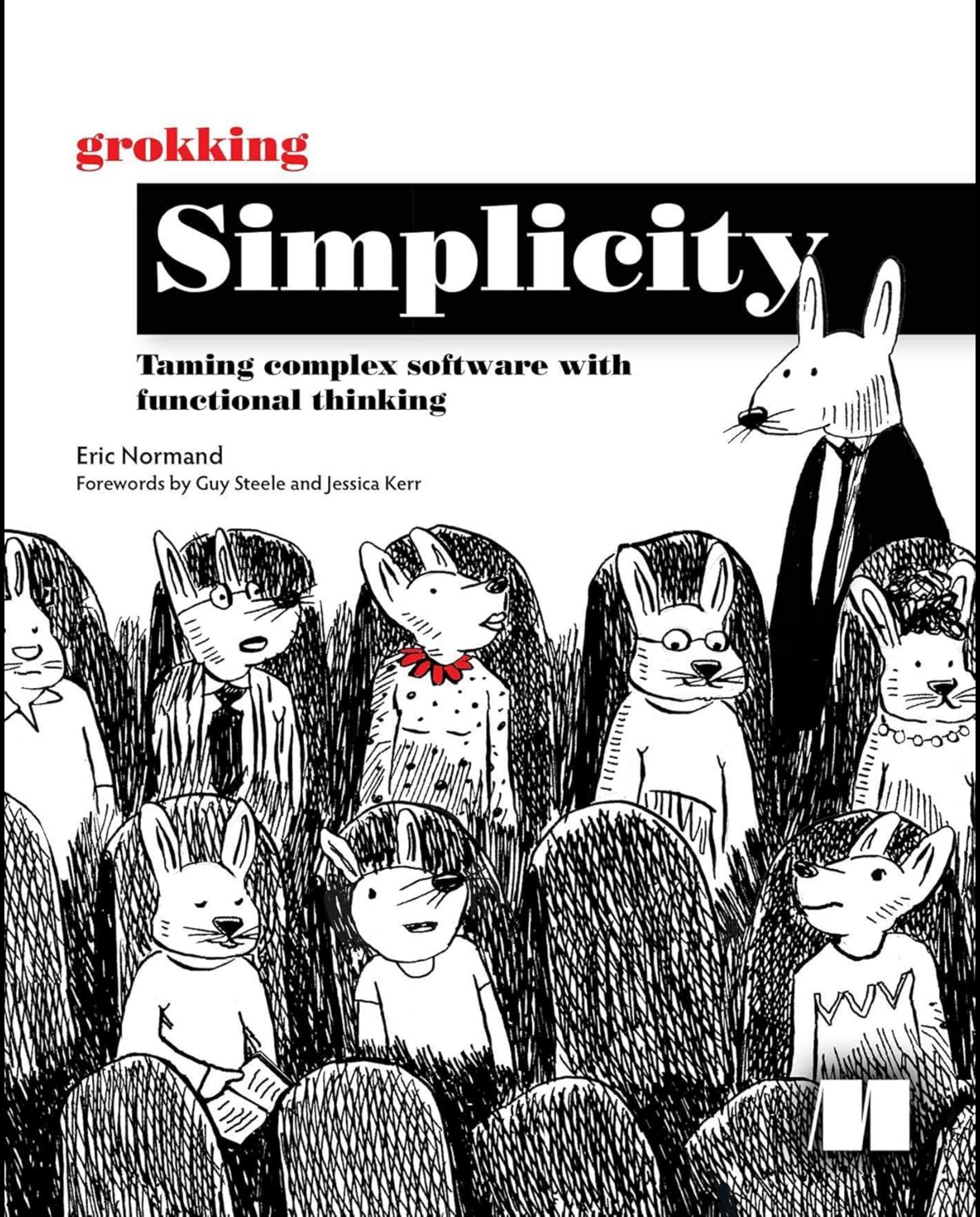
Two and a half more Domain Modeling Lenses

Eric Normand - Houston Functional Programming User Group



grokking

Simplicity

Taming complex software with functional thinking

Eric Normand

Forewords by Guy Steele and Jessica Kerr

ericnormand.me/gs

ericnormand.me/gsm TSSIMPLICITY

Software design is subtle

Good information → good decisions → good design

- Data
- Operations
- Composition
- Time

- Domain
- Scope
- Platform
- Volatility
- Runnable specifications

- Data
- Operations
- Composition
- Time

- Domain
- Scope
- Platform
- Volatility
- Runnable specifications

```
{
  “size”: “super”,


  “roast”: “burnt”,


  “add-ins”: [“espresso”, “soy”]
}
```

Better Software Design with Domain Modeling

https://ericnormand.me/speaking/func-prog-sweden-2023

- Data
- Operations
- Composition
- Time

- Domain
- Scope
- Platform
- Volatility
- Runnable specifications

Commutativity

Order of function calls doesn't matter

```
for(let i = 0; i < 100; i++) {
  let coffee = anyCoffee();
  let addInA = anyAddIn();
  let addInB = anyAddIn();
  assert(sameCoffee(
    coffee.add(addInA).add(AddInB),
    coffee.add(addInB).add(AddInA)
  ));
}
```

```
g(f(a)) = f(g(a))
```

Four More Domain Modeling Lenses

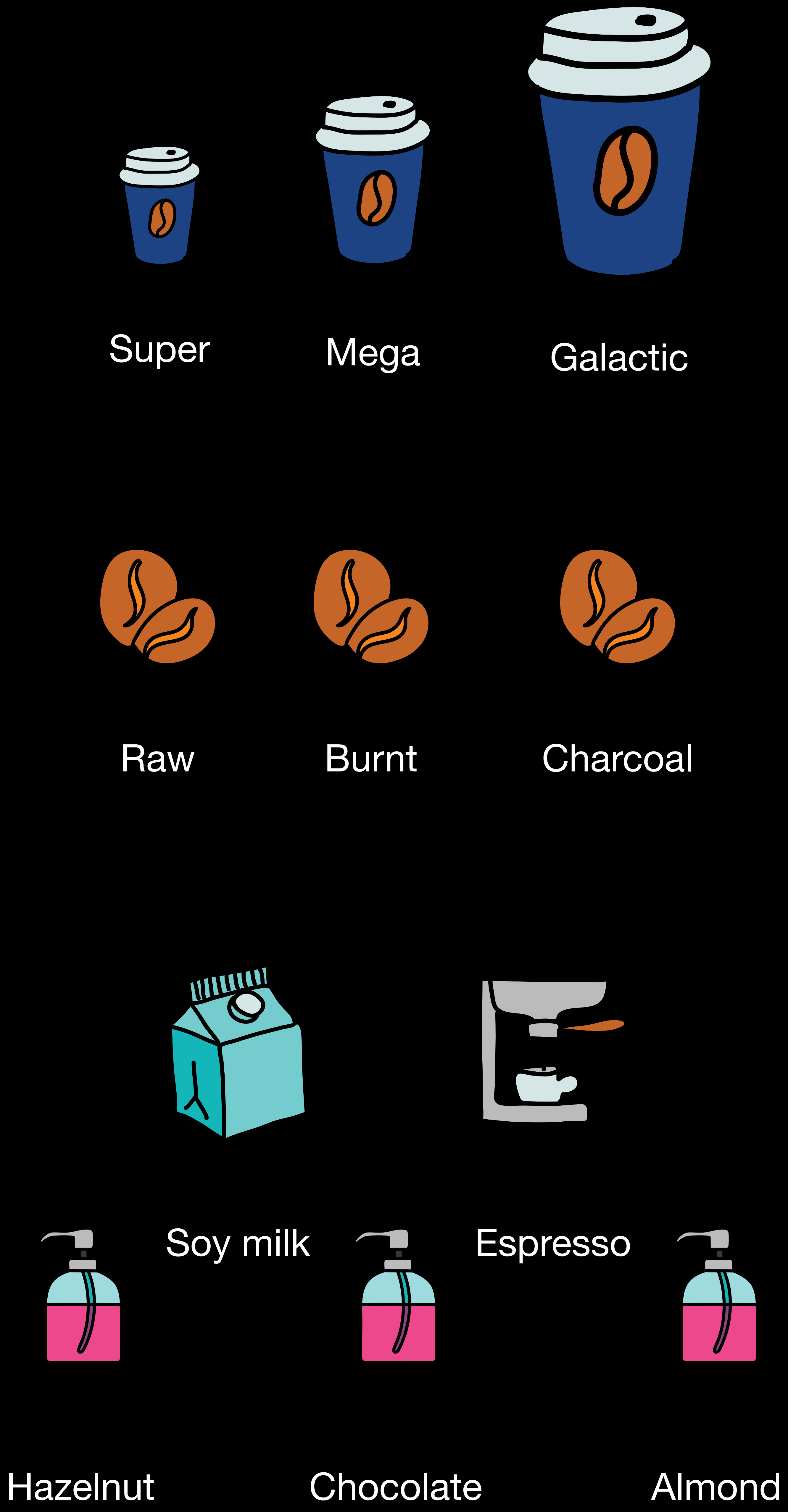
https://ericnormand.me/speaking/houston-fpug-2024

- Data
- Operations
- Composition
- Time

- Domain
- Scope
- Platform
- Volatility
- Runnable specifications

- Data
- Operations
- Composition
- Time

- Domain
- Scope
- Platform
- Volatility
- Runnable specifications

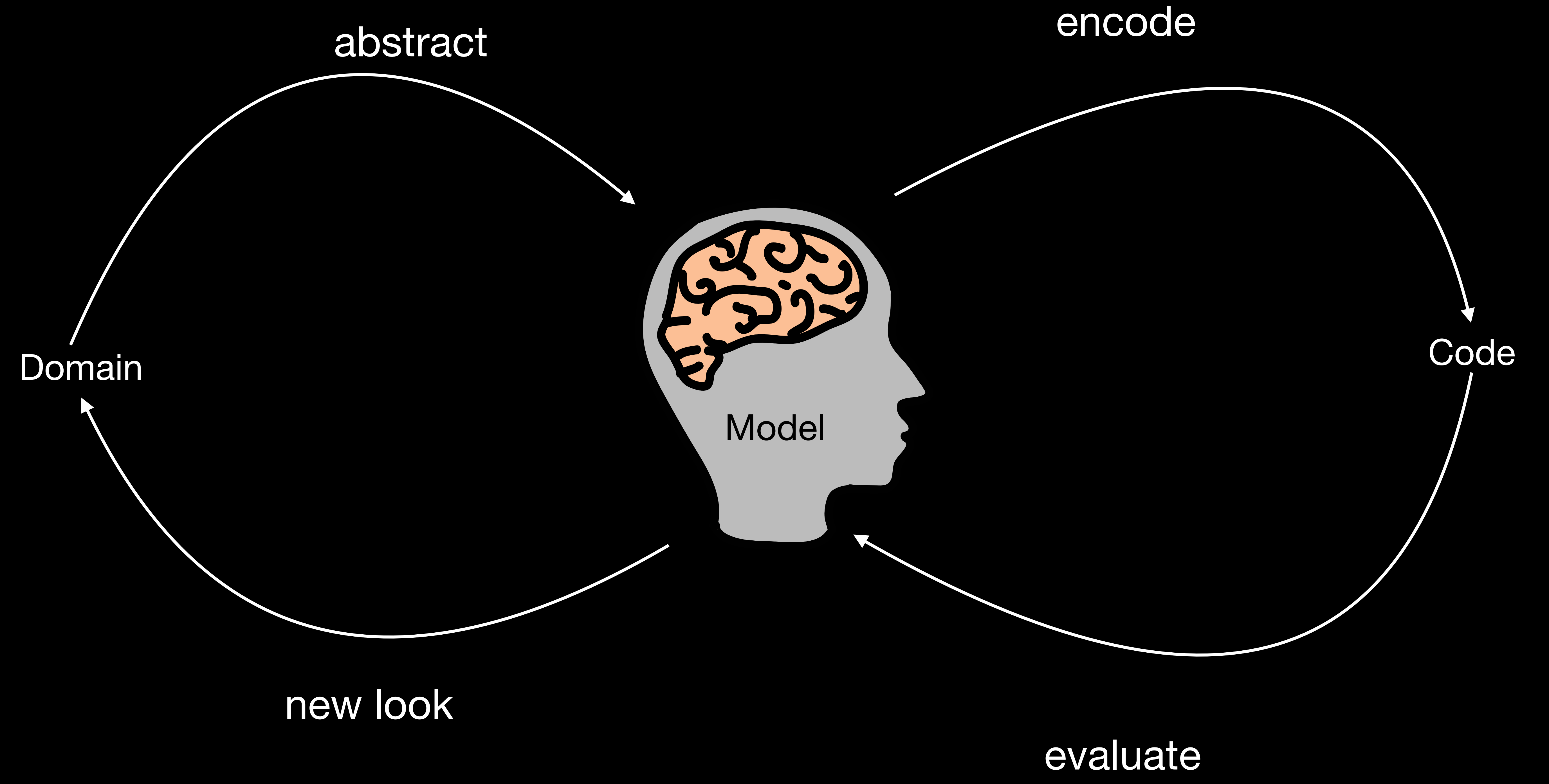


```
{
  “size”: “super”,

  “roast”: “burnt”,

  “add-ins”: {“espresso” : 1,
              “soy”      : 2}
}
```

Runnable specifications



abstract

encode

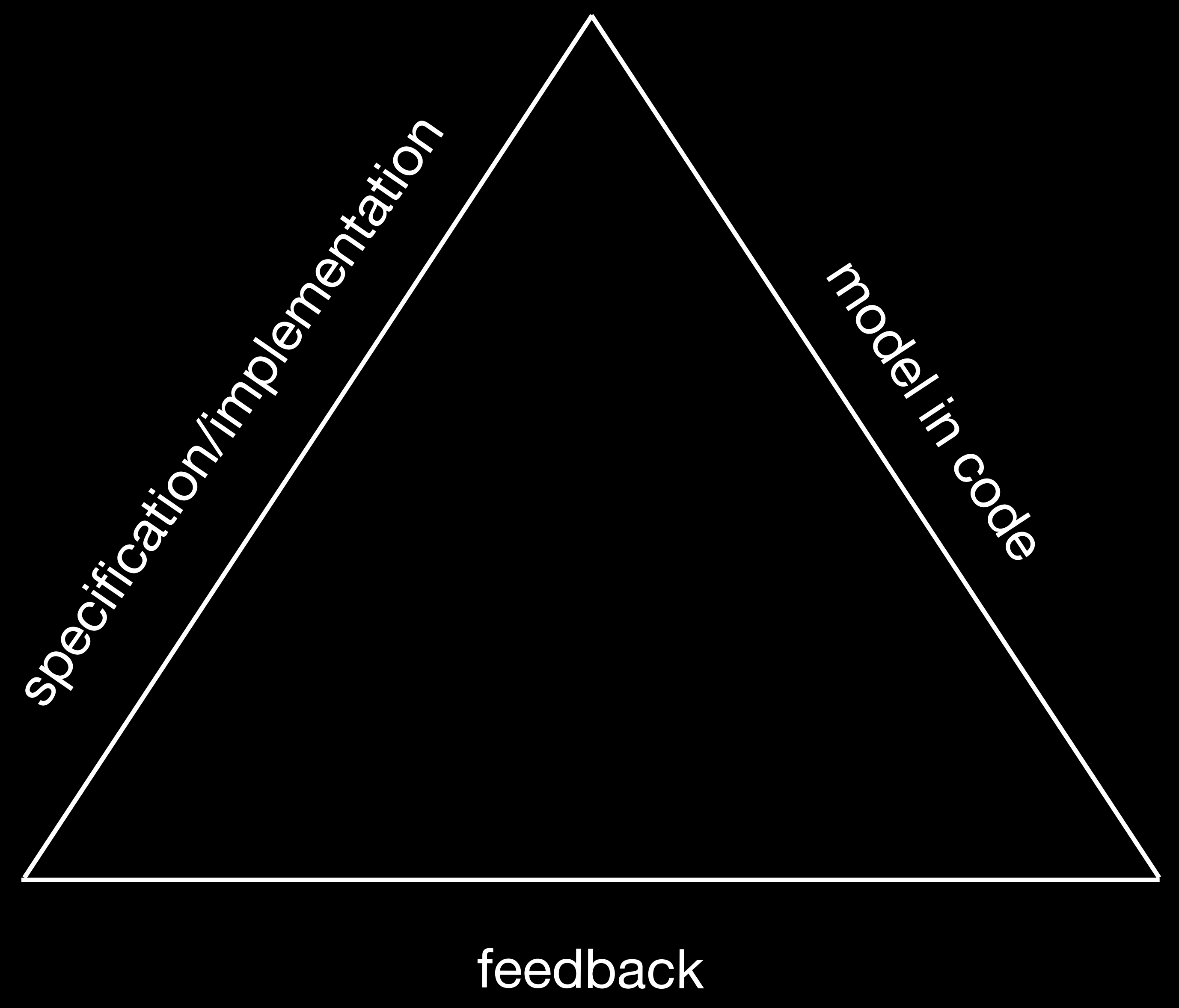
Domain

Model

Code

new look

evaluate


specification/implementation

model in code

feedback

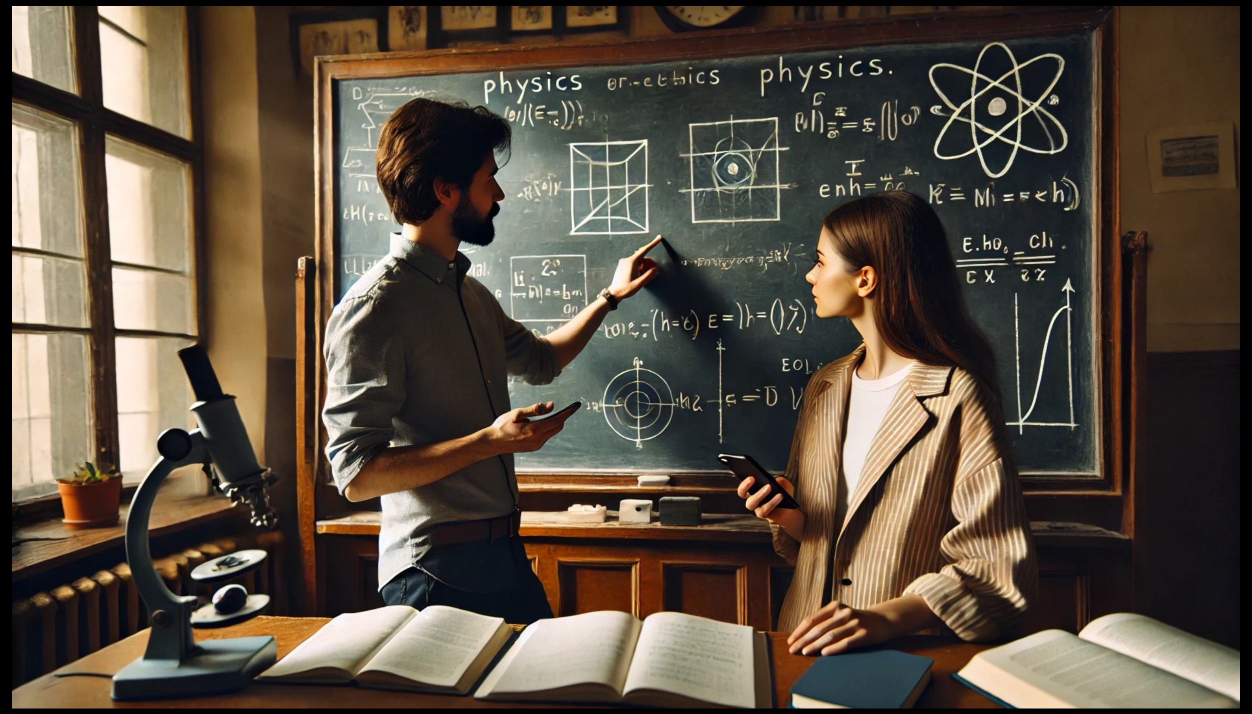
Model in code

What is modeling?


physics
physics.

Why do we model?

Model to learn



Model to communicate

Model to record data and relationships

Model to learn

- Reduce the number of variables/constrain the problem
- Build as little as possible to answer the question
- Defer decisions that don’t answer question
  - Don’t implement
  - Implement with a stub
- Answer the hard questions first

Model to communicate

communicating to programmers, stakeholders, and the computer

- Try your model on use cases
- Run your model
    - Set up scenarios
    - Visualize

Model to record data and relationships

- This is the goal
- The other steps lead up to this
- Testing

Separate implementation from specification

Specification

English description

UML diagrams

runnable specifications

code

Implementation

code

code

code

Specification

type names
+
function signatures

```
type Coffee;
type AddIn;
type AddInCollection;

function addIns(coffee) //=> AddInCollection
function addAddIn(coffee, addIn) //=> Coffee
```

Implementation

type definition
+
function body

```
type Coffee = {
  …
  addIns: AddInCollection;
};

type AddIn = Soy | Espresso | Hazelnut | …;

type AddInCollection = { [addIn: string]: number; };

function addAddIn(coffee, addIn) { //=> Coffee
  return update(coffee, “addIns”, append, addIn);
}
```

Specification

denotational semantics

```
function factorial(n) {
  if(n === 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

Implementation

operational semantics

```
function factorial(n) {
  let ret = 1;
  for (let i = 1; i <= n; i++)
    ret *= i;
  return ret;
}
```

Specification

function definition

```
function coffeePrice(coffee) { //=> number
  return sizePrice(size(coffee)) +
         addInCollectionPrice(addIns(coffee));
}
```

Assumes existence of four other domain functions
- size()
- sizePrice()
- addIns()
- addInCollectionPrice()

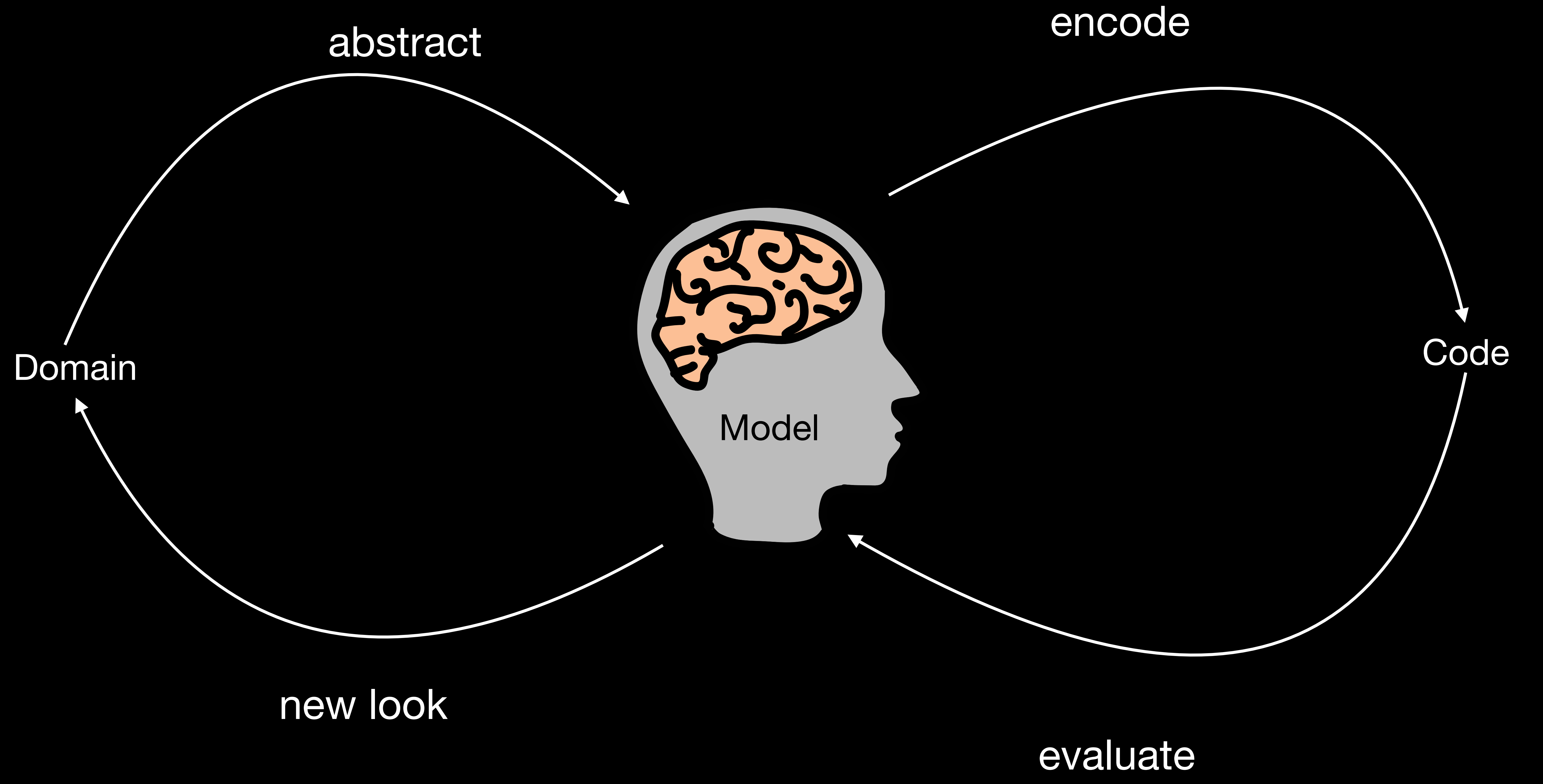
Implementation

function implementation

```
function coffeePrice(coffee) { //=> number
  let price = sizePrice(coffee.size));
  for (const [addIn, quantity] of coffee.addIns)
    price += addInPrice(addIn) * quantity;
  return price;
}
```

Assumes lots:
- sizePrice()
- structure of coffee (coffee.size)
- structure of addIns (object)
- addInPrice()
- algorithm for calculating the price

Frequent and rich feedback


abstract
encode
Domain
Model
Code
new look
evaluate

Making decisions is better with feedback

- Stubs
    - Work in-memory
- Work iteratively
    - less -> more correct
- Work incrementally
    - less -> more detail

Time

Every sophisticated model
includes at least one notion of time

Different notions of time

(not comprehensive list, nor are they mutually exclusive)

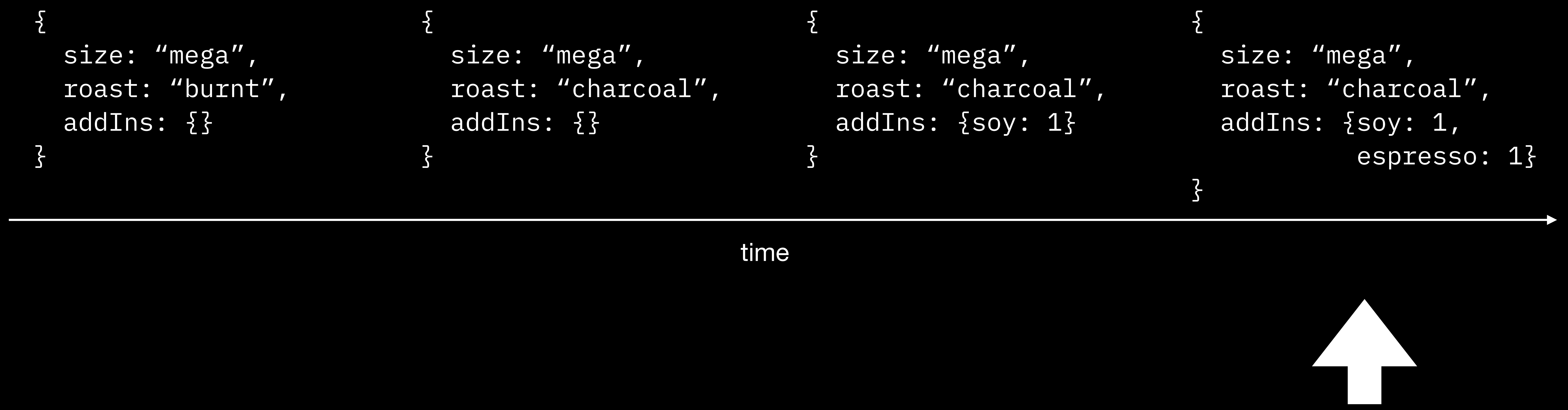
- Calendar date/time
- Order (x happens before y)
- As of
- History (audit)
- Future
- Counterfactual

The naive model updates in place (throwing away history)

Two models of history

- History of states
- History of mutations

History of states



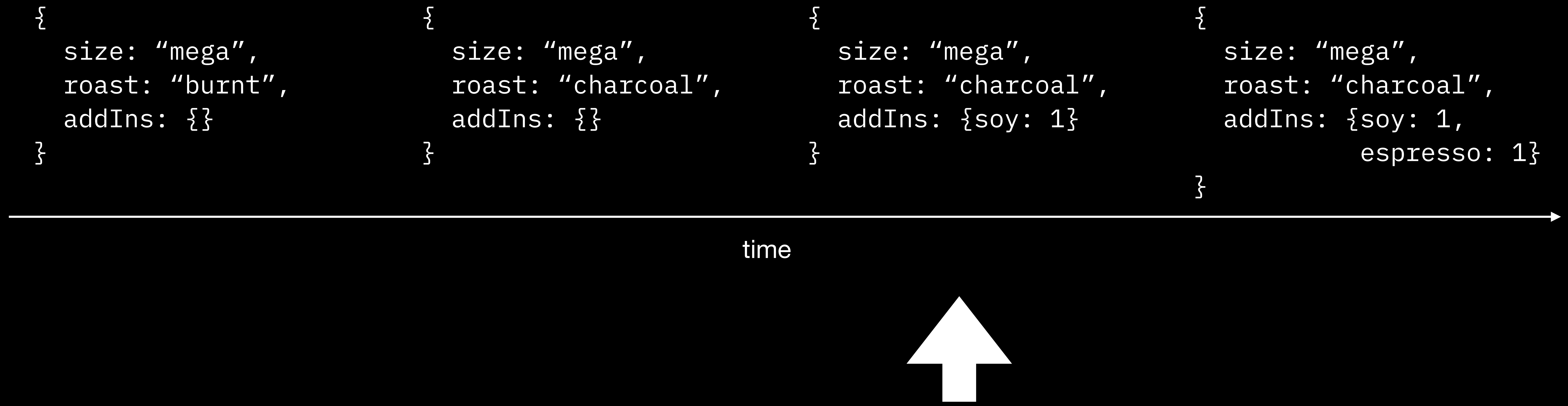
```
{
  size: “mega”,
  roast: “burnt”,
  addIns: {}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {soy: 1}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {soy: 1,
           espresso: 1}
}
```

History of states



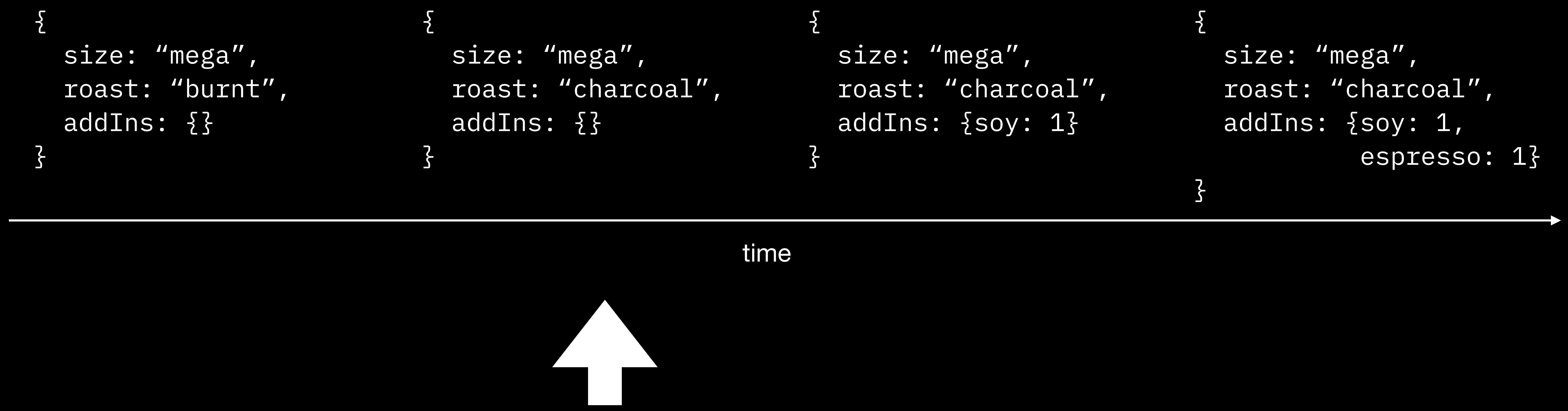
```
{
  size: “mega”,
  roast: “burnt”,
  addIns: {}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {soy: 1}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {soy: 1,
           espresso: 1}
}
```

History of states



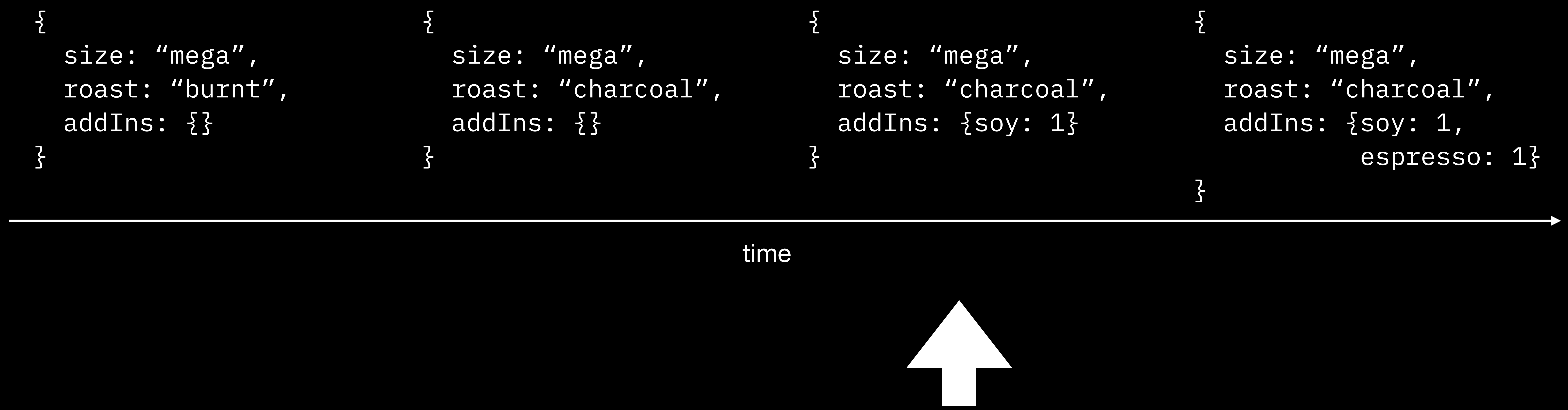
```
{
  size: “mega”,
  roast: “burnt”,
  addIns: {}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {soy: 1}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {soy: 1,
           espresso: 1}
}
```

History of states



```
{
  size: “mega”,
  roast: “burnt”,
  addIns: {}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {soy: 1}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {soy: 1,
           espresso: 1}
}
```

History of states

```
{
  size: “mega”,
  roast: “burnt”,
  addIns: {}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {soy: 1}
}
```

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {soy: 1,
           almond: 1}
}
```

time

History of mutations

```
{operation: “newDefaultCoffee”}
```

```
{operation: “setRoast”,
 roast: “charcoal”}
```

```
{operation: “addAddIn”,
 roast: “soy”}
```

```
{operation: “addAddIn”,
 roast: “espresso”}
```

time

```
{
  size: “mega”,
  roast: “charcoal”,
  addIns: {soy: 1,
           espresso: 1}
}
```

Comparing history models

History of states

- Easy to implement
- Expensive to store

History of mutations

- Hard to set up
- Cheaper to store
- Captures intentions

Domain

What is the problem we're trying to solve with coffees?

Represent a coffee.

Model how coffees behave.

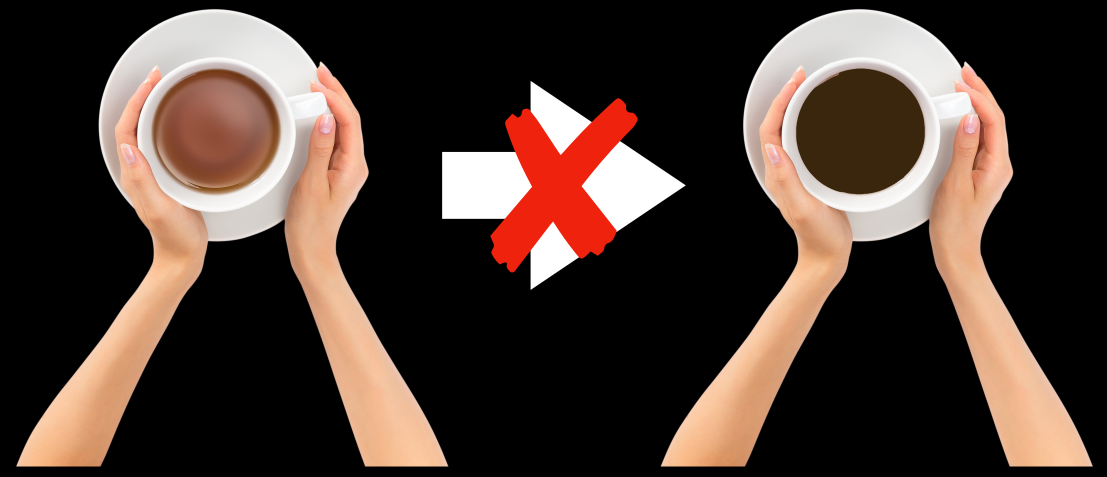
Turn nouns into classes and verbs into methods

Encode isA and hasA relationships.

Represent a coffee.

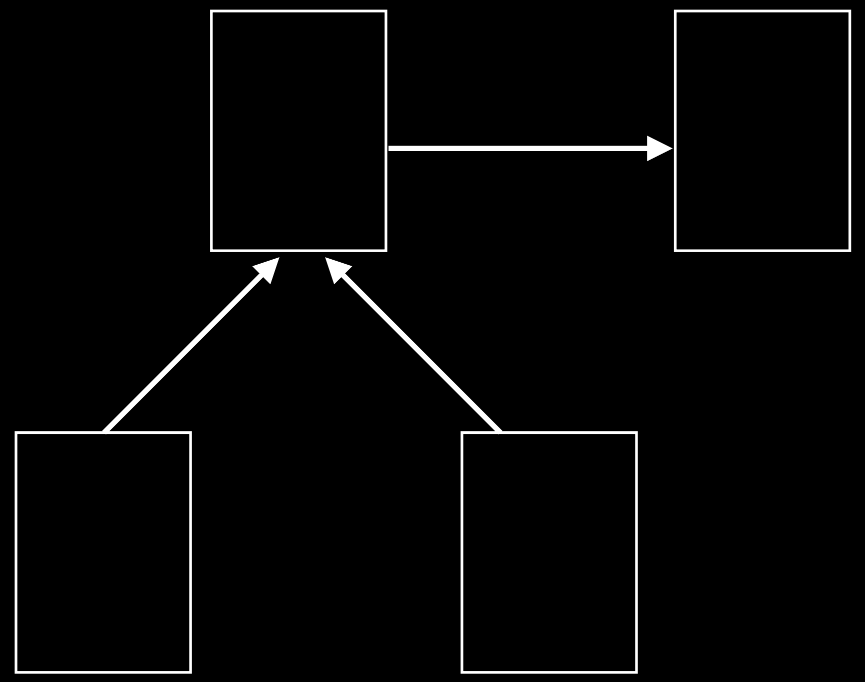
?

Model how coffees behave.

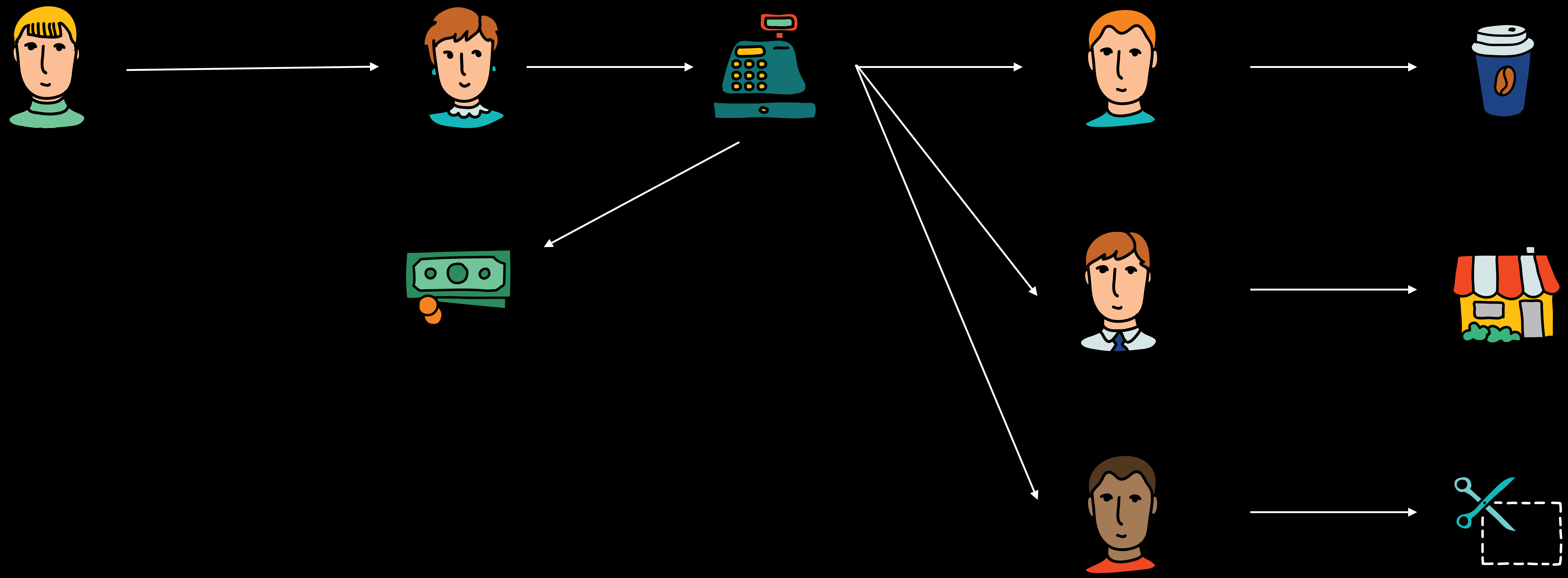


Nouns and verbs =>
Classes and methods

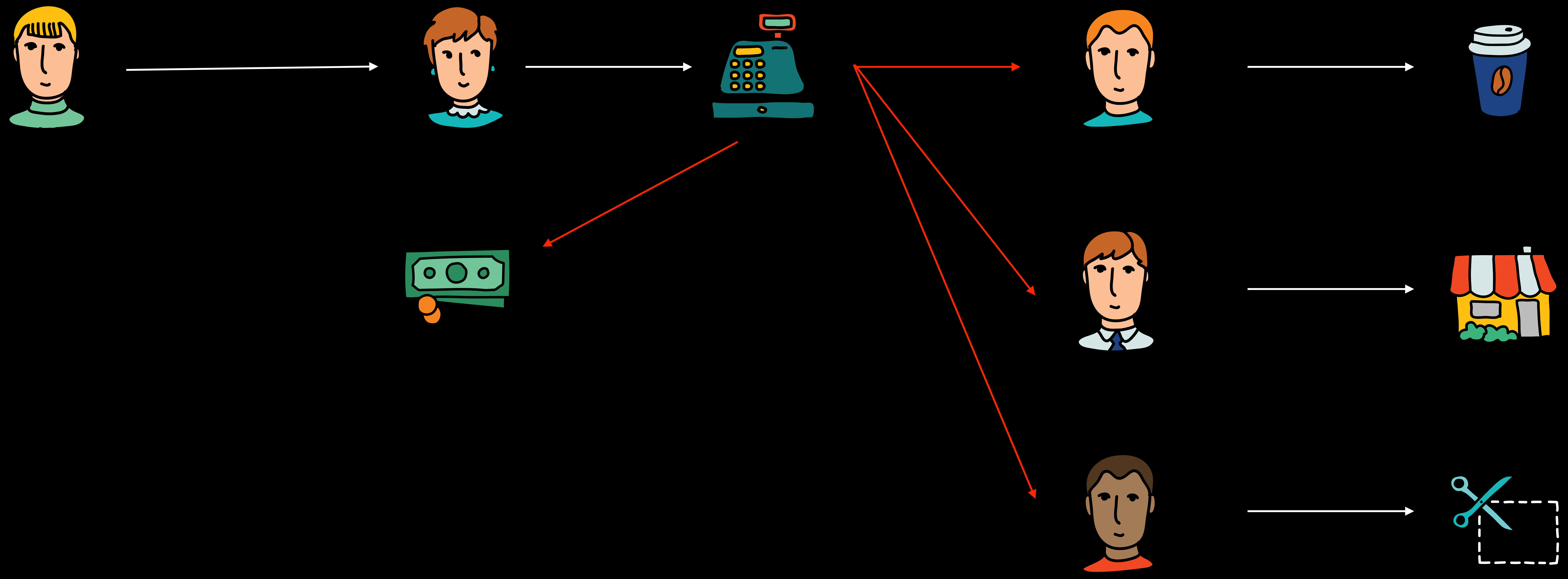
isA/hasA
Classic OODA



Where does the software fit in the process?



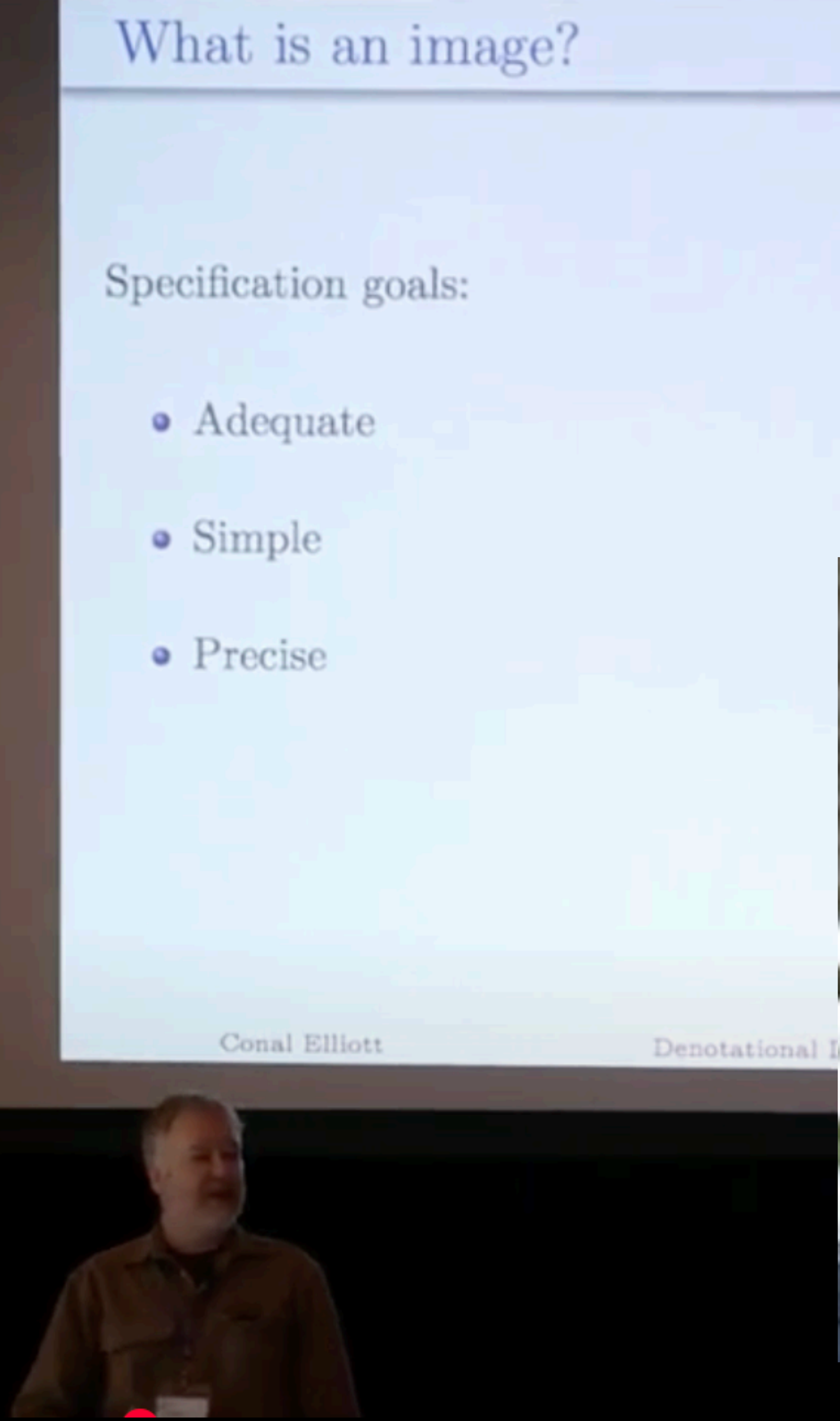
Where does the software fit in the process?



How would we do it with paper and pencil?

Zen out

What *is* a coffee order?



What is an image?

Specification goals:

- Adequate
- Simple
- Precise

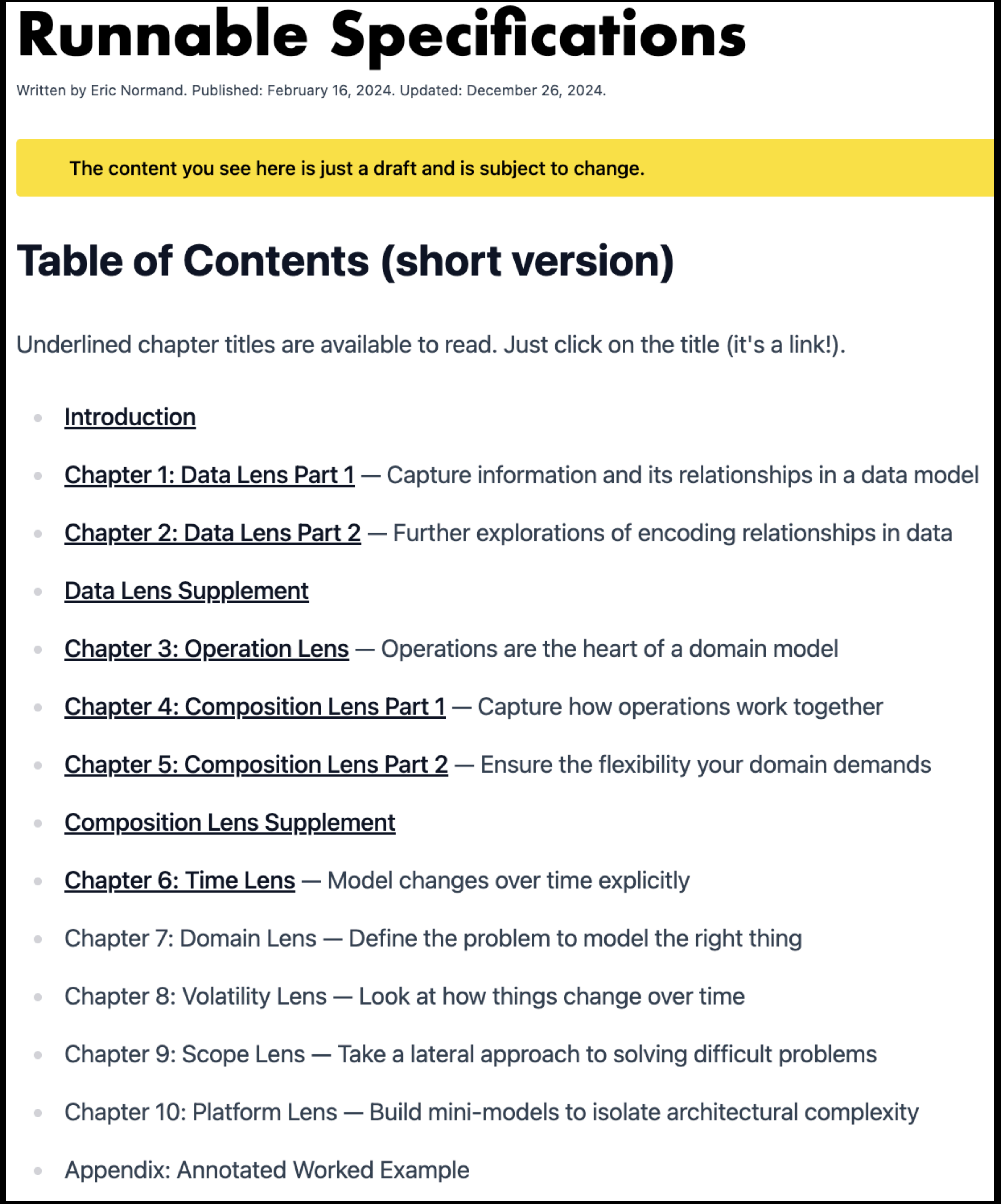
Conal Elliott

Denotational I

What *is* an image?

What is an image?

- A rectangular grid of RGBA values
- A collection of vector graphics elements
- An algorithm for drawing
- Mapping from 2D space to color
    - Point -> Color



Runnable Specifications

Written by Eric Normand. Published: February 16, 2024. Updated: December 26, 2024.

The content you see here is just a draft and is subject to change.

Table of Contents (short version)

Underlined chapter titles are available to read. Just click on the title (it's a link!).

- Introduction
- Chapter 1: Data Lens Part 1 — Capture information and its relationships in a data model
- Chapter 2: Data Lens Part 2 — Further explorations of encoding relationships in data
- Data Lens Supplement
- Chapter 3: Operation Lens — Operations are the heart of a domain model
- Chapter 4: Composition Lens Part 1 — Capture how operations work together
- Chapter 5: Composition Lens Part 2 — Ensure the flexibility your domain demands
- Composition Lens Supplement
- Chapter 6: Time Lens — Model changes over time explicitly
- Chapter 7: Domain Lens — Define the problem to model the right thing
- Chapter 8: Volatility Lens — Look at how things change over time
- Chapter 9: Scope Lens — Take a lateral approach to solving difficult problems
- Chapter 10: Platform Lens — Build mini-models to isolate architectural complexity
- Appendix: Annotated Worked Example

ericnormand.me/domain-modeling

Introduction +
6 Chapters +
2 Supplements =
250 pages

Newsletter

ericnormand.substack.com