The AI-Native Smells Detector, Lesson 1: Fragility

Hands-On: A Tiny Change That Breaks Three Things

Overview

This hands-on shows fragility live, in code. We will:

- Start with a small Cart class and three downstream modules that all work today.
- Make a one-line, finance-approved, 'obviously safe' change inside Cart - switching prices from float to Decimal to fix a rounding bug.
- Watch features in three unrelated modules break, even though we touched exactly zero of their lines.
- Apply encapsulation and a Money value object, then redo the SAME change and watch all three modules remain perfectly happy.

Before You Start

- Have Python 3.10+ installed. Verify with python3 --version.
- Open three files from the source directory: cart_before.py, cart_breaking.py, and cart_after.py.
- Have a terminal open in the same directory, ready to run each file.
- Have the lecture slides visible on a second screen - the visual of the 'tiny change' moment lands harder when the room sees the diff.

REMEMBER

Fragility is the smell of an unwritten contract. The Cart never agreed that prices would stay as floats forever. The downstream modules ASSUMED they would. When the assumption breaks, distant code breaks. The fix is always the same shape: make the contract explicit, then enforce it.

During Hands-On - Three Steps at a Glance

- **STEP 1** - Read and run the original Cart. Three modules use it. Everything works today. ~3 min
- **STEP 2** - Make the 'tiny safe change' - float to Decimal - and watch distant code blow up. This is the moment. ~4 min

- **STEP 3** - Refactor with encapsulation and a Money value object. Make the same internal change again. Nothing breaks. ~6 min

Step 1 - The Cart Today, Everything Working

WHAT TO LEARN

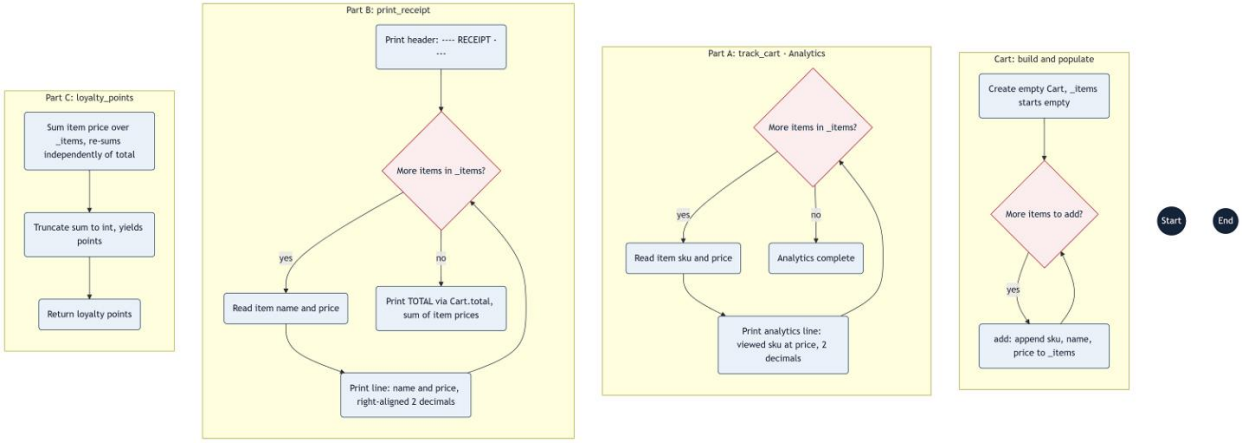
Fragility hides in the public surface that nobody decided was public. The Cart stores items in a list. The list is technically prefixed with an underscore, meaning 'please don't touch.' But Python doesn't enforce that, and the three downstream modules - analytics, receipt printing, loyalty - all reach in anyway. The internal data structure has accidentally become the public contract.

Open cart_before.py. Walk through the three modules that touch the cart's internals:

```
def track_cart(cart):
    for item in cart._items:                    # reaches in
        print(f"... at ${item['price']:.2f}")    # assumes float

def print_receipt(cart):
    for item in cart._items:                    # reaches in
        print(f"{item['name']} ${item['price']:>6.2f}")

def loyalty_points(cart):
    return int(sum(item['price'] for item in cart._items))   # reaches in
```



Flowchart for cart_before.py

Run it:

```
python3 cart_before.py
```

WHAT YOU SHOULD GET

Three modules running happily. Analytics tracks. The receipt prints. Loyalty points come out to twelve. The output is uninteresting. The point to make to the room is the SHAPE of the dependency: every downstream module knows what shape Cart's internals have. The Cart has accidentally promised that those internals will never change. That promise was never written down, and we'll see what happens when finance asks us to change one tiny detail.

Step 2 - The 'Tiny' Change That Detonates

WHAT TO LEARN

Floats and money are a known-bad combination. Finance reports a one-cent rounding error in production. The fix is OBVIOUS: store prices as Decimal. We make that change in Cart, and ONLY in Cart. We don't touch analytics, the receipt printer, or loyalty. There is no reason to. The change is internal to Cart. Everything should be fine. (It will not be fine.)

Open cart_breaking.py. The 'fix' is a one-line edit inside the add method:

```
def add(self, sku, name, price):
    self._items.append({"sku": sku, "name": name,
                        "price": Decimal(str(price))})  # the small change
```

Run the breaking version:

```
python3 cart_breaking.py
```

Most of the output looks fine at first. Analytics tracks. The receipt prints. Loyalty points come out to twelve. The room may think the demo failed - everything works. Then read the last lines:

```
BOOM: distant code broke ->
    unsupported operand type(s) for *: 'decimal.Decimal' and 'float'
That is fragility. One small change, cascading damage.
```

WHAT YOU SHOULD GET

A TypeError in apply_tax - a function that has not been touched, in a module nobody thought to look at. It is downstream of the Cart by two hops, and it assumed all monetary values were floats. Pause here for the room. Ask them: in your codebase, how many functions are like apply_tax? How many downstream consumers depend on a type that was never written down as part of the contract? This is the moment that justifies the rest of the lesson.

Step 3 - Encapsulation + Value Objects

WHAT TO LEARN

There are two moves and they always go together. First, encapsulate truly - double-underscore the items list so callers cannot reach it. Defensive copy on the public items method. Second, introduce a Money value object that owns the storage choice. Downstream code talks to Money - it never sees Decimal, never sees float, never sees the choice at all. Storage can now change freely.

Open cart_after.py. The interesting new types are at the top:

```
@dataclass(frozen=True)
class Money:
    amount: Decimal              # the storage choice lives HERE
    @classmethod
    def of(cls, value): return cls(Decimal(str(value)))
    def __add__(self, other):   return Money(self.amount + other.amount)
    def format(self):           return f"${self.amount:.2f}"
    def as_int_dollars(self):   return int(self.amount)

@dataclass(frozen=True)
class LineItem:
    sku: str
    name: str
    price: Money
```

And the Cart now has a real wall around it:

```
class Cart:
    def __init__(self):
        self.__items: list[LineItem] = []   # truly private

    def items(self) -> list[LineItem]:
        return list(self.__items)            # defensive copy

    def total(self) -> Money: ...
```

Downstream modules now talk to the contract, not the guts:

```
def print_receipt(cart):
    for item in cart.items():               # public method, not _items
        print(f"{item.name} {item.price.format()}")  # Money formats itself

def loyalty_points(cart):
    return cart.total().as_int_dollars()    # Money owns the conversion
```

Run it:

```
python3 cart_after.py
```

Now show the room the magic. Open Money in the editor. Change the storage from Decimal to integer cents (multiply by 100 in 'of', divide in 'format'). Run cart_after.py again. Everything still works. NO downstream code changed. The contract held. The fragility is gone.

WHAT YOU SHOULD GET

Identical output regardless of how Money stores its value internally. Show this twice - once with Decimal, once with int cents. The downstream modules are immune. That immunity is the whole prize. Tell the room: every value object you write earns this same immunity for that domain concept. Money. Email. UserId. Date. The number of fragility bugs in your system drops in proportion to how many of these you have.

Wrap-Up - The Pattern, The Assignment Handoff

REMEMBER

Fragility is the smell of an unwritten contract. The defense is to write the contract down in code that the compiler can check: truly private internals, value objects for domain concepts, and contract tests that pin down public behavior instead of internal structure. Every value object you write is a small bet against a future production fire.

Key takeaways:

- Fragility shows up as breakage in code you did not touch. The bug count is uncorrelated with the change.
- The three classic sources are leaking internals, implicit contracts, and hidden coupling. They overlap and reinforce each other.
- Encapsulate ruthlessly, use value objects for domain concepts, write contract tests, and put static types at the boundaries. Those four moves kill almost all fragility.
- Rigidity costs time. Fragility costs trust. Trust is harder to rebuild than throughput - so removing fragility is a long-term investment in the team's relationship with the rest of the business.
- AI coding assistants are particularly weak at implicit contracts because they can't see them. Making contracts explicit makes the assistant safer to use - and lets it write contract tests for you in seconds.

Source File Reference

The three files for this demo are cart_before.py, cart_breaking.py, and cart_after.py. Each file is under a hundred lines of standard Python with no external dependencies beyond the Decimal module in the standard library. Open all three before class, run them once in order, and confirm the BOOM actually prints when running cart_breaking.py. The moment of failure is the moment the lesson lands.