# What is Functional Programming?

Eric Normand

PurelyFunctional.tv

## Outline

The problem with software: complexity

Mastering time

Mastering (state)space

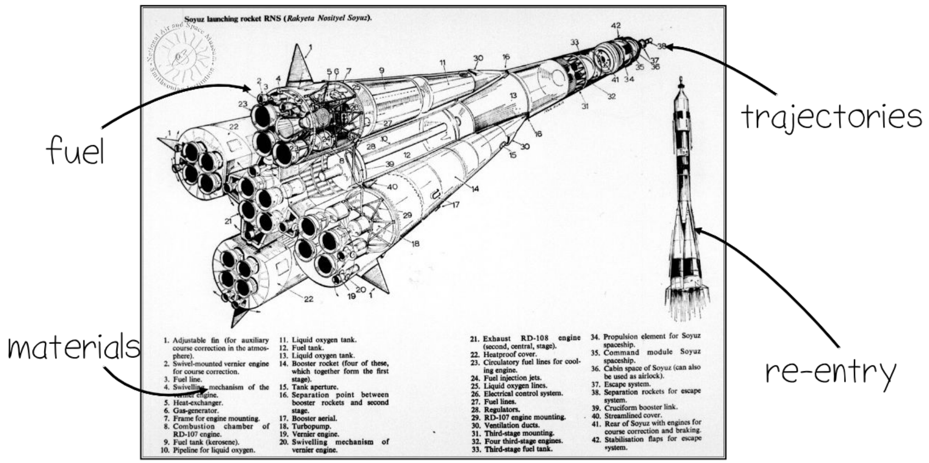
Mastering architecture

A model of functional programming

## The problem with software: complexity

Essential complexity

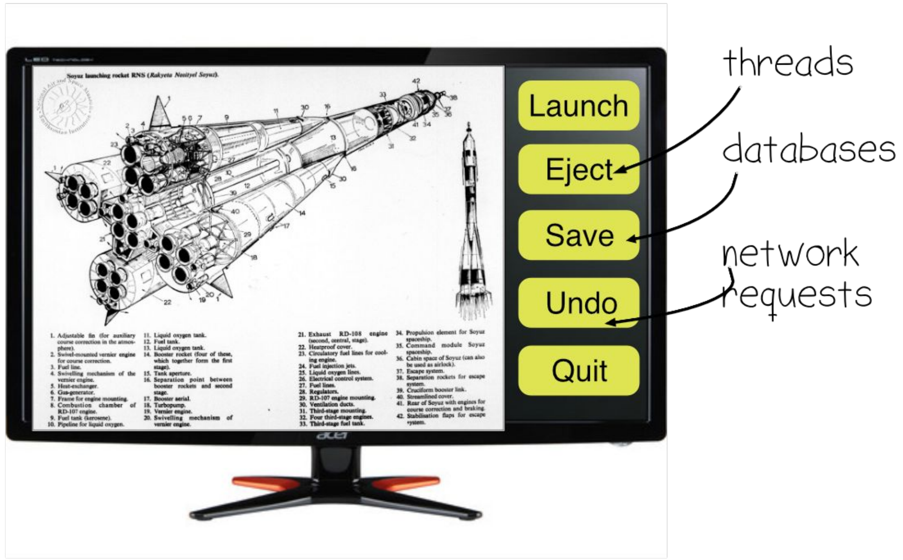
Accidental complexity



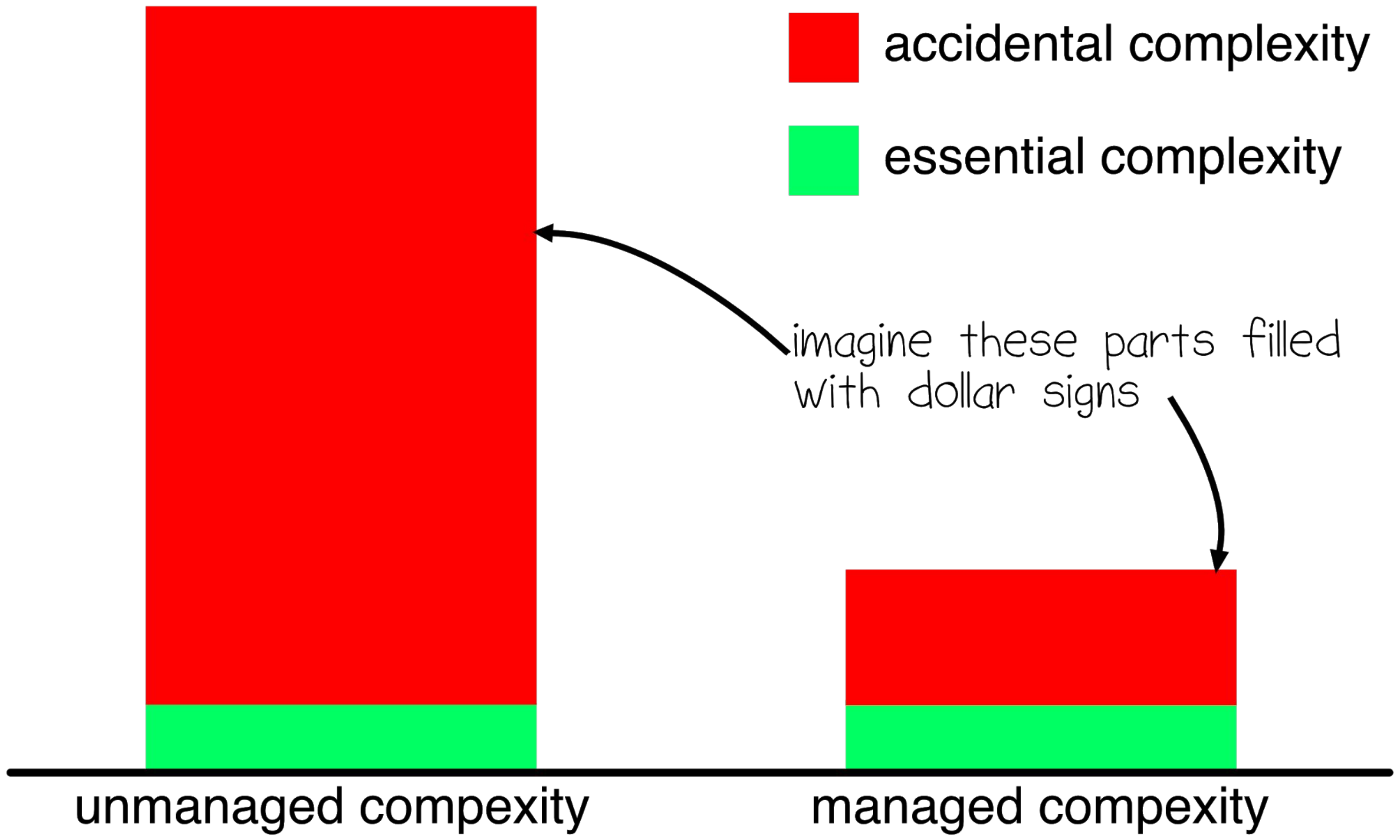
Soyuz launching rocket RNS (*Rakyeta Nosityel Soyuz*).

1. Adjustable fin (for auxiliary course correction in the atmosphere).
2. Swivel-mounted vernier engine for course correction.
3. Fuel line.
4. Swivelling mechanism of the vernier engine.
5. Heat-exchanger.
6. Gas-generator.
7. Frame for engine mounting.
8. Combustion chamber of RD-107 engine.
9. Fuel tank (kerosene).
10. Pipeline for liquid oxygen.
11. Liquid oxygen tank.
12. Fuel tank.
13. Liquid oxygen tank.
14. Booster rocket (four of these, which together form the first stage).
15. Tank aperture.
16. Separation point between booster rockets and second stage.
17. Booster aerial.
18. Turbopump.
19. Vernier engine.
20. Swivelling mechanism of vernier engine.
21. Exhaust RD-108 engine (second, central, stage).
22. Heatproof cover.
23. Circulatory fuel lines for cooling engine.
24. Fuel injection jets.
25. Liquid oxygen lines.
26. Electrical control system.
27. Fuel lines.
28. Regulators.
29. RD-107 engine mounting.
30. Ventilation ducts.
31. Third-stage mounting.
32. Four third-stage engines.
33. Third-stage fuel tank.
34. Propulsion element for Soyuz spaceship.
35. Command module Soyuz spaceship.
36. Cabin space of Soyuz (can also be used as airlock).
37. Escape system.
38. Separation rockets for escape system.
39. Cruciform booster link.
40. Streamlined cover.
41. Rear of Soyuz with engines for course correction and braking.
42. Stabilisation flaps for escape system.

Rocket Science



Software About Rocket Science

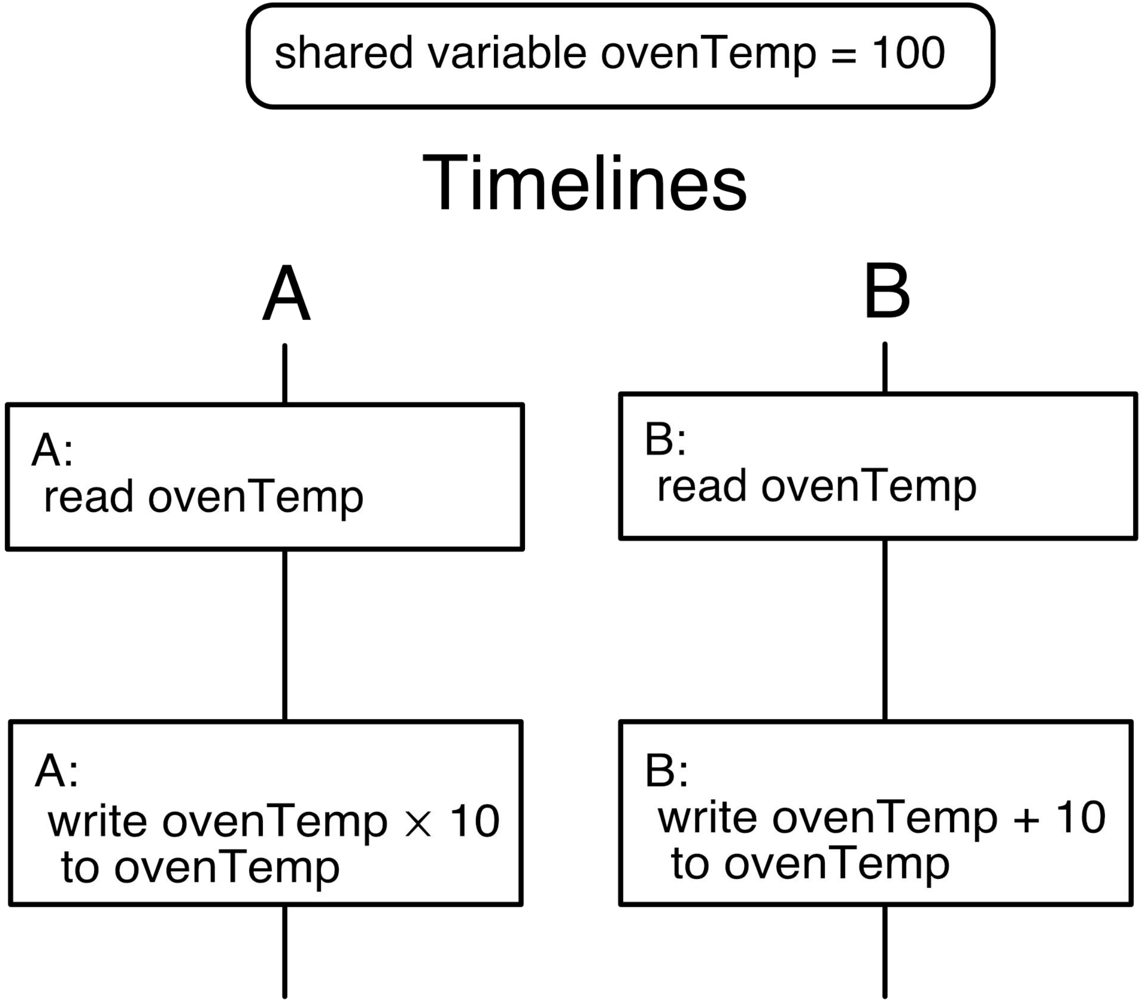

accidental complexity
essential complexity
imagine these parts filled with dollar signs
unmanaged compexity
managed compexity

## Sources of complexity

| | |
|---|---|
| Possible histories | Mastering time |
| Possible codepaths | Mastering (state)space |
| Possible changes | Mastering architecture |

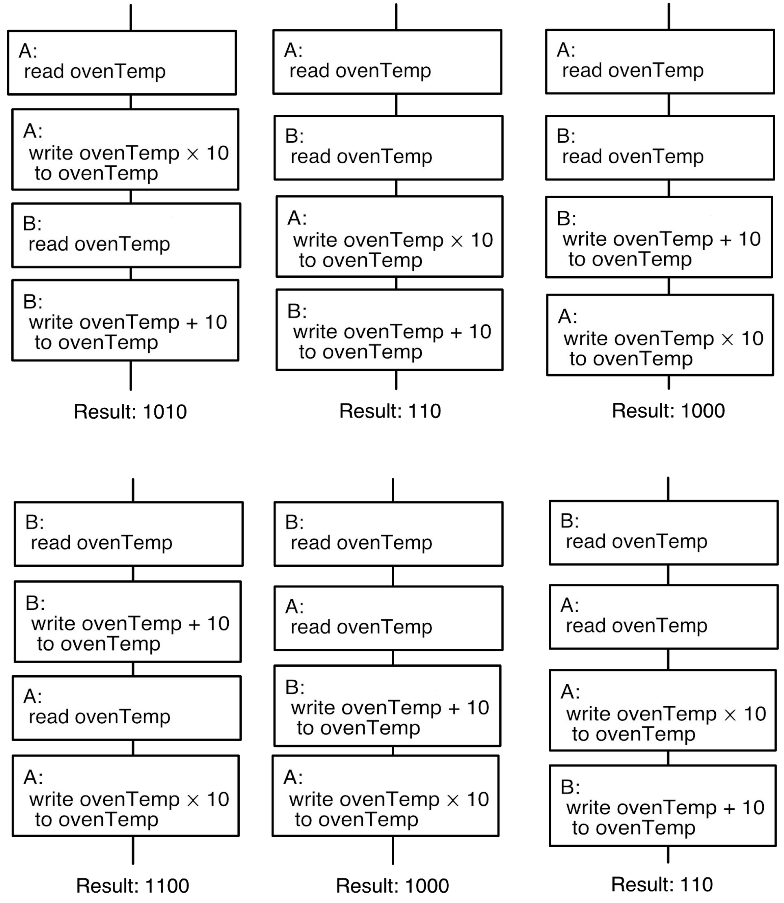
## Mastering time

# Timelines


shared variable ovenTemp = 100
A
B
A: read ovenTemp
B: read ovenTemp
A: write ovenTemp × 10 to ovenTemp
B: write ovenTemp + 10 to ovenTemp

shared variable ovenTemp = 100

## 6 Histories


A: read ovenTemp
A: write ovenTemp × 10 to ovenTemp
B: read ovenTemp
B: write ovenTemp + 10 to ovenTemp
Result: 1010
A: read ovenTemp
B: read ovenTemp
A: write ovenTemp × 10 to ovenTemp
B: write ovenTemp + 10 to ovenTemp
Result: 110
A: read ovenTemp
B: read ovenTemp
B: write ovenTemp + 10 to ovenTemp
A: write ovenTemp × 10 to ovenTemp
Result: 1000
B: read ovenTemp
B: write ovenTemp + 10 to ovenTemp
A: read ovenTemp
A: write ovenTemp × 10 to ovenTemp
Result: 1100
B: read ovenTemp
A: read ovenTemp
B: write ovenTemp + 10 to ovenTemp
A: write ovenTemp × 10 to ovenTemp
Result: 1000
B: read ovenTemp
A: read ovenTemp
A: write ovenTemp × 10 to ovenTemp
B: write ovenTemp + 10 to ovenTemp
Result: 110

## JavaScript has this problem, too

```
var ovenTemperature = 100;

ajaxGet("http://api.com/number", function(number) {
  ovenTemperature *= number;
});

ajaxGet("http://api.com/number", function(number) {
  ovenTemperature += number;
});
```

## Where do timelines come from?

Multiple threads

Multiple processes

Multiple machines

Async operations

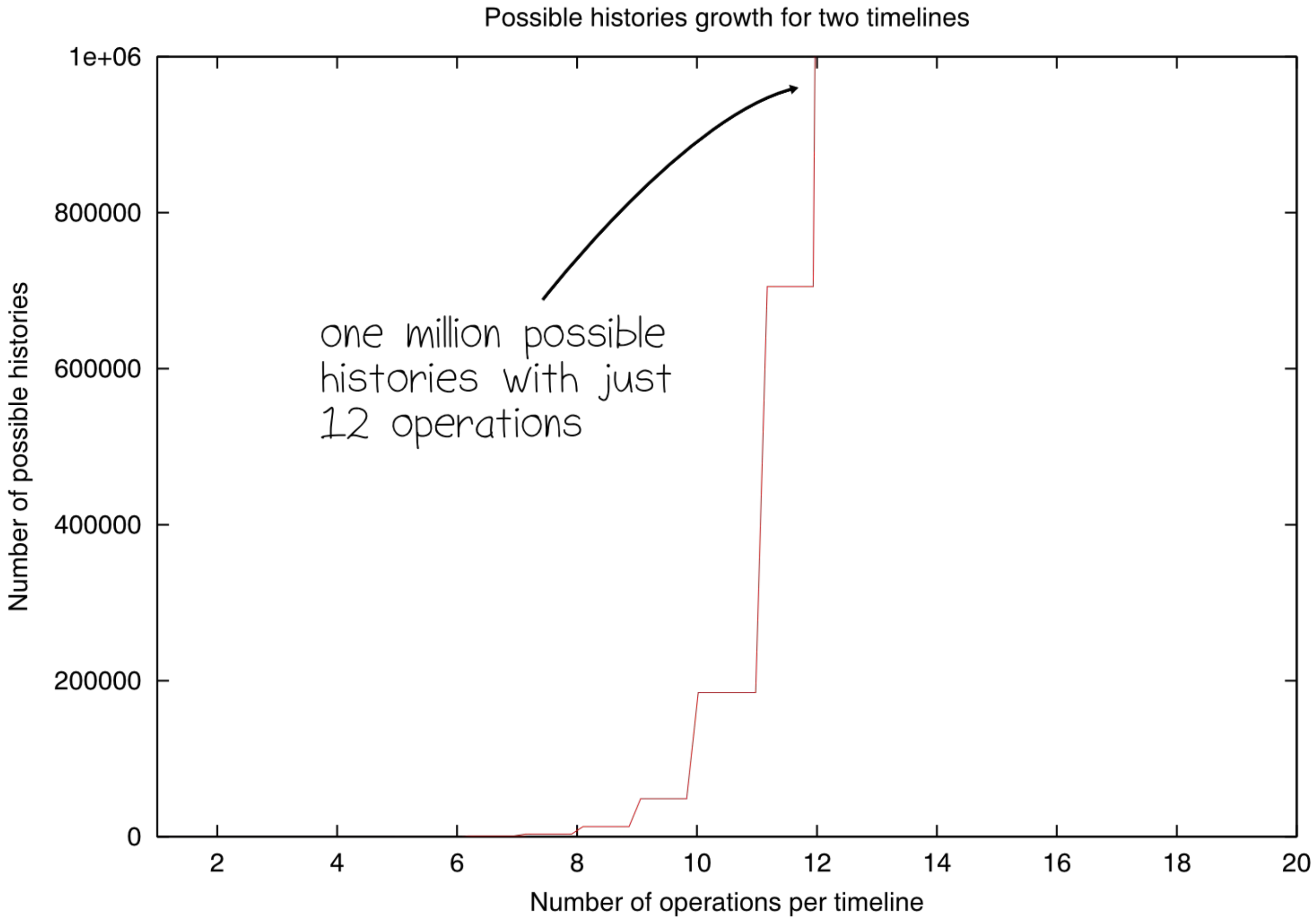
## What's the problem?

Many histories are more than we can keep in our heads

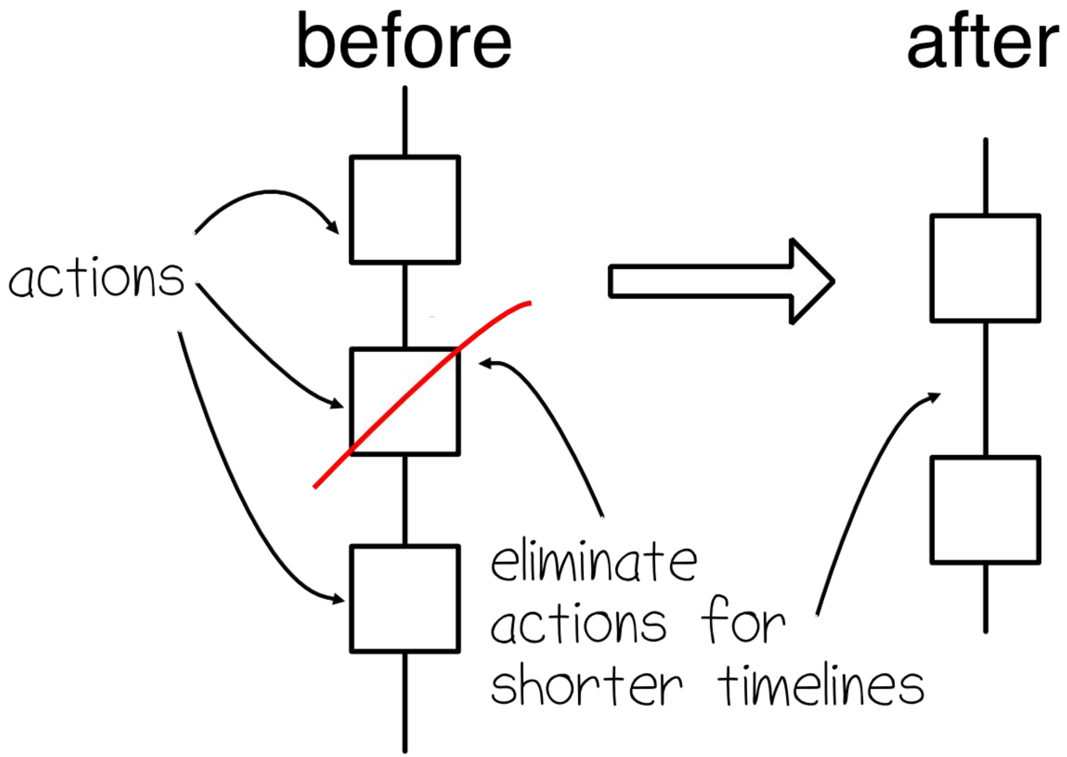
Different histories give different results

Sometimes we can't reproduce the bad history (heisenbug)

$$h = \frac{(ta)!}{(a!)^t}$$


Possible histories growth for two timelines
Number of possible histories
Number of operations per timeline
1e+06
800000
600000
400000
200000
0
2
4
6
8
10
12
14
16
18
20
one million possible histories with just 12 operations

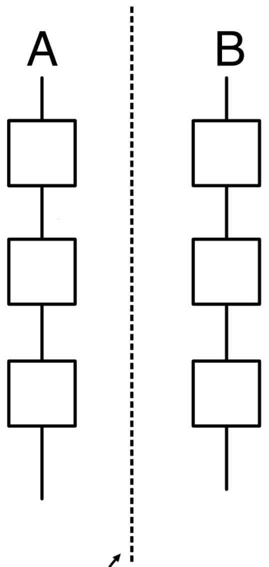
# Timeline



3 actions = 20 histories

2 actions = 6 histories

## Timelines



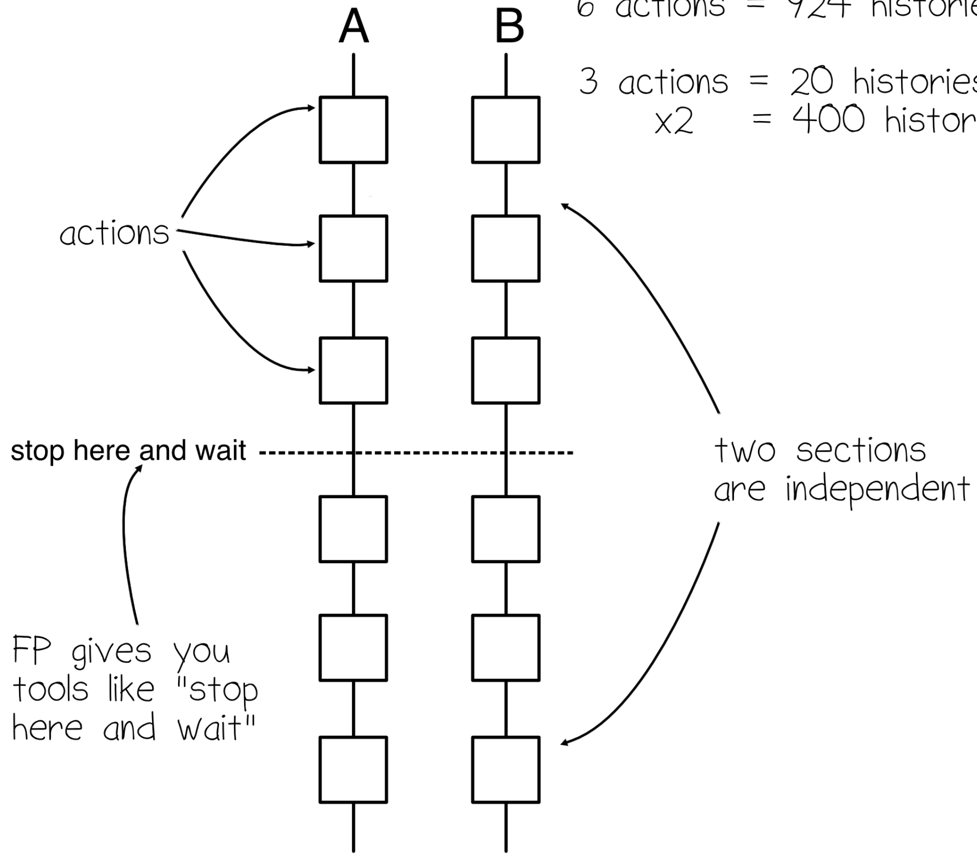
No shared resources,
all histories give the same
result

No shared resources:
the two are isolated and
we don't have to worry
about interactions

# Timelines

6 actions = 924 histories

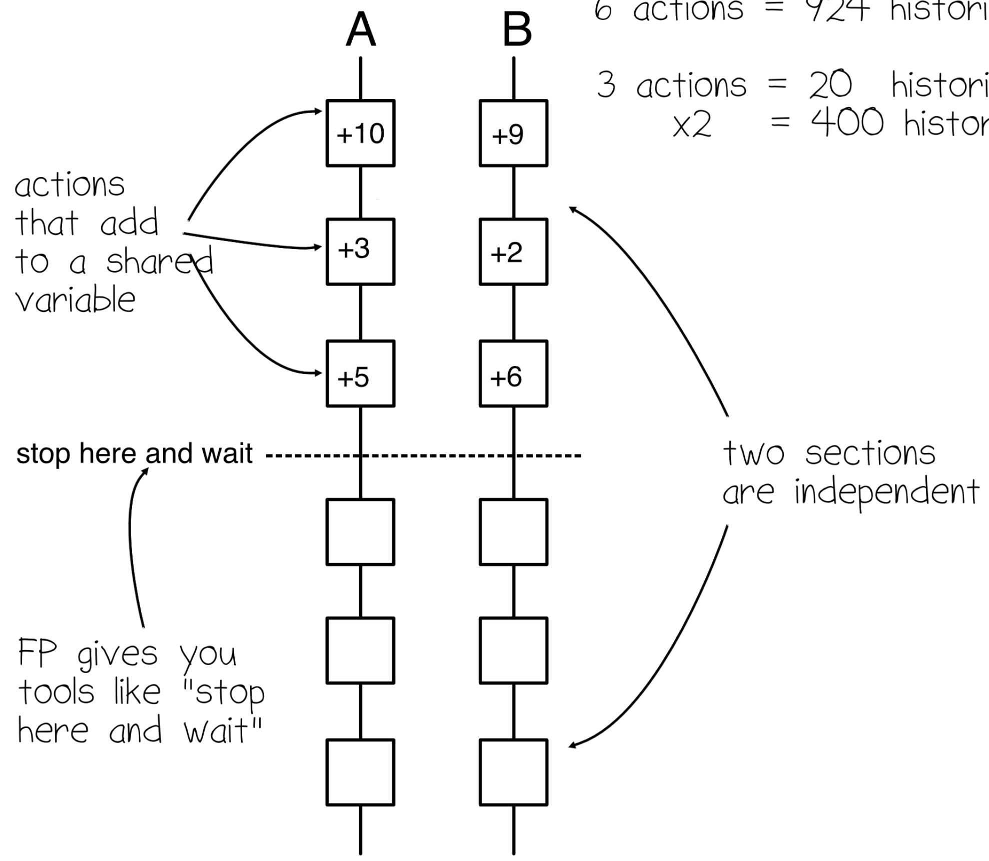
3 actions = 20 histories
x2 = 400 histories

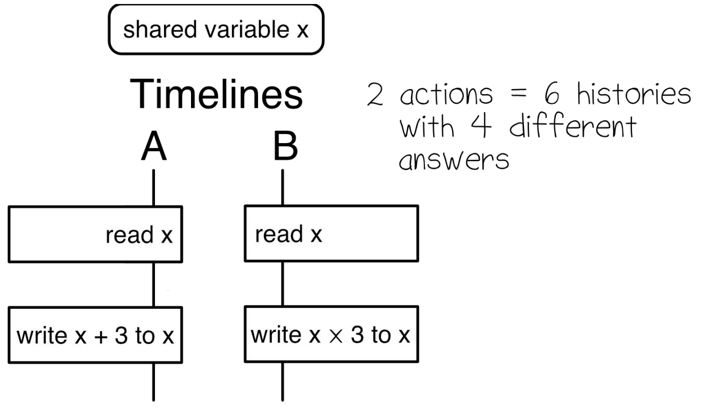

A
B
actions
stop here and wait
two sections are independent
FP gives you tools like "stop here and wait"

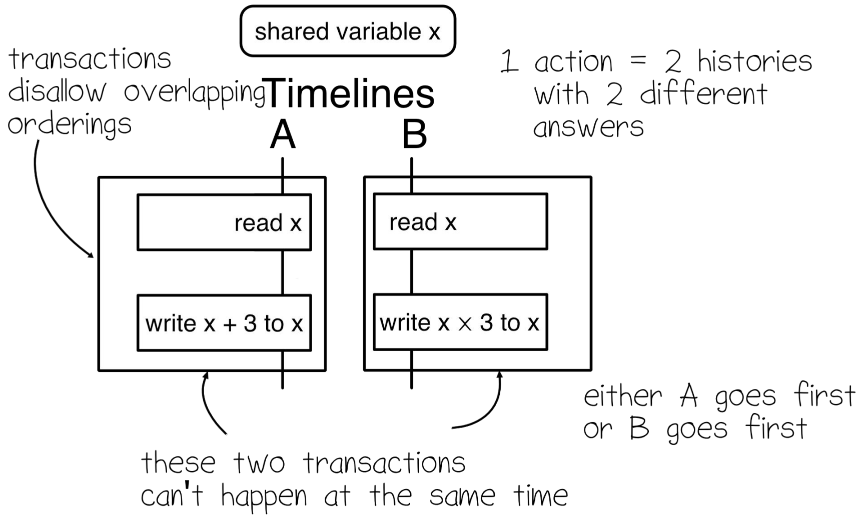
# Timelines

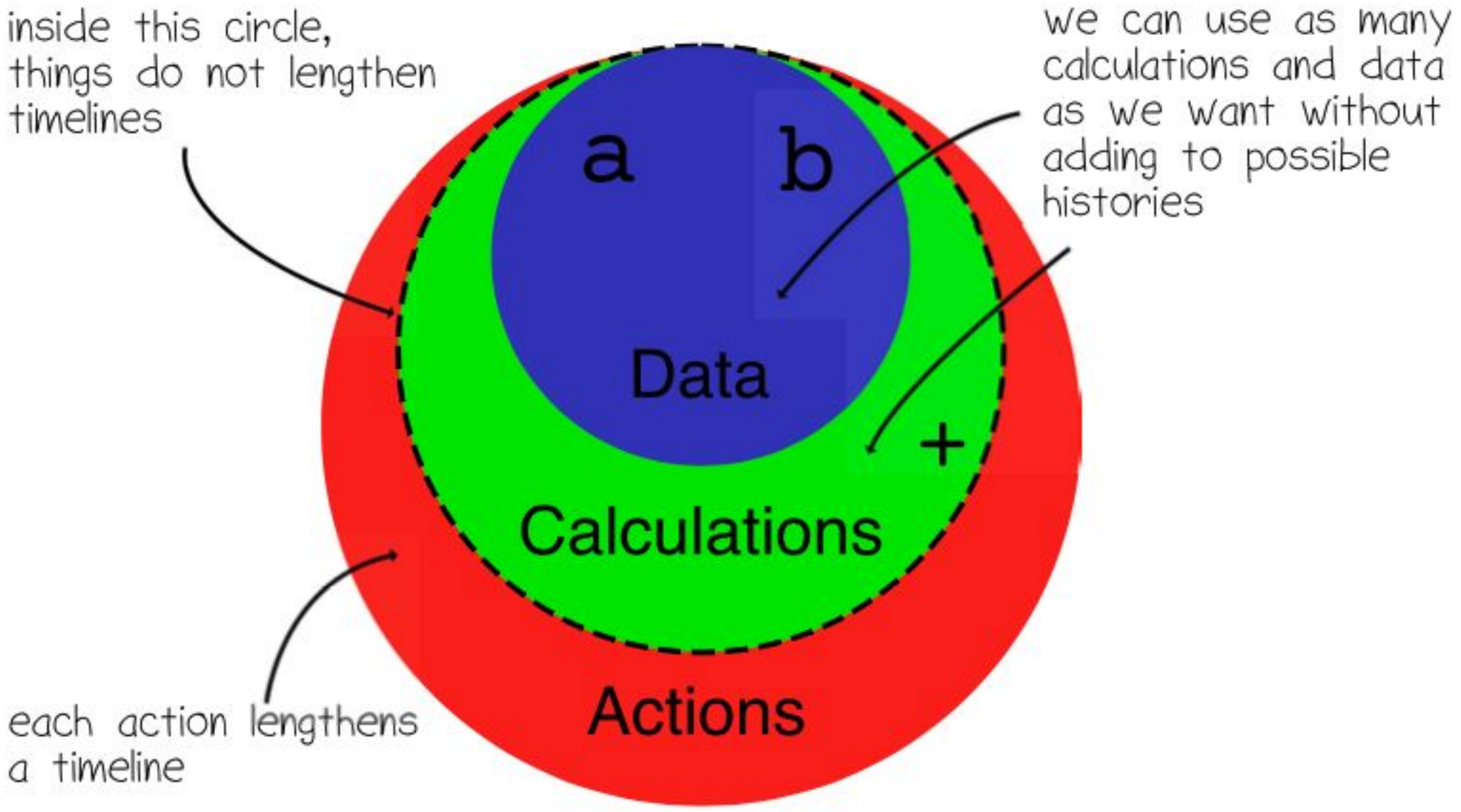
6 actions = 924 histories

3 actions = 20 histories
x2 = 400 histories


A
B
+10
+3
+5
+9
+2
+6
actions that add to a shared variable
stop here and wait
FP gives you tools like "stop here and wait"
two sections are independent


shared variable x
Timelines
A
B
read x
write x + 3 to x
read x
write x × 3 to x
2 actions = 6 histories with 4 different answers


shared variable x
transactions disallow overlapping orderings
Timelines
A
B
1 action = 2 histories with 2 different answers
read x
write x + 3 to x
read x
write x × 3 to x
either A goes first or B goes first
these two transactions can't happen at the same time


inside this circle, things do not lengthen timelines
we can use as many calculations and data as we want without adding to possible histories
a
b
Data
+
Calculations
Actions
each action lengthens a timeline

## Mastering (state)space

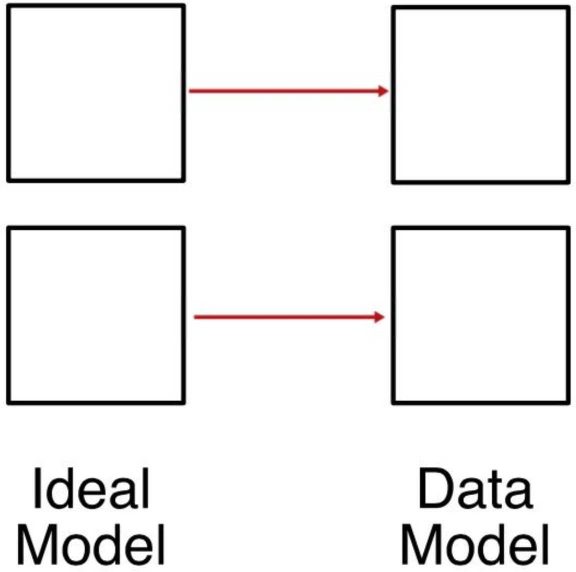
Each conditional creates at least 2 branches

Branches multiply the number of possible codepaths

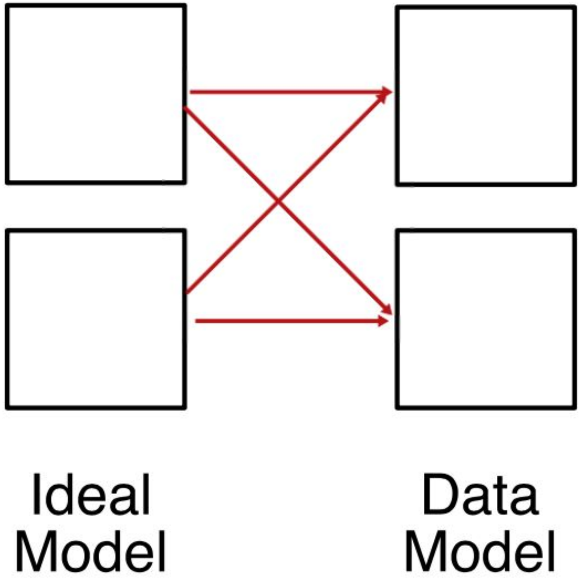
More codepaths means it's harder to hold in your head

Do all codepaths do the right thing?

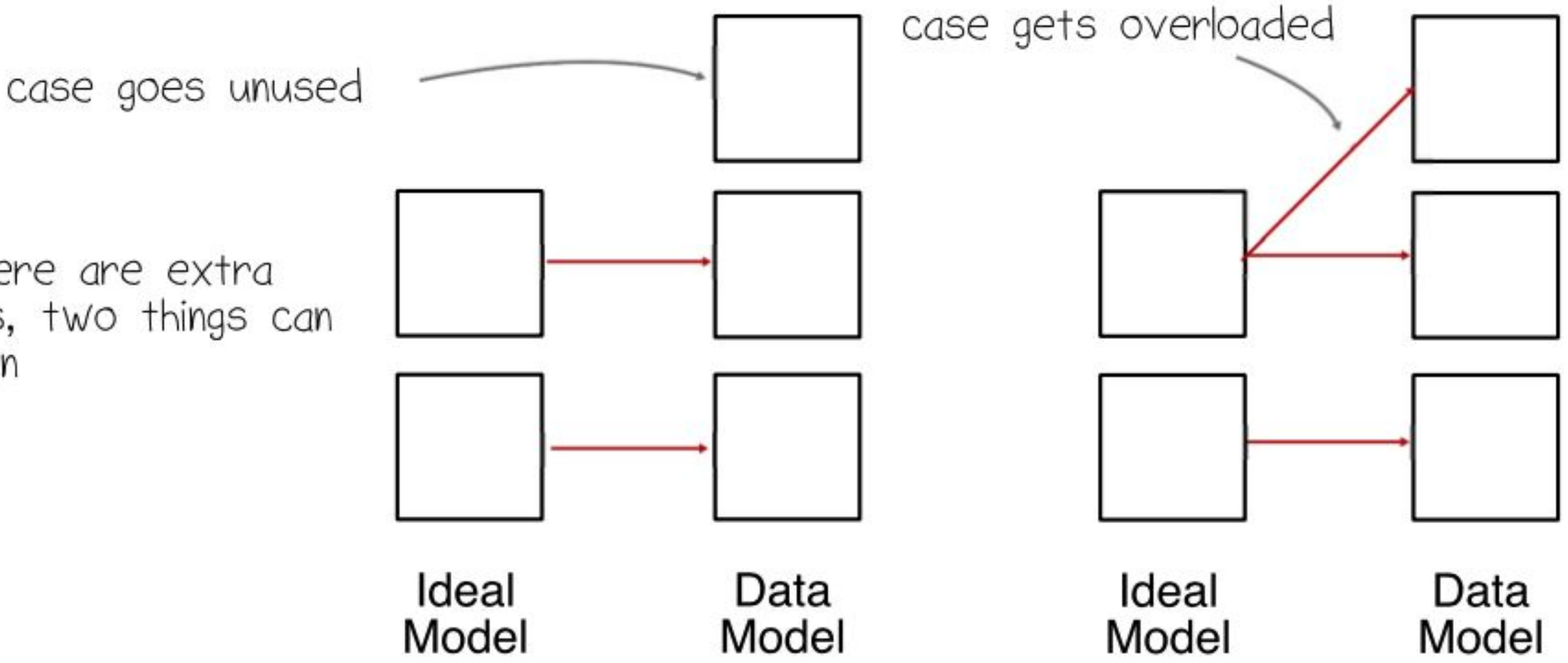
in the ideal world
the ideal model's cases
map cleanly to the data
model cases



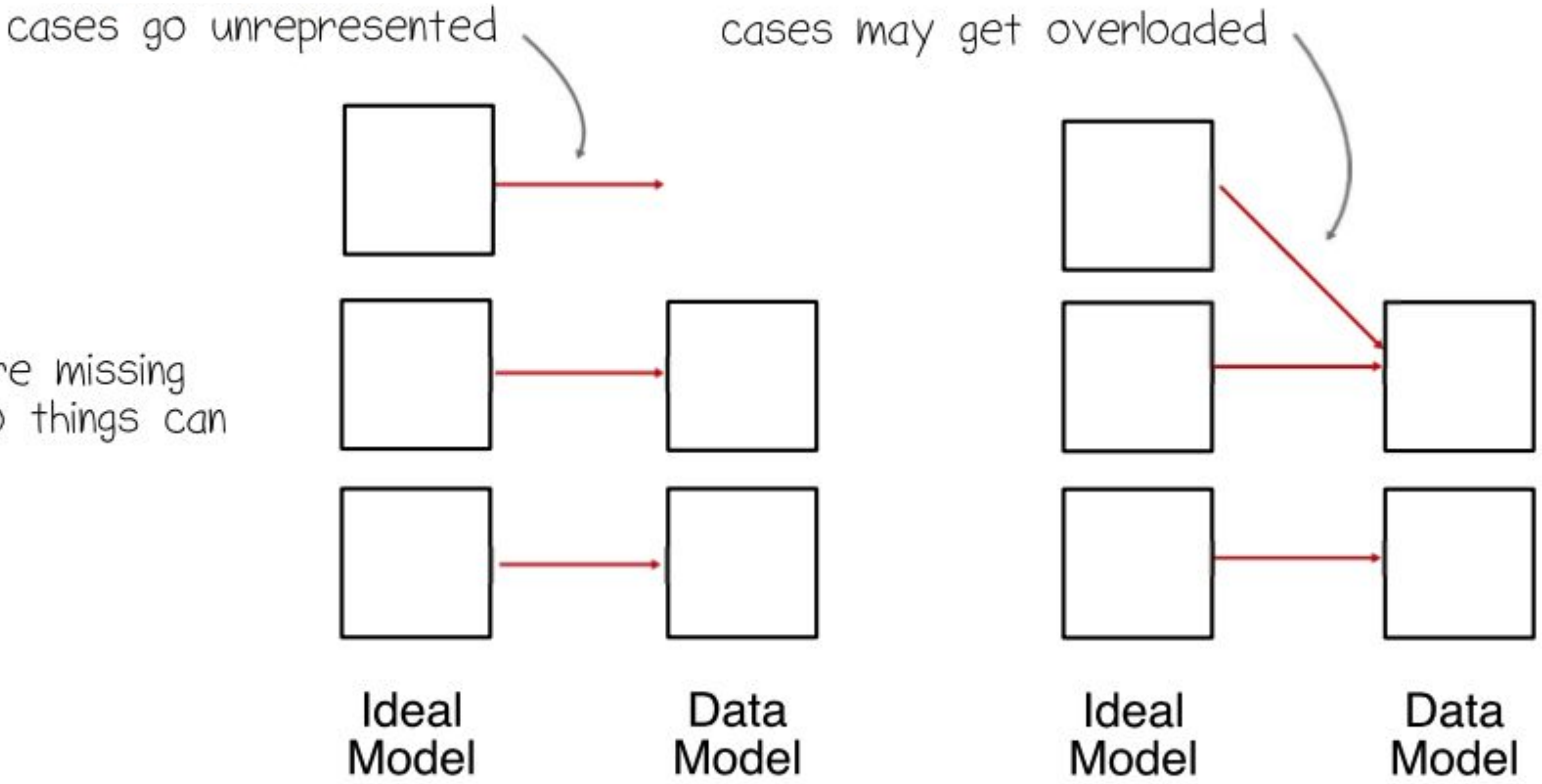
sometimes the mapping is convoluted

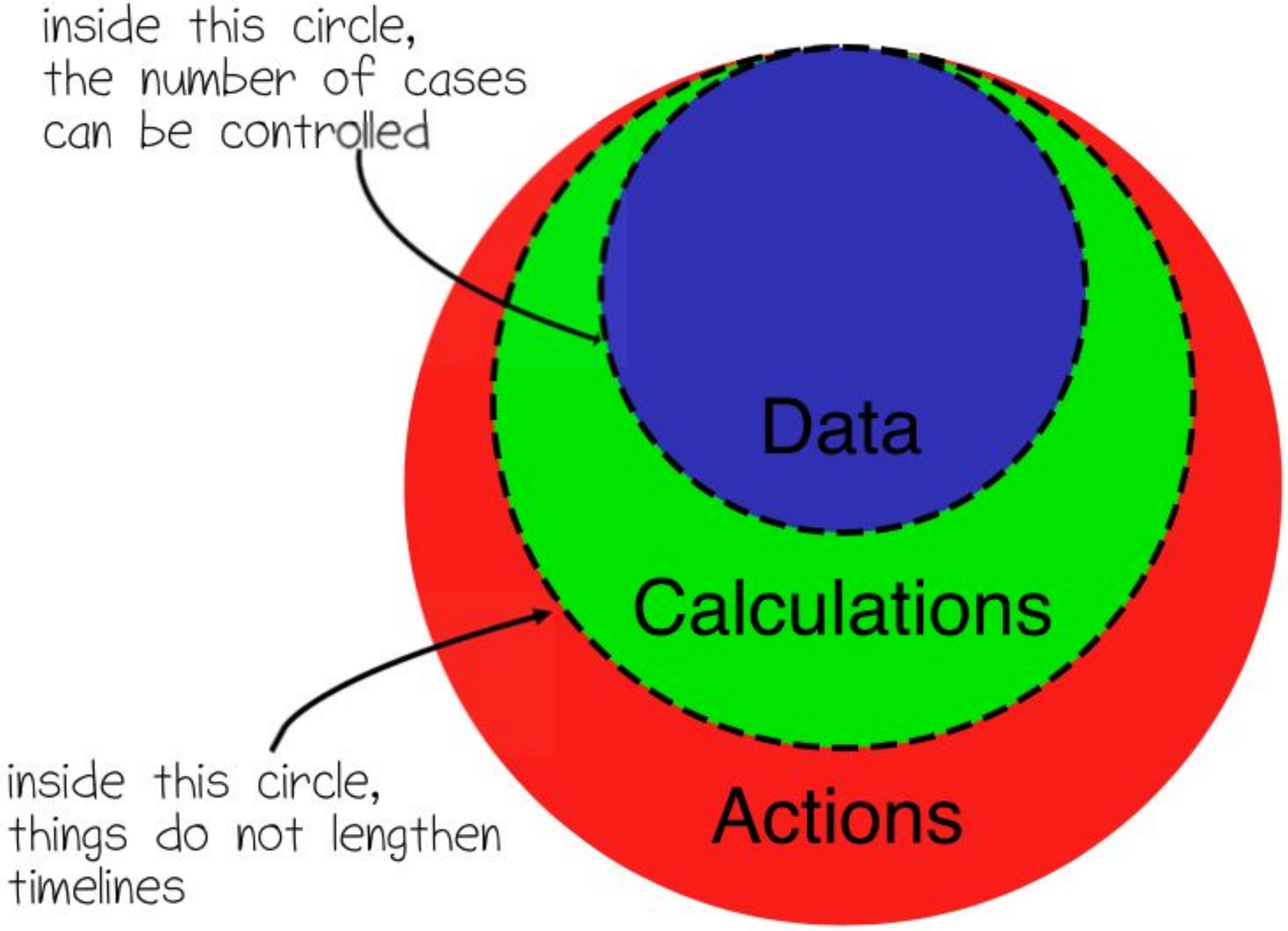


if there are extra cases, two things can happen



if there are missing cases, two things can happen




inside this circle, the number of cases can be controlled
Data
Calculations
Actions
inside this circle, things do not lengthen timelines

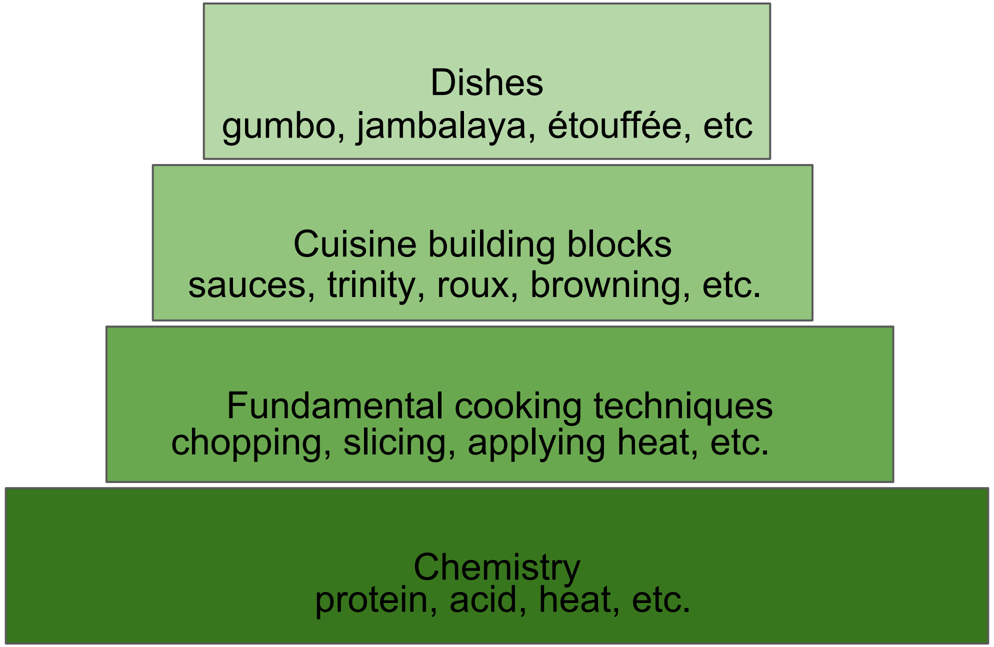
## Mastering architecture

Guarding against unforeseen change

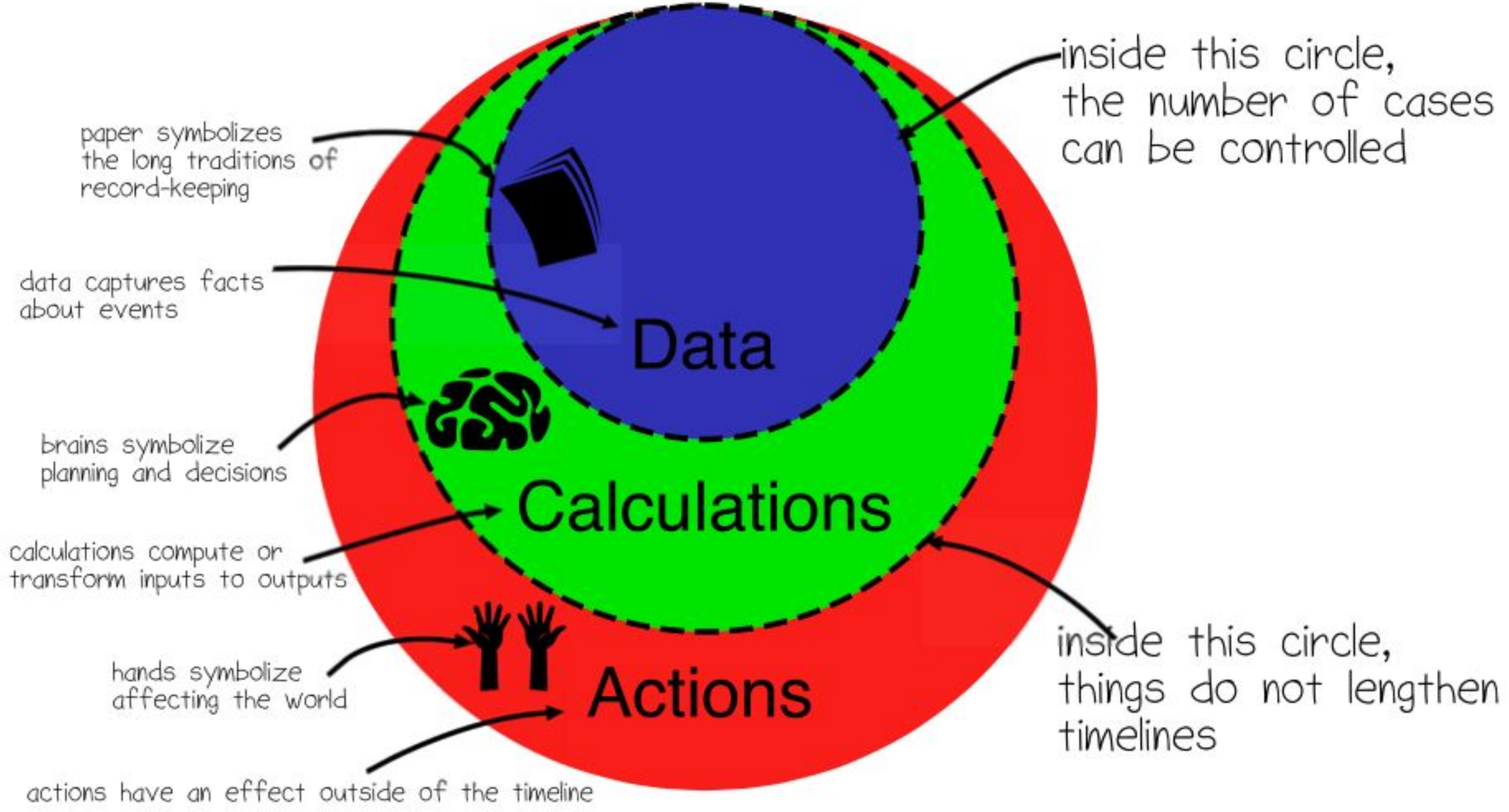
## Stratified design

Layers built on layers

Each layer adds domain meaning to the layer below it


Dishes
gumbo, jambalaya, étouffée, etc
Cuisine building blocks
sauces, trinity, roux, browning, etc.
Fundamental cooking techniques
chopping, slicing, applying heat, etc.
Chemistry
protein, acid, heat, etc.

## A model of functional programming


paper symbolizes the long traditions of record-keeping
data captures facts about events
brains symbolize planning and decisions
calculations compute or transform inputs to outputs
hands symbolize affecting the world
actions have an effect outside of the timeline
inside this circle, the number of cases can be controlled
inside this circle, things do not lengthen timelines
Data
Calculations
Actions

Action + Action => Action
Action + Calculation => Action
Action + Data => Action

Calculation + Calculation => Calculation
Calculation + Data => Calculation

Data + Data => Data

## Eric Normand

LispCast

**Follow Eric on:**

Eric Normand

@EricNormand

lispcast.com

eric@lispcast.com