutes of work.) The 'before' design was JUSTIFIED by anticipated need for additional strategies - but the 'after' design accommodates that need faster than the 'before' design does. Speculative generality is not just expensive to maintain. It is usually slower than the simple version even at the moment when its anticipated need finally materializes.

Final demonstration - the AI prompt that produces 'after' instead of 'before'. Tell the room: open your AI assistant. Type: 'write me a function in Python that computes the average of a list of numbers.' Note what it generates. Now type: 'write me the simplest possible function in Python that computes the average of a list of numbers - no classes, no patterns, use the standard library.' Same model, completely different output. The prompt decides which version of the assistant you are working with.

Wrap-Up - End of Course

REMEMBER

The skill we just demonstrated - looking at correct code and asking whether each piece earns its place - is the senior engineer's signature move. Most engineers add code. Senior engineers ask whether the code should exist at all. AI assistants will always add. The job of the engineer using them is to subtract. If you remember one habit from this course, let it be this: before merging any AI-generated code, ask, for each new class or layer, what does it protect me from. If you can't answer, delete it. The simpler version is almost always the right one.

Key takeaways from this hands-on:

- Correctness is not enough. Code can be correct and still smell. The 'before' file works perfectly and is still bad.

- The diagnostic questions - what does this protect me from, when has that happened, how many subtypes today - are the cheapest, fastest way to identify needless complexity in your own code.
- The standard library is your friend. statistics.mean replaced six classes. Most needless complexity dissolves when you reach for the built-ins.
- AI assistants generate the 'before' version by default. The 'after' version requires you to ASK for simplicity explicitly. Without the ask, you will pay for the ceremony forever.
- Deleting code is a senior skill. Practice it. It is the most valuable thing you can do for a codebase, and it is the least celebrated.

Course Complete - Source File Reference

The two files for this hands-on are average_before.py and average_after.py. With this lesson, you have completed The AI-Native Smells Detector. You now have working knowledge of all five smells - Rigidity, Fragility, Immobility, Viscosity, and Needless Complexity - and concrete refactoring patterns for each. If you are ready to go deeper, The AI-Native Engineer covers the practice of shipping production software at AI-native speed without inheriting these smells. The AI-Native Game Guru covers the same material for game developers. Thank you for taking the course.