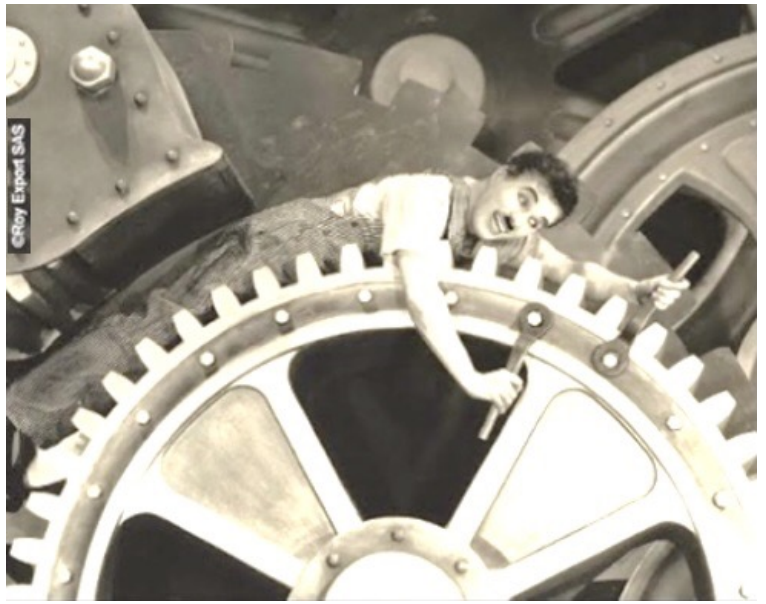
Simplifying Software Design:

The Genius of Bureaucracies, or how "Not-My-Job" sharpens your design





Alistair Cockburn

©Alistair Cockburn 2026

Past and the future meet in ‘Not my job’.

(It’s new, It’s improved, It’s old-fashioned! --Tom Waits)

1. Anchored in Ward Cunningham / Kent Beck *CRC cards*, and Rebecca Wirfs-Brock *Responsibility-Driven Design*. Compatible with Eric Evans *Domain-Driven Design*.

2. The added wrinkle:
   “Not my job” and “No need to know.”

3. LLMs love it, a good sign for the future.

©Alistair Cockburn 2026



Bureaucracy gives software design a quality gradient.

(Yes, we hate it, that's why we're so good at it.)

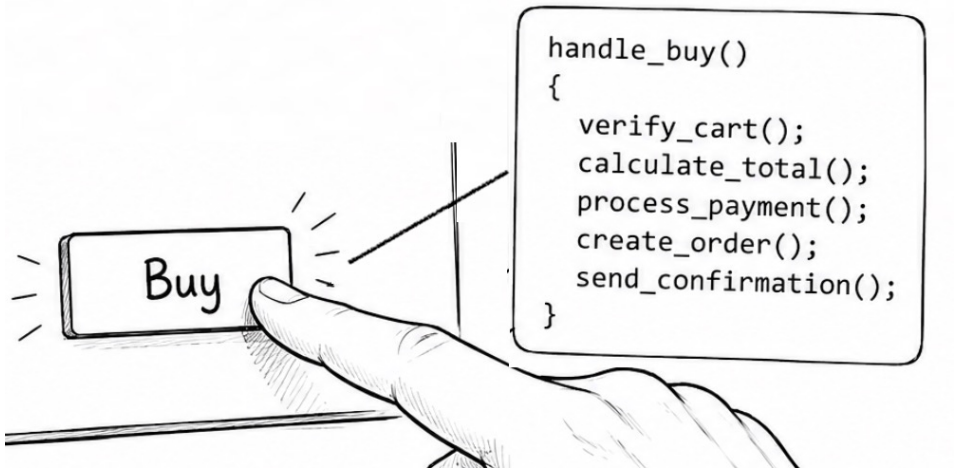
Key phrases :

1. Design in software is deciding where to put each line of code.
2. Discussing 'design quality' is discussing the futures a design supports.
3. Treat an object as a petty bureaucrat with files and a phone.
4. Responsibilities tell the short form of the design.
5. ***Not-my-job*** and ***No-need-to-know*** help you sharpen your thoughts.

©Alistair Cockburn 2026



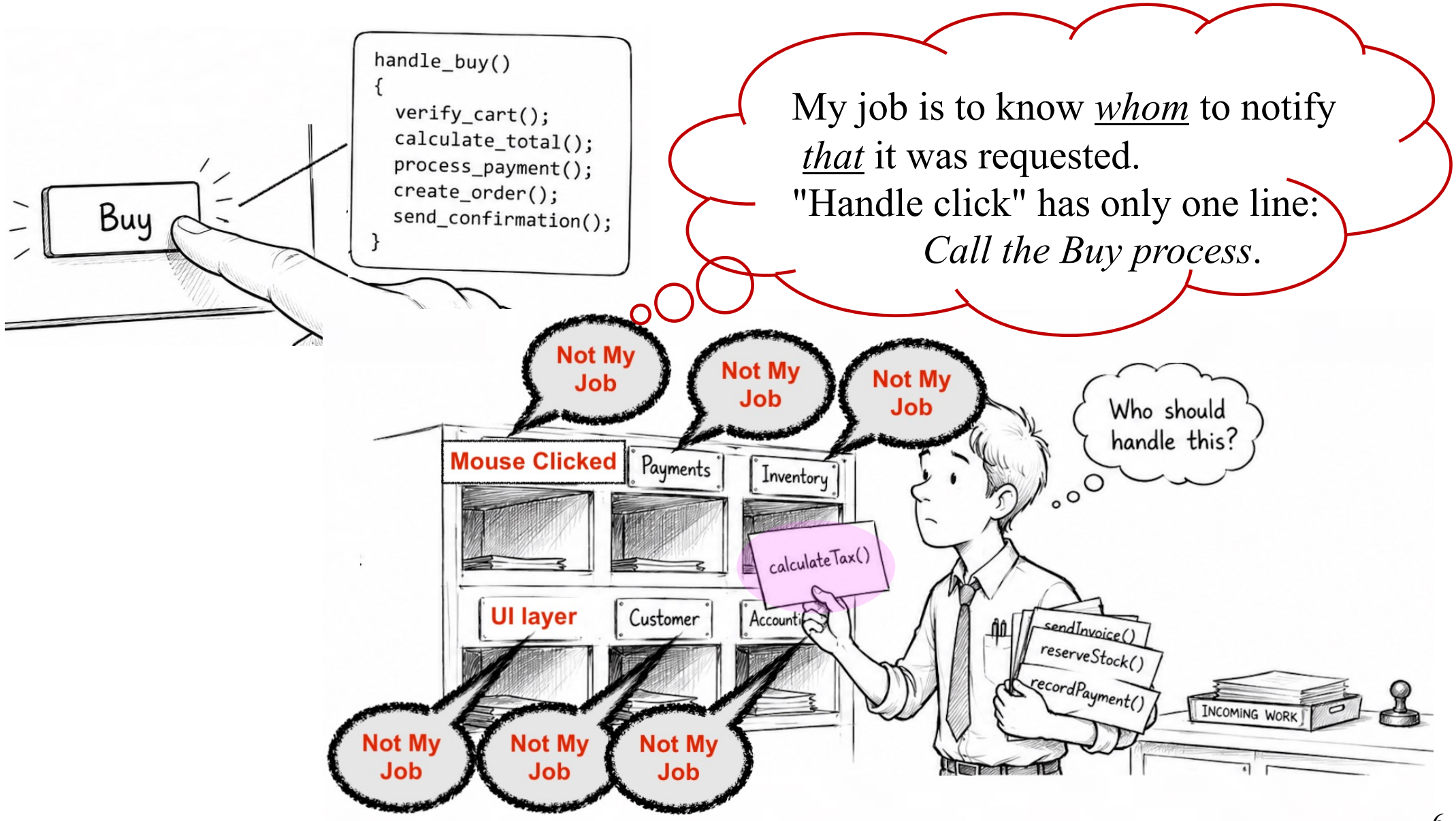
Design in software is deciding where to put each line of code.





©Alistair Cockburn 2026

‘Not my job’ provides a *gradient* for choosing where to put it.



©Alistair Cockburn 2026

Discussing ‘design quality’ is discussing the futures that modularization naturally supports.

On the Criteria To Be Used in Decomposing Systems into Modules 1972

D.L. Parnas

Carnegie-Mellon University

**Mostly. There are of course other design properties to evaluate: performance, security, etc.

* Avoid ‘simplicity.’ There are too many variants of simplicity ☺

The benefits expected of modular programming are:

(1) managerial_development time should be shortened because separate groups would work on each module with little need for communication:

(2) product flexibility_it should be possible to make drastic changes to one module without a need to change others;

(3) comprehensibility_it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

©Alistair Cockburn 2026



The futures-search is about changes in decisions.
(Reading into a crystal ball)

"We propose that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others

(Replace 'hide' with 'localize')

On the Criteria To Be Used in Decomposing Systems into Modules
D.L. Parnas
Carnegie-Mellon University

The Single Responsibility Principle (general statement):
"Gather together the things that change for the same reasons. Separate those things that change for different reasons."

(Yes, but how?)

©Alistair Cockburn 2026



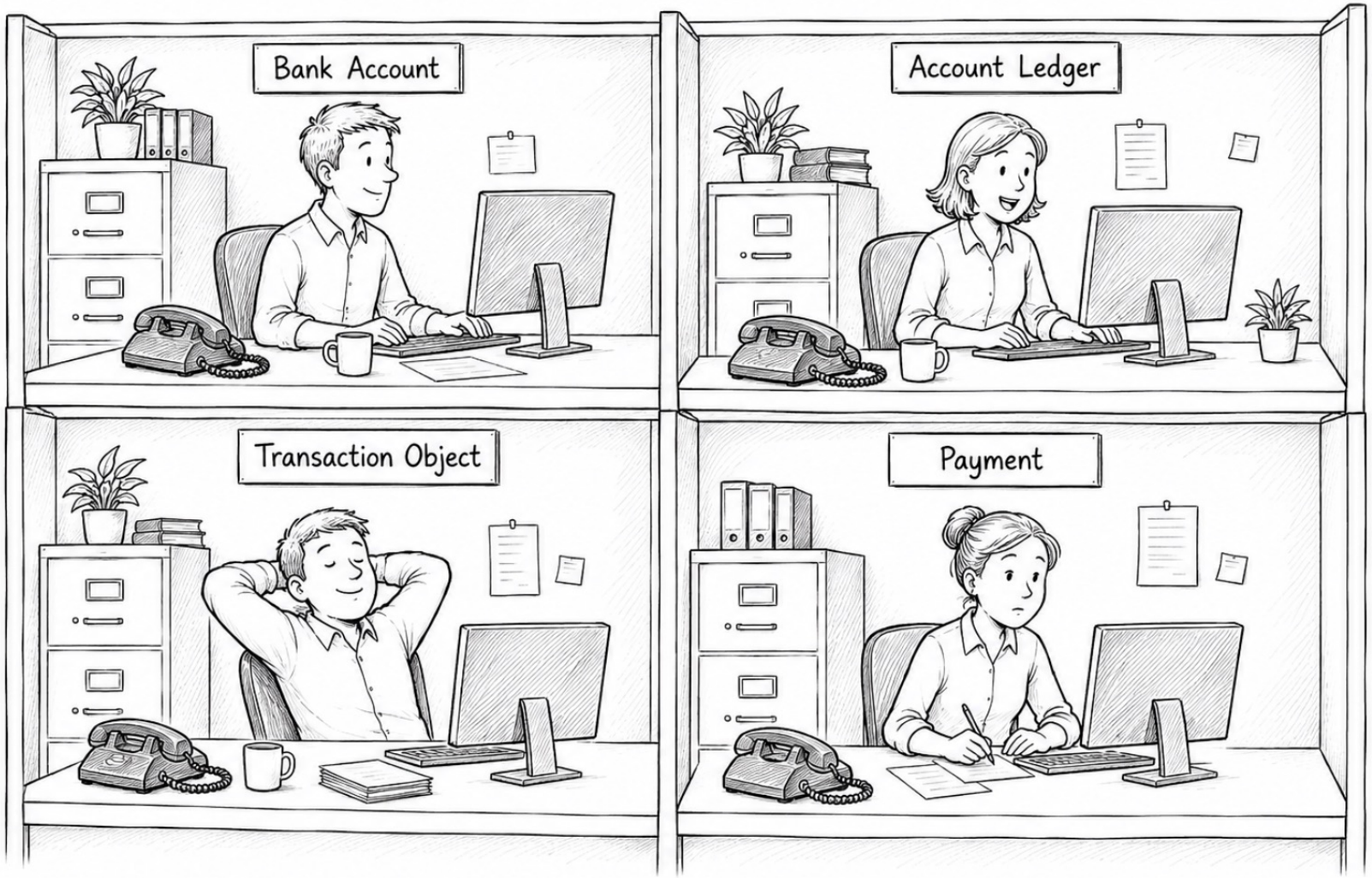
‘Not my job’ sharpens our sense of future changes.



©Alistair Cockburn 2026

Bureaucracy design:

Each *object* is a person with filing cabinet & phone.



©Alistair Cockburn 2026

Reviewing object-design:

Definition 1: An object is a "thing", with all the state and services that come with it.

What is a telephone?

... a "thing" that cost money, has a number, and which we use to make and receive calls.

In your program, what will "a Telephone" be?

... an "object" with selected properties and services.

- may have a cost (if you are a retailer)
- may dial for you
- likely to "know" its phone number

Metaphor 1: Think of an object as a "thing" with a personal secretary to handle your commands.

"Dial this number", you command...

(the secretary takes the phone off hook, dials...)

the object takes the phone off hook, dials...

"How much did you cost?", you ask...

(the secretary looks into the phone's records...)

the object looks into its records...

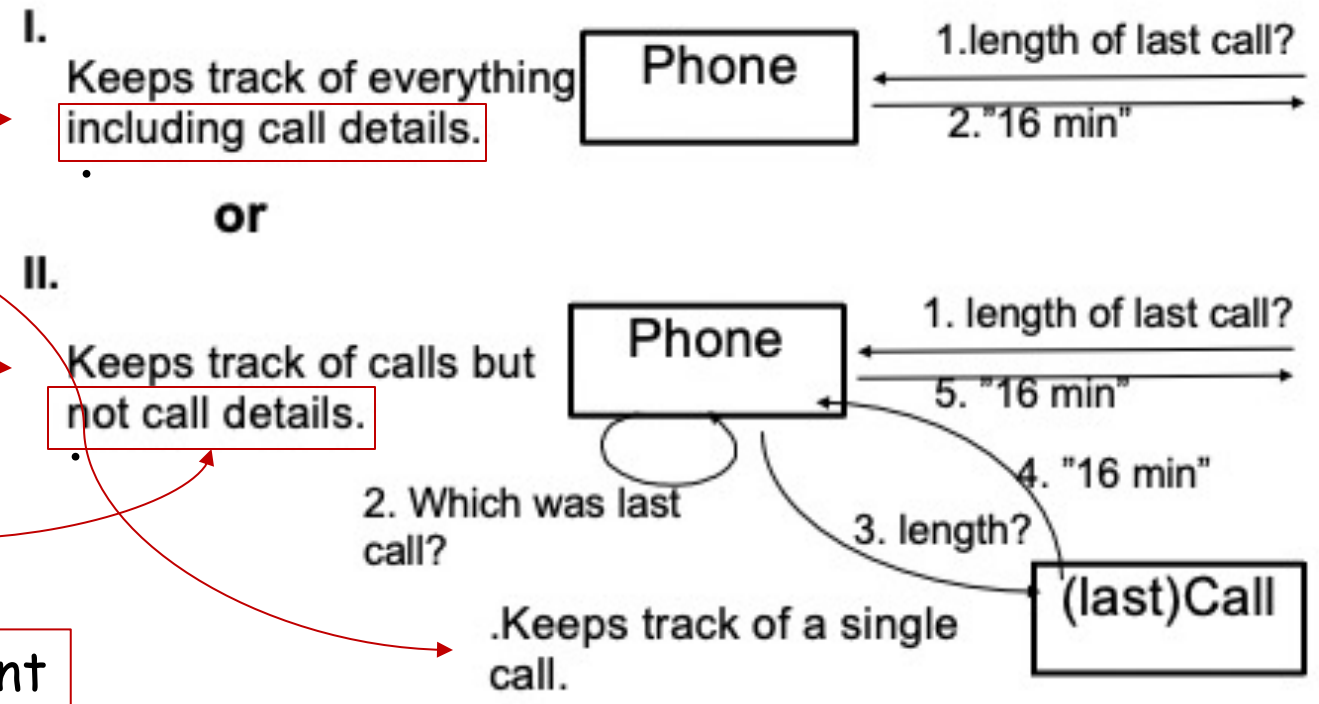
©Alistair Cockburn 2026



Behavior allocation is bureaucracy design. Responsibilities tell us what to expect.

"Object design" is the design of how much knowledge and control goes into each object.

These are 'responsibility statements'

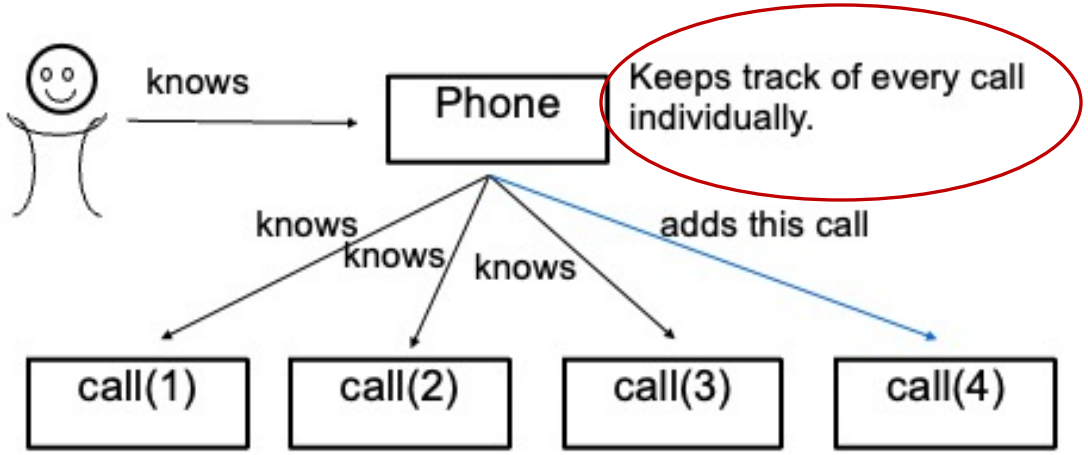


A clear 'Not my job' statement

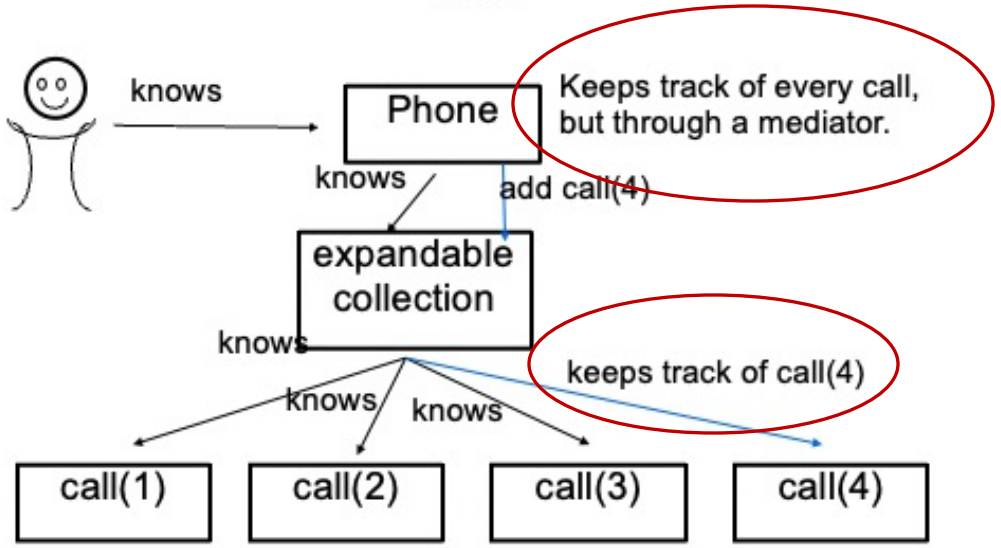
©Alistair Cockburn 2026

We can add or remove intermediaries.

A large part of design is controlling which object knows about which other objects.



Sometimes we add an object to control how much an object knows about others, or with how much work.



Keep your eye on the
'responsibility statements'

©Alistair Cockburn 2026

A responsibility can be stated in one or two short phrases.

Responsibilities of an Order:

"Responsible for its line items and its value.."

Sub-responsibilities:

- Knows how to add and remove line items.
- Knows the tax computation.
- Knows its final value,

a Transfer Transaction
Responsibility: Handle transfer from one account to another
Active: Ensures completion of the transfer → Accounts
Mediates: Amount From-account To-account Status

'Single' responsibility is tricky:

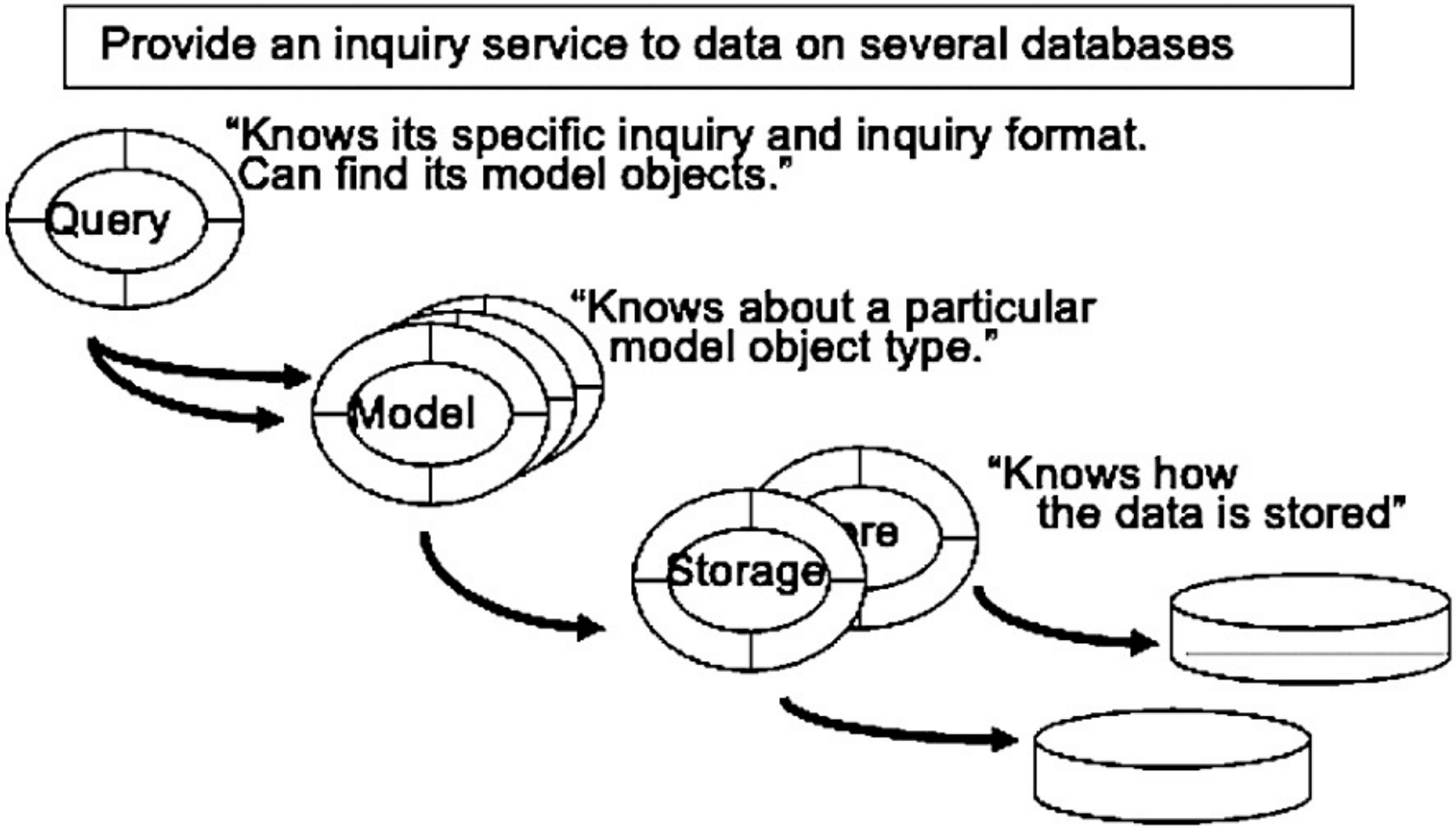
"a class, module, or function should have only one responsibility and exactly one reason to change."

what exactly is a 'single' responsibility?

©Alistair Cockburn 2026

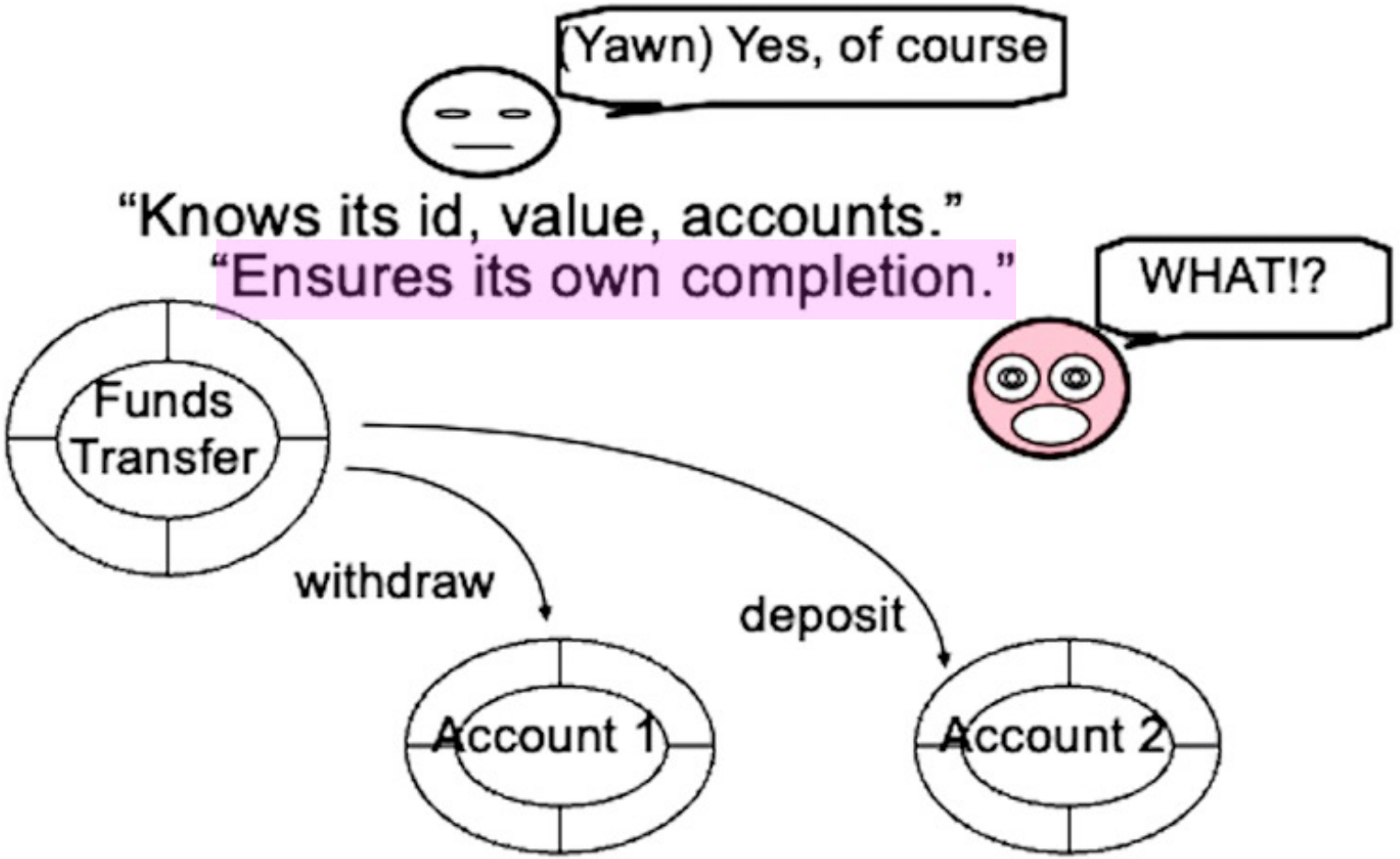


Responsibilities are the shortest description of a design.



©Alistair Cockburn 2026

The active responsibilities shape the system architecture.



©Alistair Cockburn 2026

Put 'not my job' line on CRC cards to document designs.

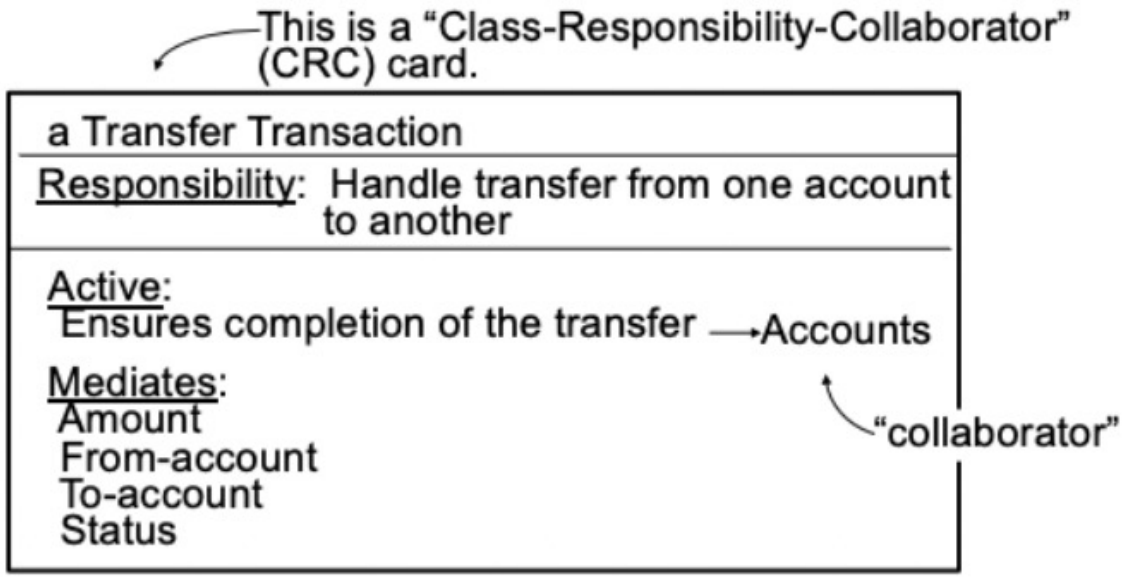
CRC Cards

Card: Competitor

Job: Hold the identity and mutable state of one living combatant.

Responsibility	Collaborators
Know name, kind, stats, hp, position	—
Compute stat total (sum of current stats)	—
Apply advance/retreat (clamped to ±100)	—
Apply damage (decrement hp)	—
Rearm: replace stats and recompute hp	—
Serialise to dict for queries	—

Not my job: deciding whether to remove itself from the game, narration, rules of engagement.



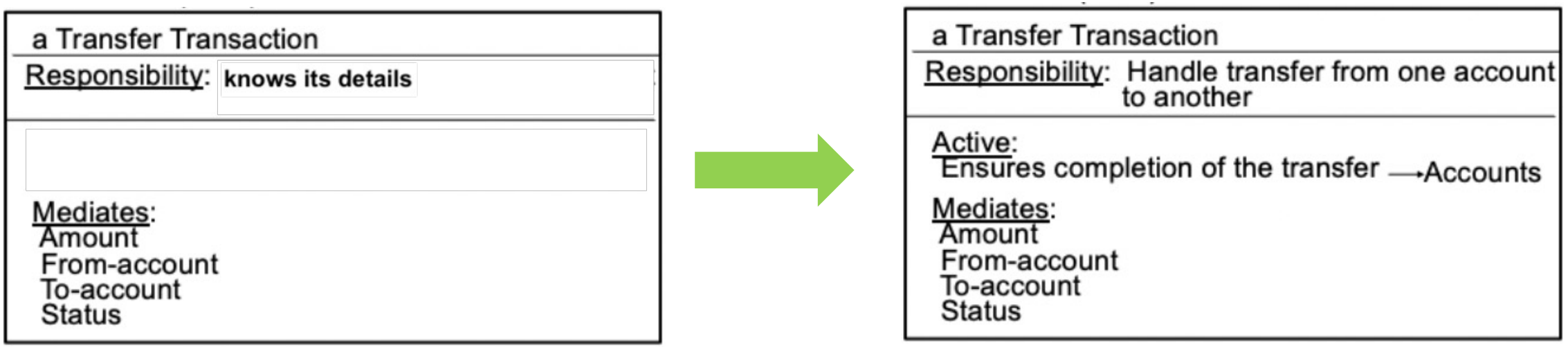
©Alistair Cockburn 2026

"Responsibility-alignment" test reveals design drift.

Naming - coding details - design intentions all drift over time.

How do we detect if it's gone off balance?

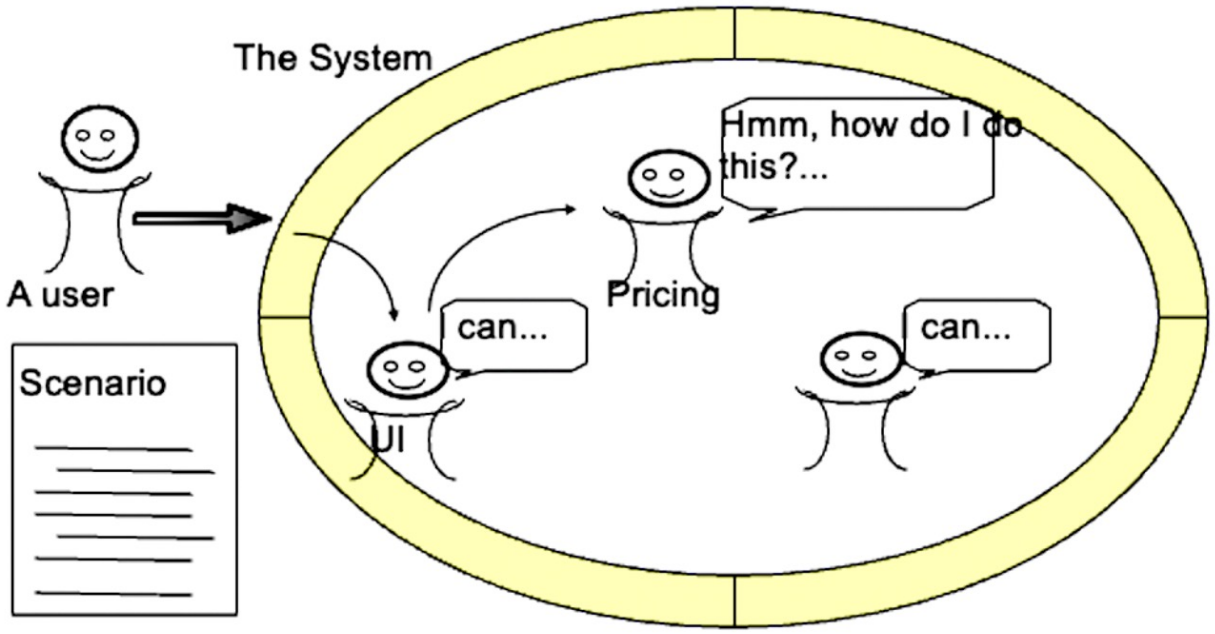
- Check the alignment of Name, Responsibility, Signatures (& Data).

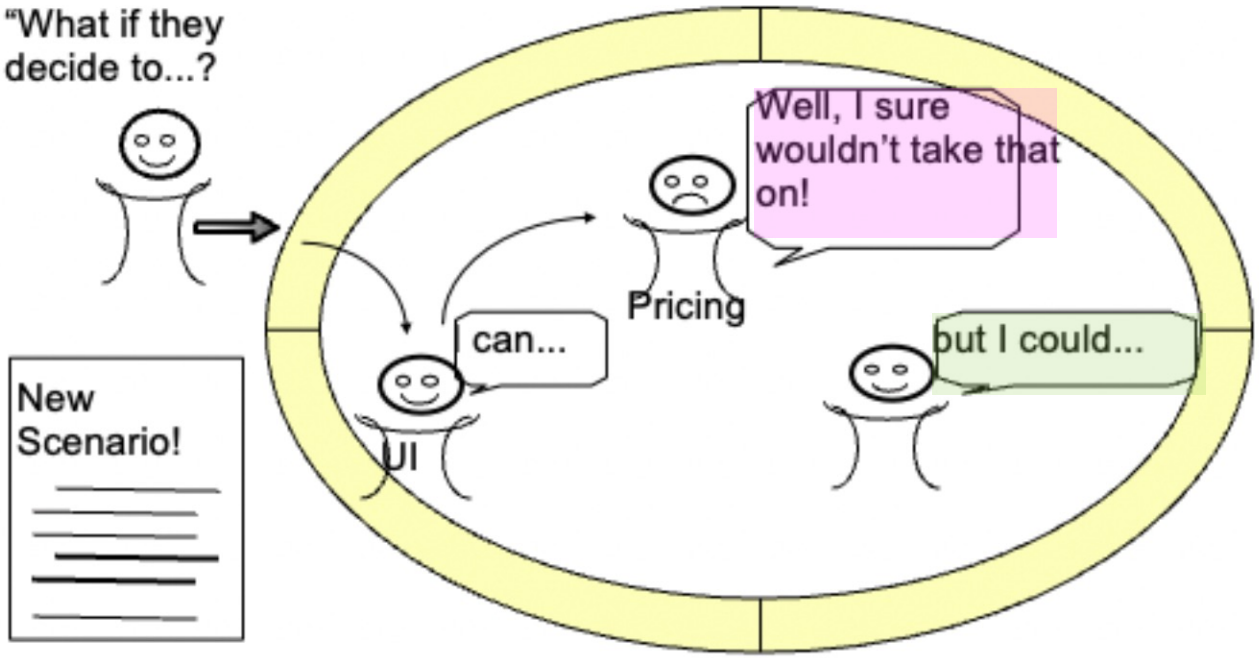


©Alistair Cockburn 2026

Role-play changes to find the *seams* and *whose job*.

Your intuition will guide you
'cuz we *feel* 'not my job'





©Alistair Cockburn 2026

Inheritance: Just like 2 objects, the generic & the specific

(Like two people sharing the same office)

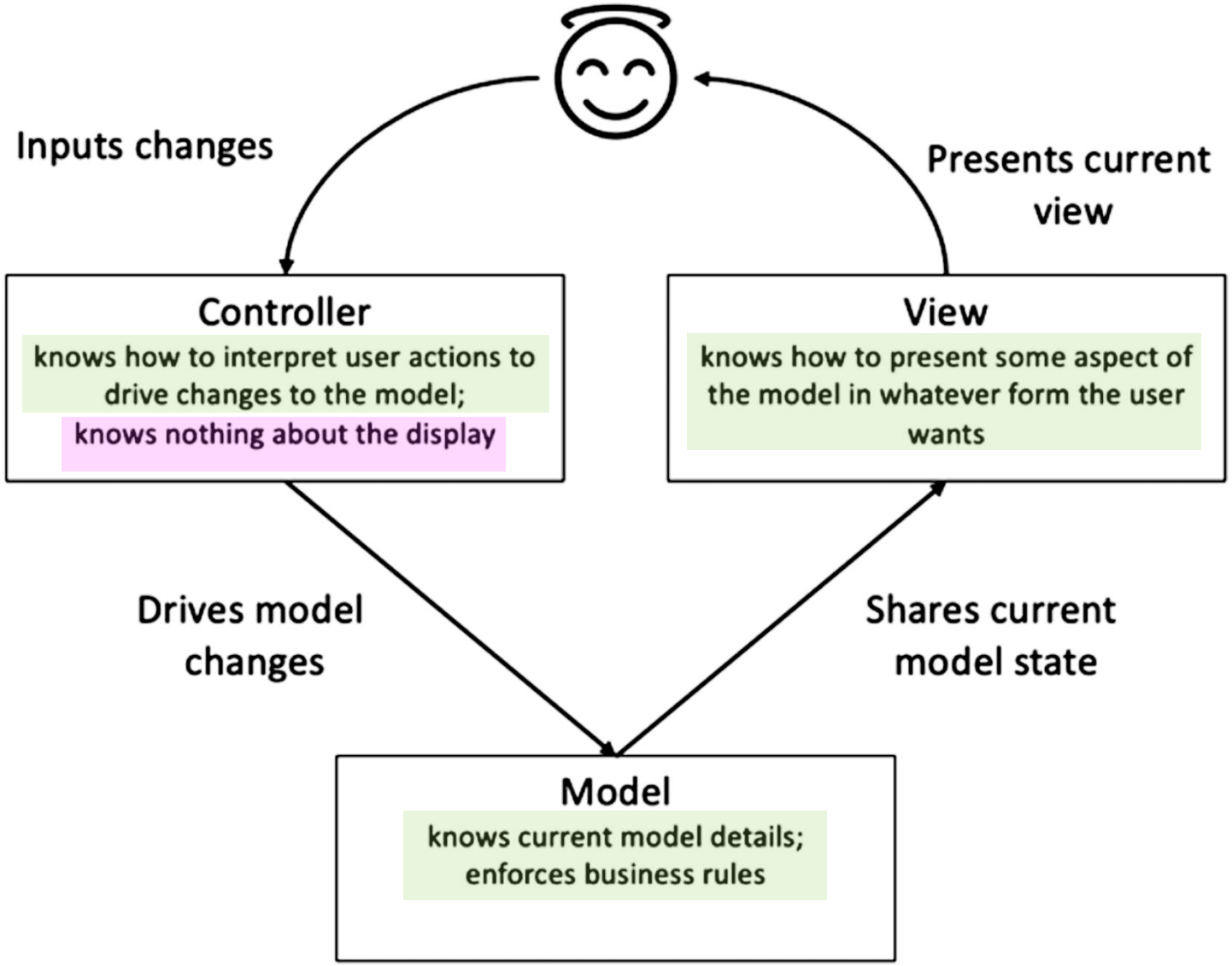
1. Upper object (generic): “Generically, it works like this…”
   Not my job to deal with the specifics; talk to my other half…

2. Lower object (specific):
   Not my job to handle all the generic back-and-forth
   just tell me when it’s my turn to do add my specific touch.

©Alistair Cockburn 2026

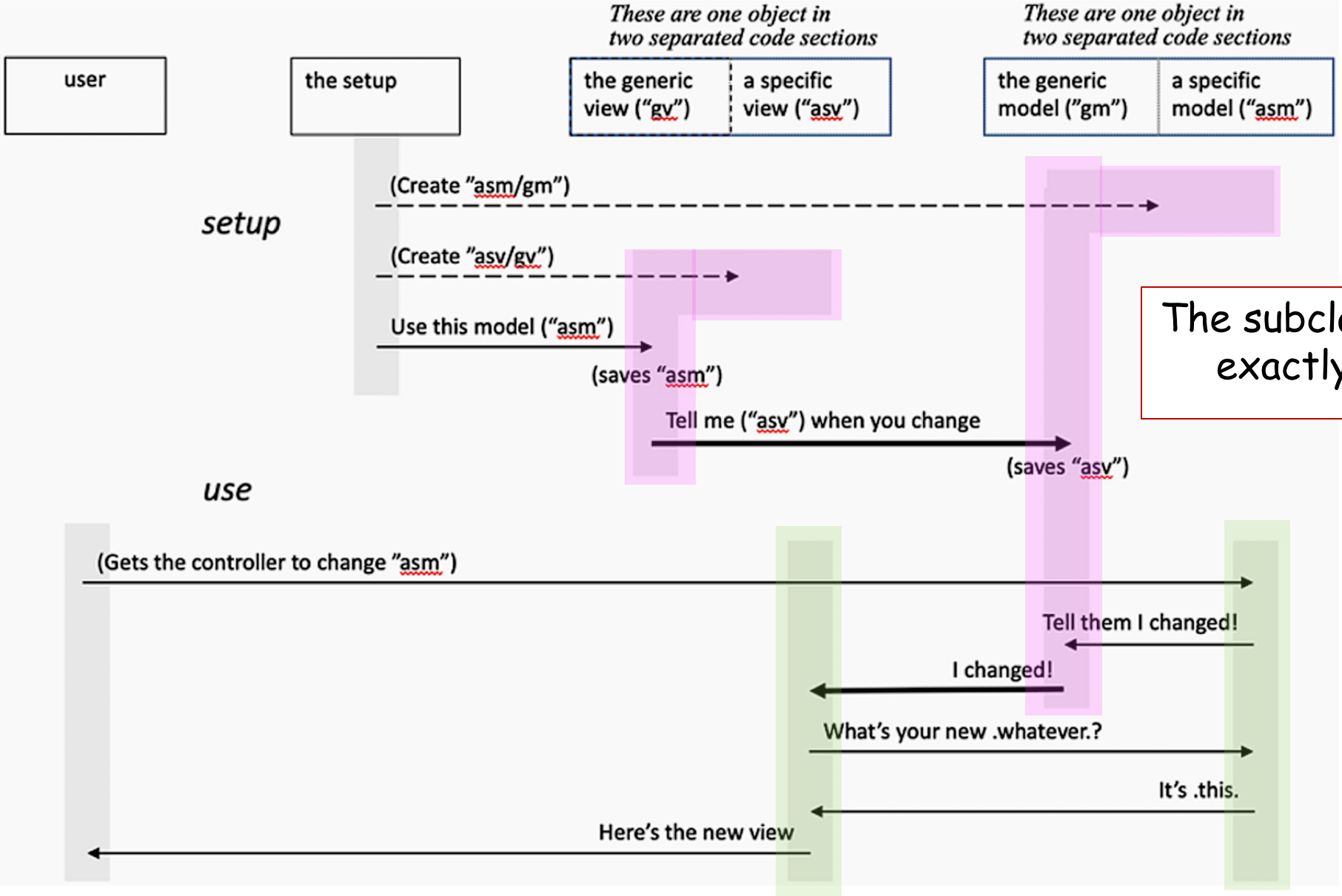


Example 1: Model-View-Controller



©Alistair Cockburn 2026

Example 1: Model-View-Controller in motion



The subclass has no *need-to-know* exactly how all that is done!

©Alistair Cockburn 2026

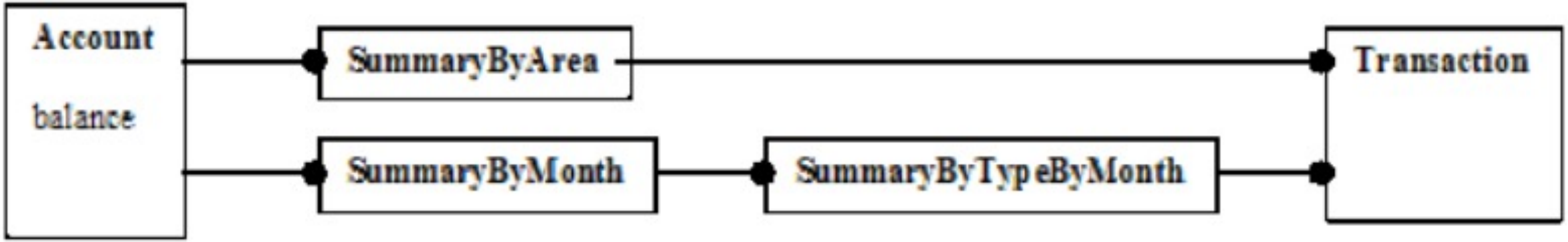
Example 2: The Squeegee technique

Situation: Considering possible move from SQL to OO DB:

Problem: Programmers making objects that just look like DB tables.
(DDD practitioners take note!)

Goal: Keep DB tables out of the discussion.

Method: Secure growing sections of the design space,
'squeegee' the remaining decisions into a localized area.



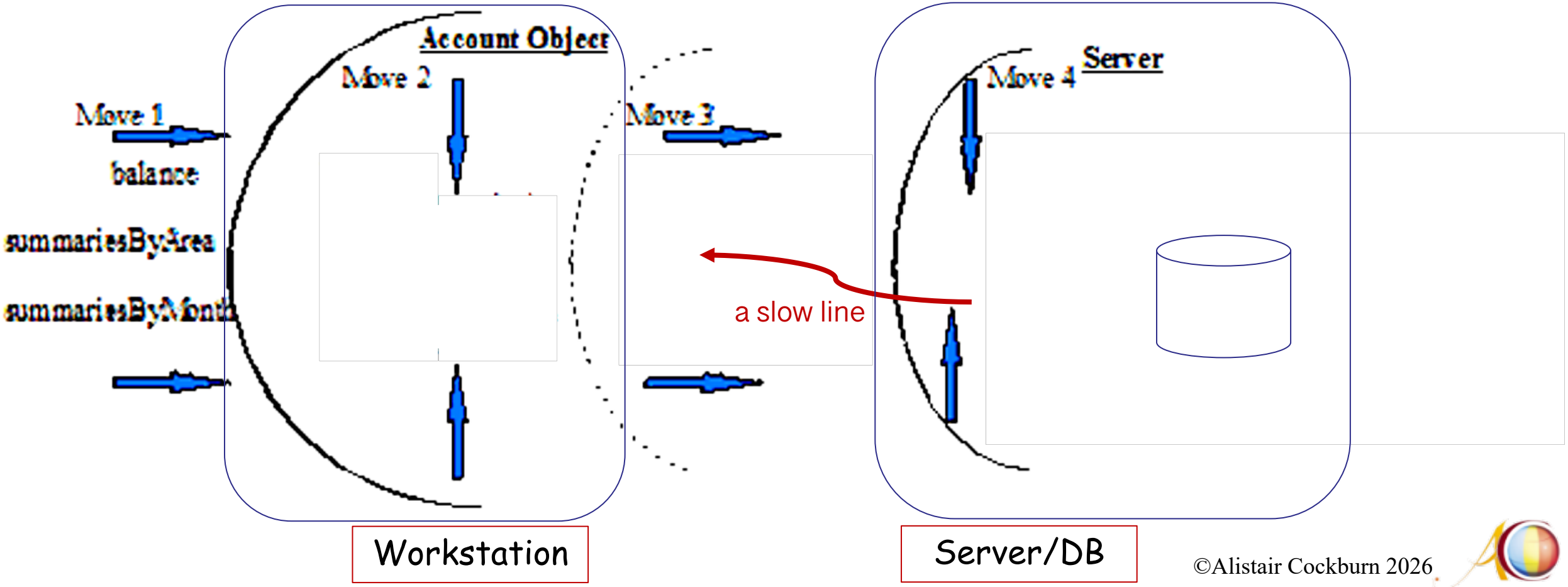
©Alistair Cockburn 2026

Example 2: The Squeegee technique (1)

Stage 1: Design a good application API to the *Account* domain object :

balance(), summariesByArea(), summariesByMonth()

Account's job is to answer those questions.



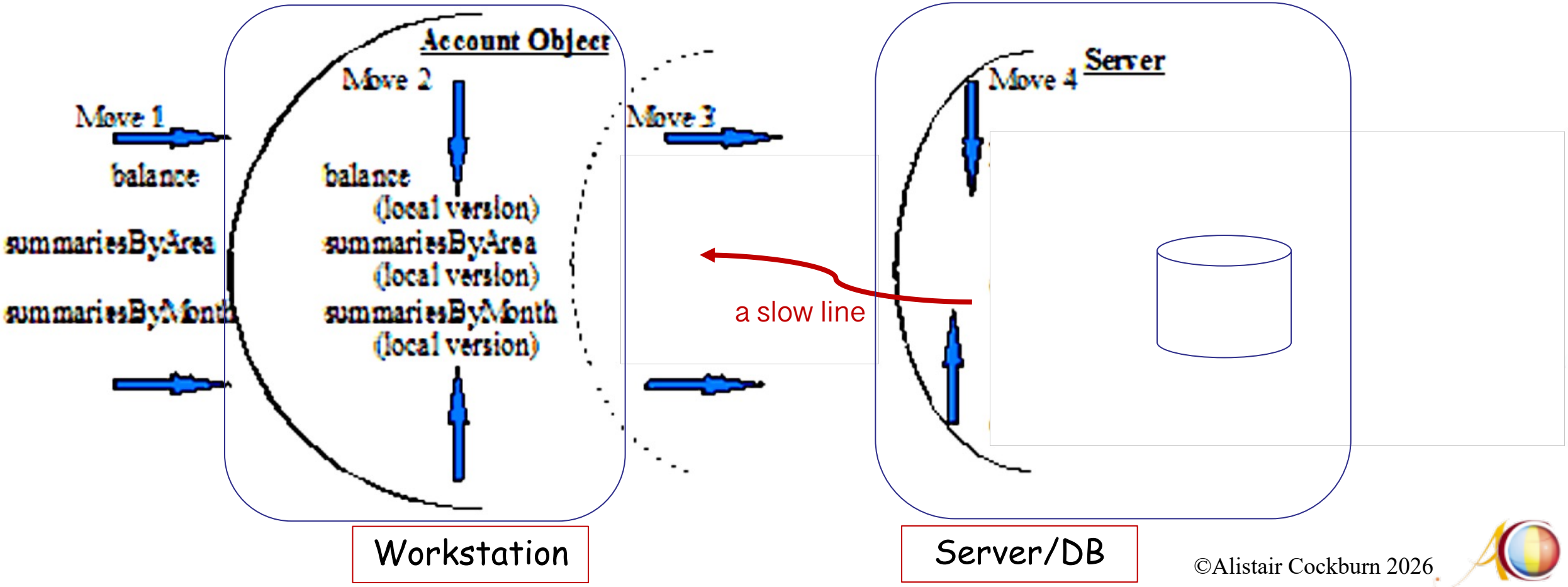
©Alistair Cockburn 2026

Example 2: The Squeegee technique (2)

Stage 2: Design as though local, no DB. Cache, don't compute summaries:

Account's job to keep cached summaries objects.

Not the DB team's job to care how these are cached.



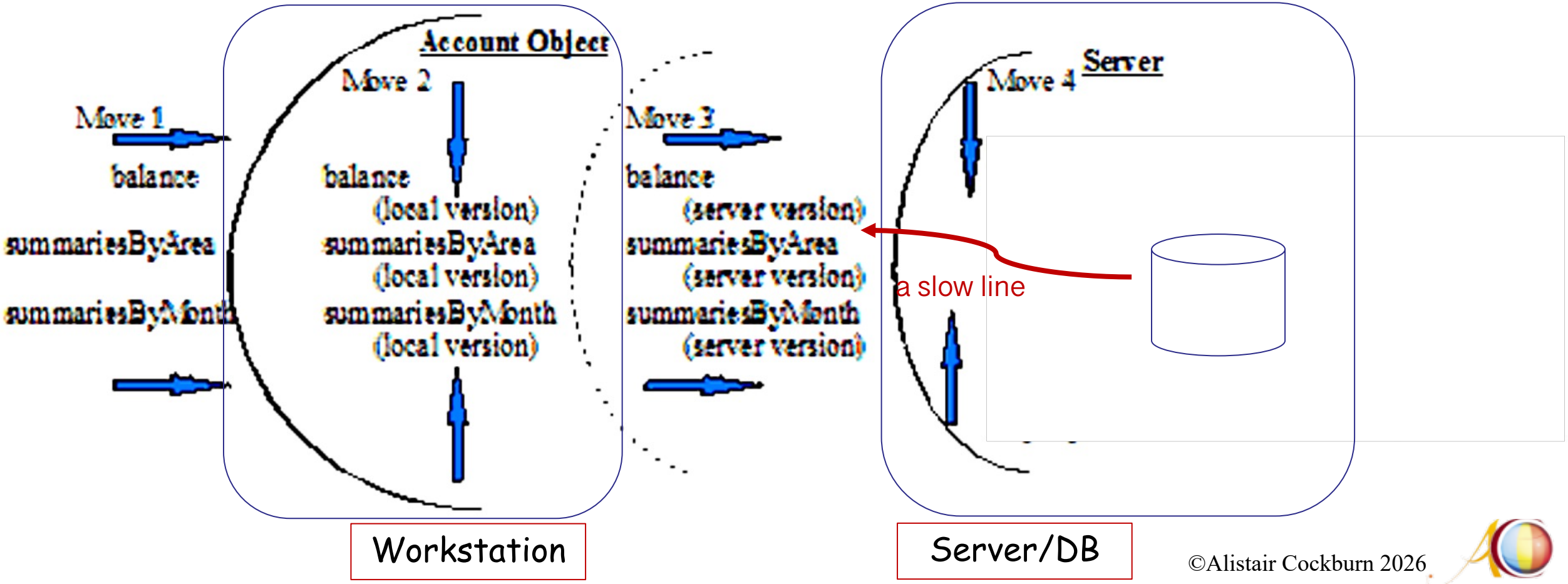
©Alistair Cockburn 2026

Example 2: The Squeegee technique (3)

Stage 3: Design the network: Send all transactions or summaries?

Slow lines => Server's job to sum the transactions.

balance(), summariesByArea() are server requests.

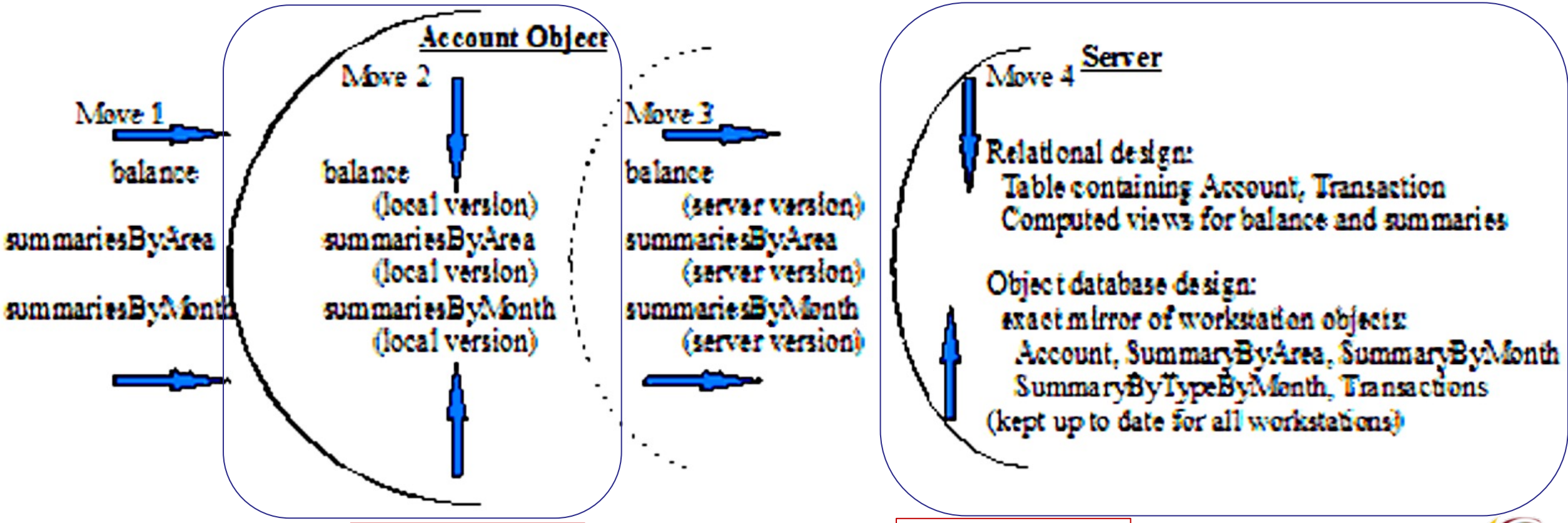


©Alistair Cockburn 2026

Example 2: The Squeegee technique (4)

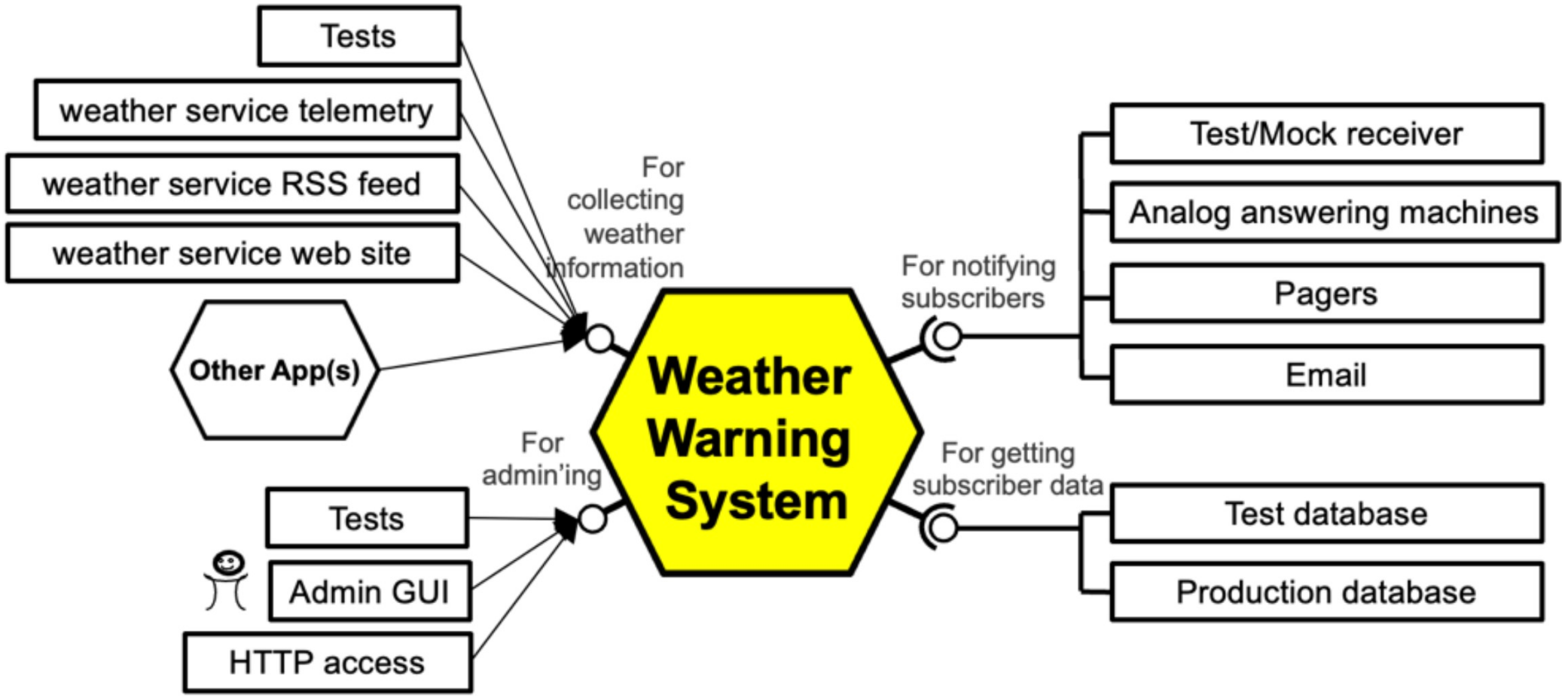
Stage 4: DB designers can move to OO DB or not!

Not the app team's job to know how these are produced.



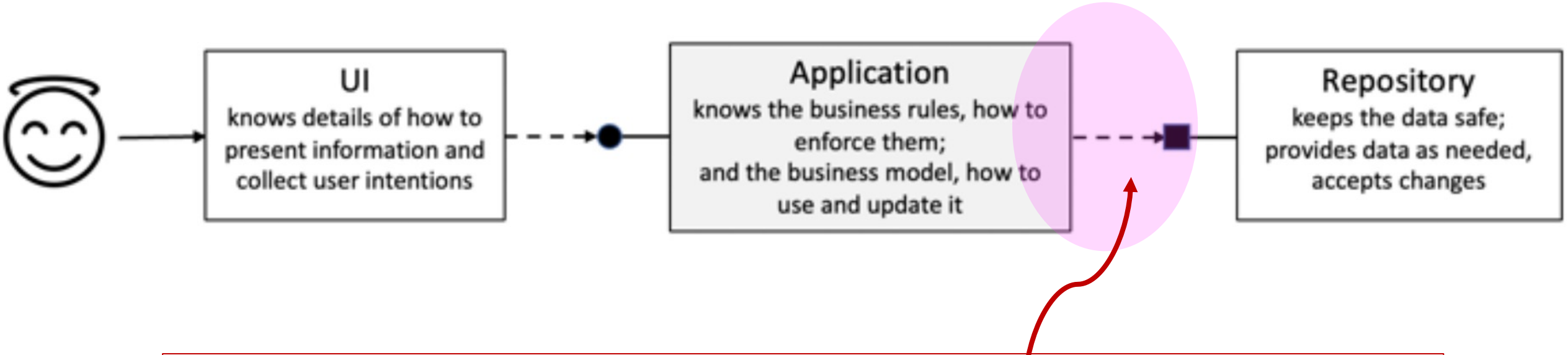
©Alistair Cockburn 2026

Example 3: Hexagonal / Ports & Adapters Architecture



©Alistair Cockburn 2026

Example 3: Hexagonal / Ports & Adapters Architecture (1)



The standard "3-layer"architecture violation:

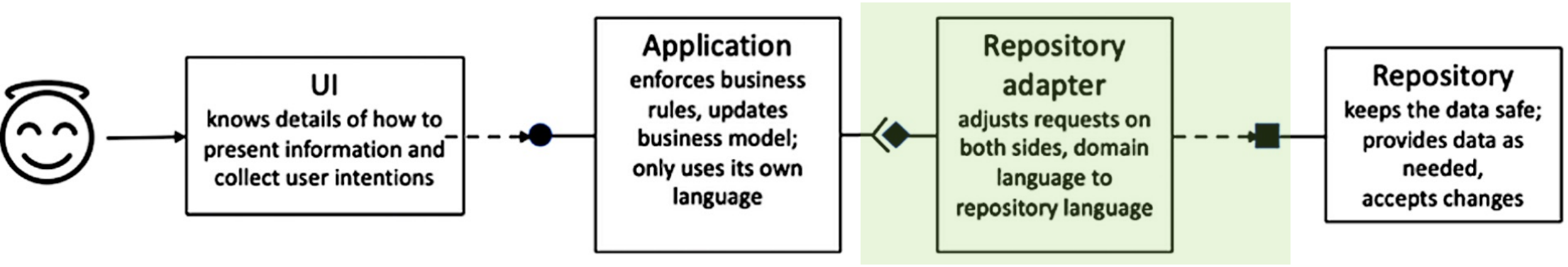
The app knows the location and type of the repository.

Change anything about the repository and the app has to change.

(booooo)

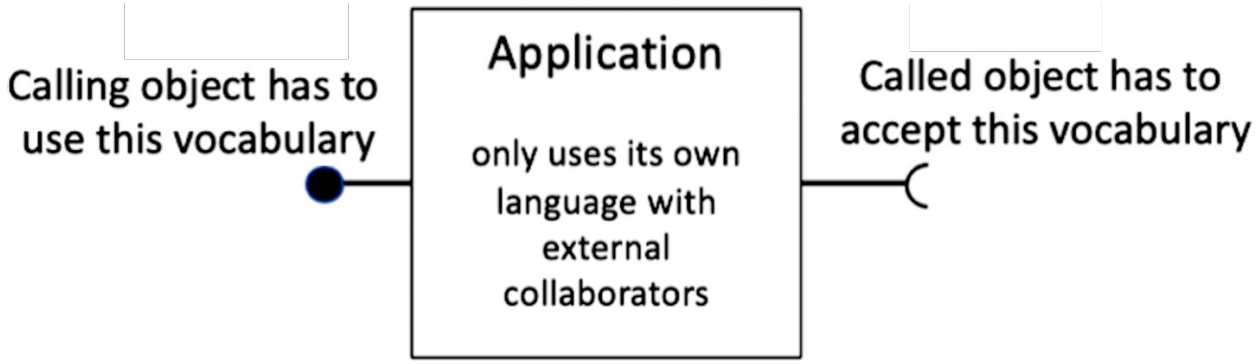
©Alistair Cockburn 2026

Example 3: Hexagonal / Ports & Adapters Architecture (2)



The app has no "need to know" anything about the repo. Zero.
It sends out calls in its preferred domain language.

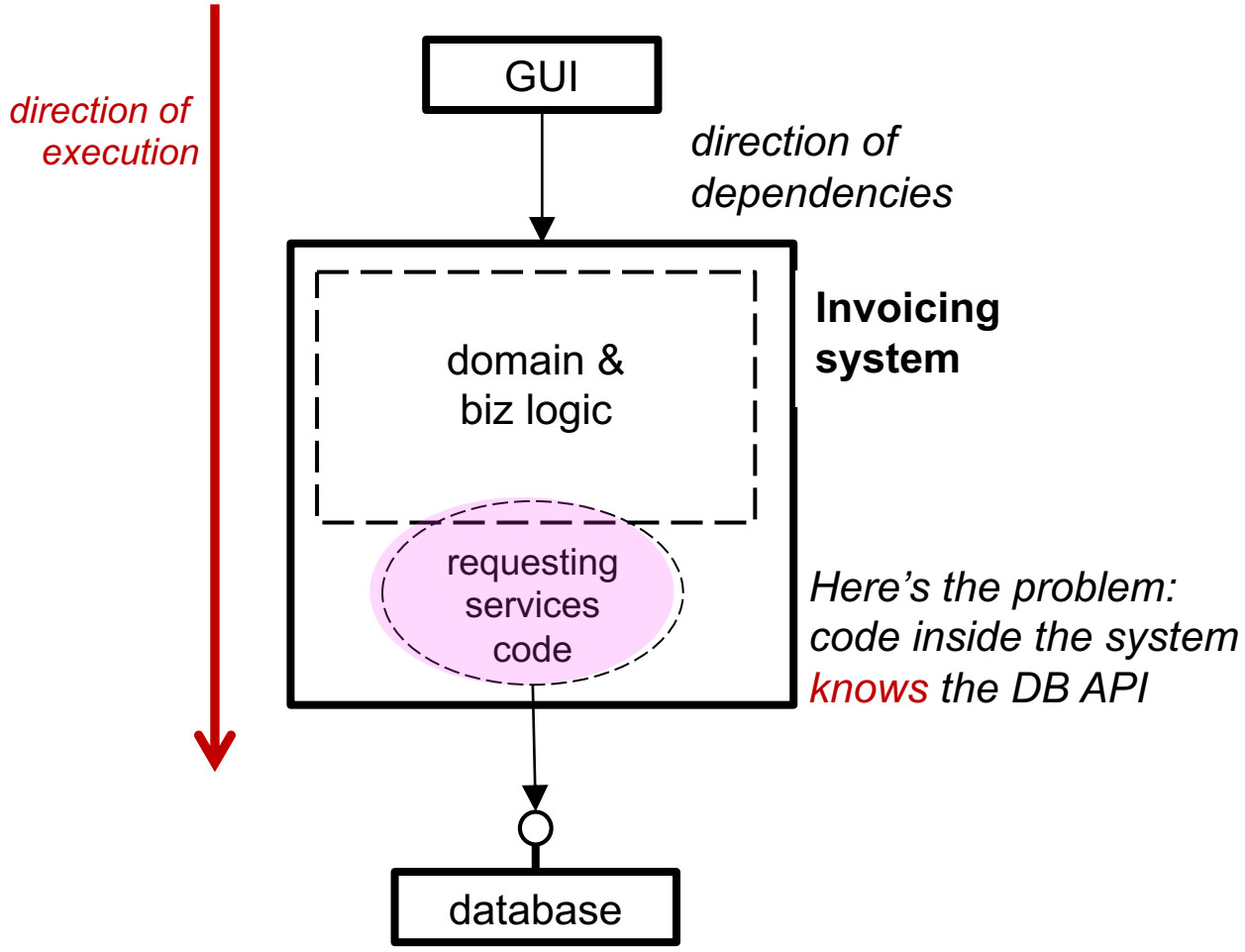
An adapter knows the interfaces of both app & repository.



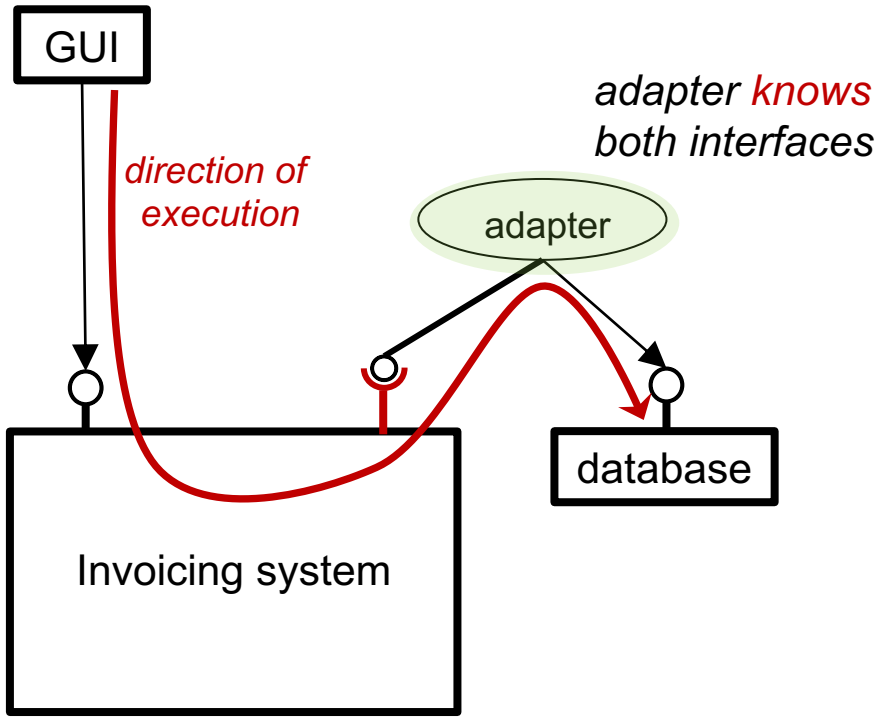
©Alistair Cockburn 2026

Example 3: Hexagonal / Ports & Adapters Architecture (3)

Standard 3-layer dependencies with difficulties



Ports & Adapters with all dependencies downward



©Alistair Cockburn 2026

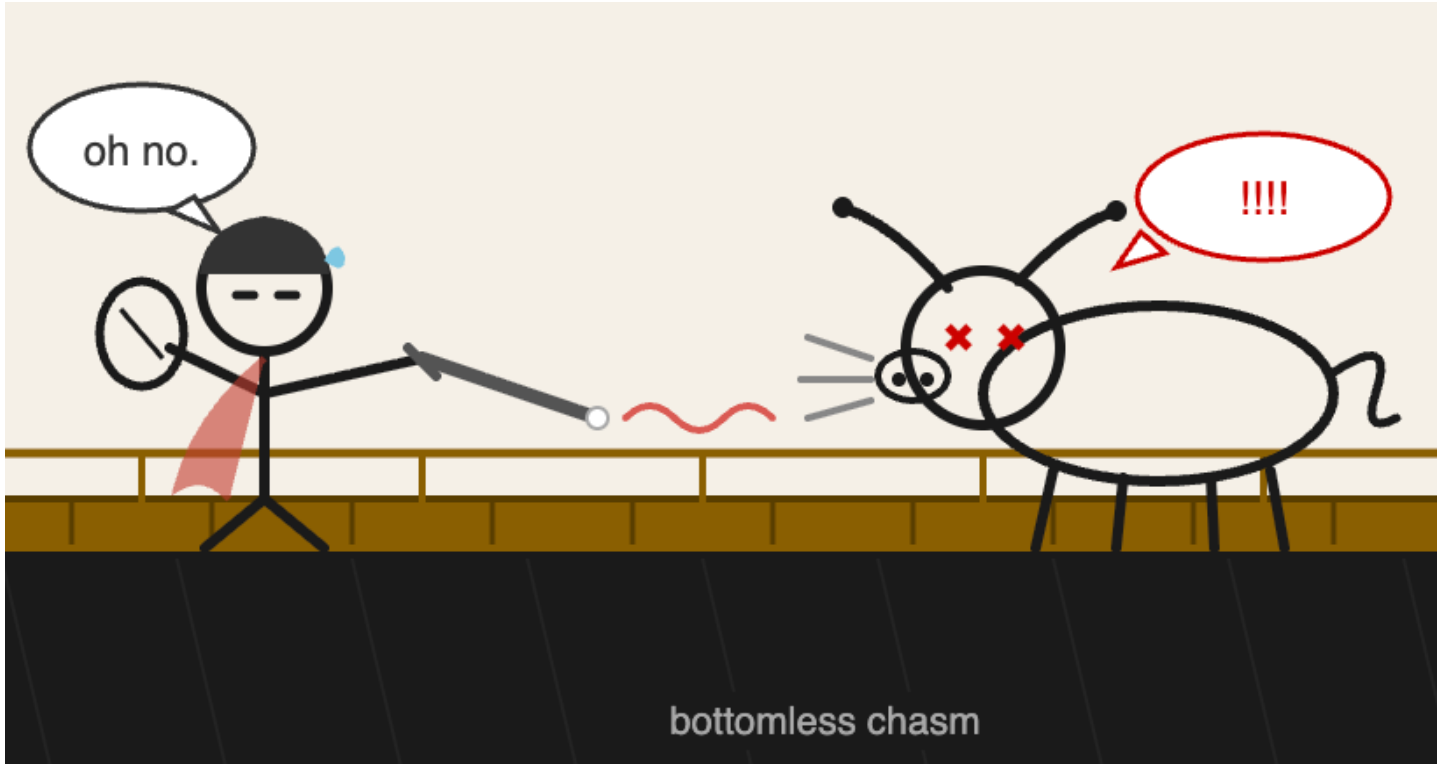
Example 4: LLMs working on a role-playing game.

Problem: What object knows how to resolve an attack?

(Throwing a knife at a monster)

The player?
The monster?
The knife?
The game system?

A ‘resolve’ function,
that can be exported to
an external hexagon and
tested/evolved separately.

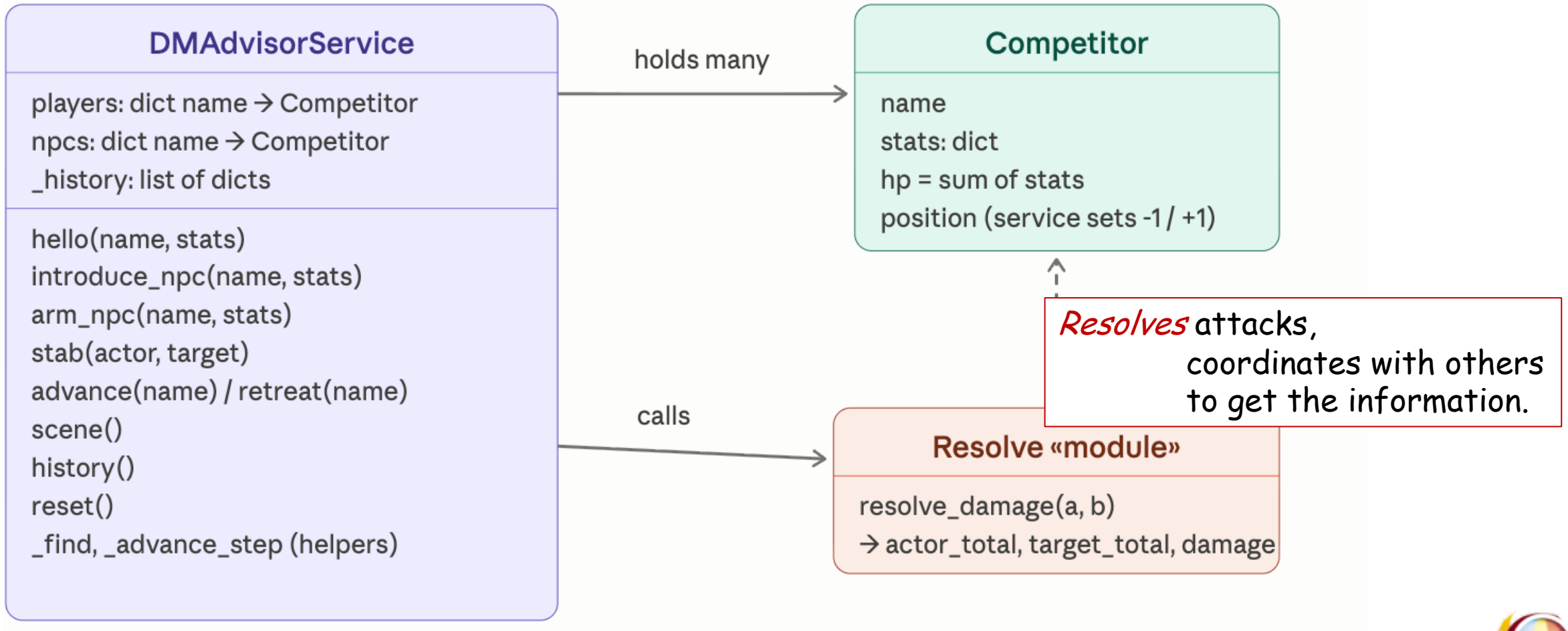


©Alistair Cockburn 2026

Example 4: LLMs working on a role-playing game.

Knows competitors, terrain, history,
but not players' positions
or rules for resolving attacks

Knows its stats & position



Resolves attacks,
coordinates with others
to get the information.

©Alistair Cockburn 2026

Example 4: LLM applies ‘not my job’ to its work!

Typical response in my conversations with an LLM:

"If experiments are now the work, that method needs a living home.

By your own "not my job": PROCESS owns build workflow;
I'd give experiments their own EXPERIMENTS.md
rather than bloat PROCESS.

Your call."

©Alistair Cockburn 2026



Summary: You already have the reflexes to modularize well

(*It's called "allergy to bureaucracies"*☺)

1. "**Not my job**" & "**No need to know**" apply at all scales.
   System partitioning, microservices, classes.
   Applies to your .md files as well as designs.
2. The *responsibility-alignment test* helps 'feel' a design,
   and detects design drift.
3. LLMs understand 'not my job' and 'no need to know.'
4. This design razor is good for humans, even better for LLMs.

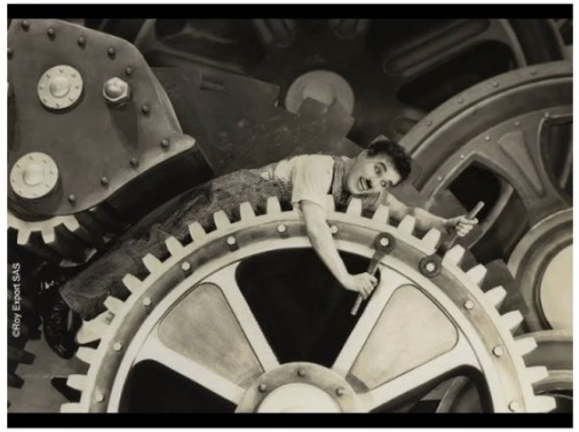
©Alistair Cockburn 2026



Simplifying Software Design

The Genius of Bureaucracies, or How Not-my-Job *Sharpens Your Design*

Preview/Bookstore edition



Alistair Cockburn

The Simplifying Series



Alistair Cockburn

©Alistair Cockburn 2026