Domain Modeling

How rich meaning improves your code

Eric Normand - Houston Functional Programmers - January 18, 2023

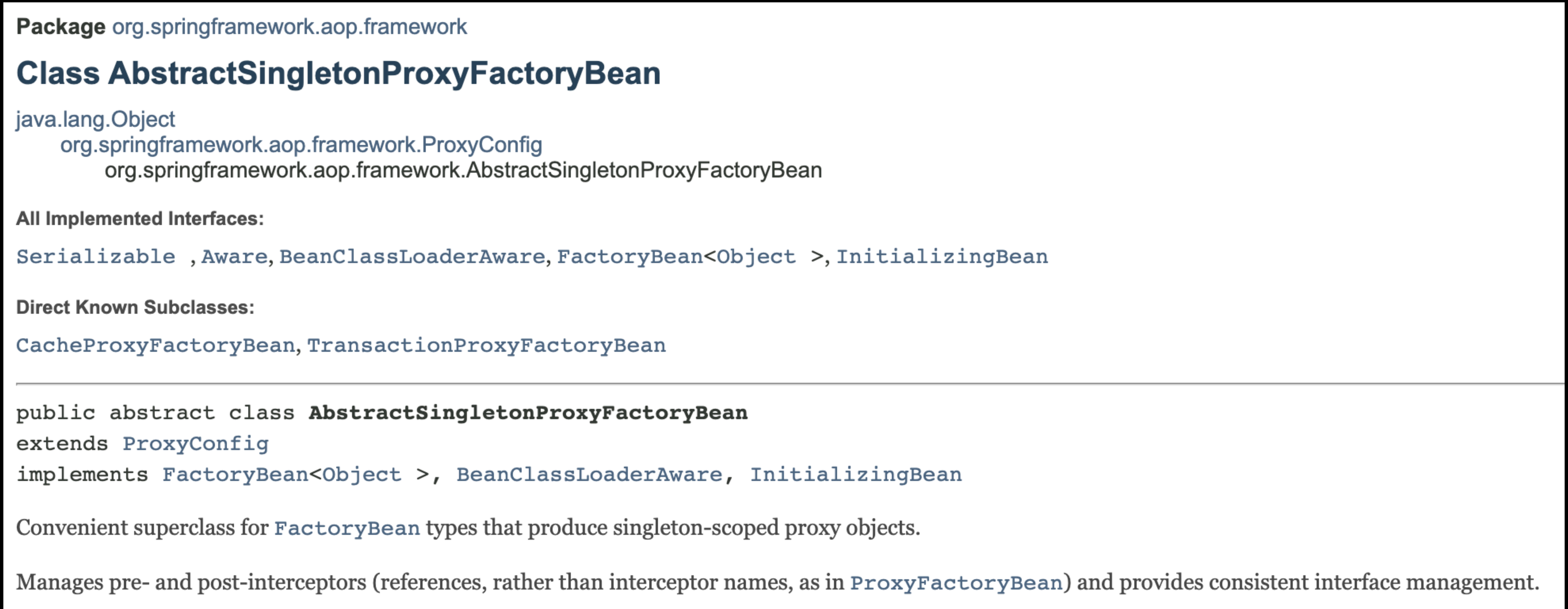
Software Design has failed

“I began to notice, by the late 70s, some weaknesses in our work with patterns and the pattern languages.

“By the late 70s, I had begun to see many buildings that were being made in the world when the patterns were applied. I was not happy with what I saw. It seemed to me that we had fallen far short of the mark that I had intended. But, I also realized that whatever was going wrong wasn't going to be corrected by writing a few more patterns or making the patterns a little bit better.”

Christopher Alexander

https://www.patternlanguage.com/archive/ieee.html



Package org.springframework.aop.framework

Class AbstractSingletonProxyFactoryBean

java.lang.Object
    org.springframework.aop.framework.ProxyConfig
        org.springframework.aop.framework.AbstractSingletonProxyFactoryBean

All Implemented Interfaces:

`Serializable` , `Aware`, `BeanClassLoaderAware`, `FactoryBean<Object >`, `InitializingBean`

Direct Known Subclasses:

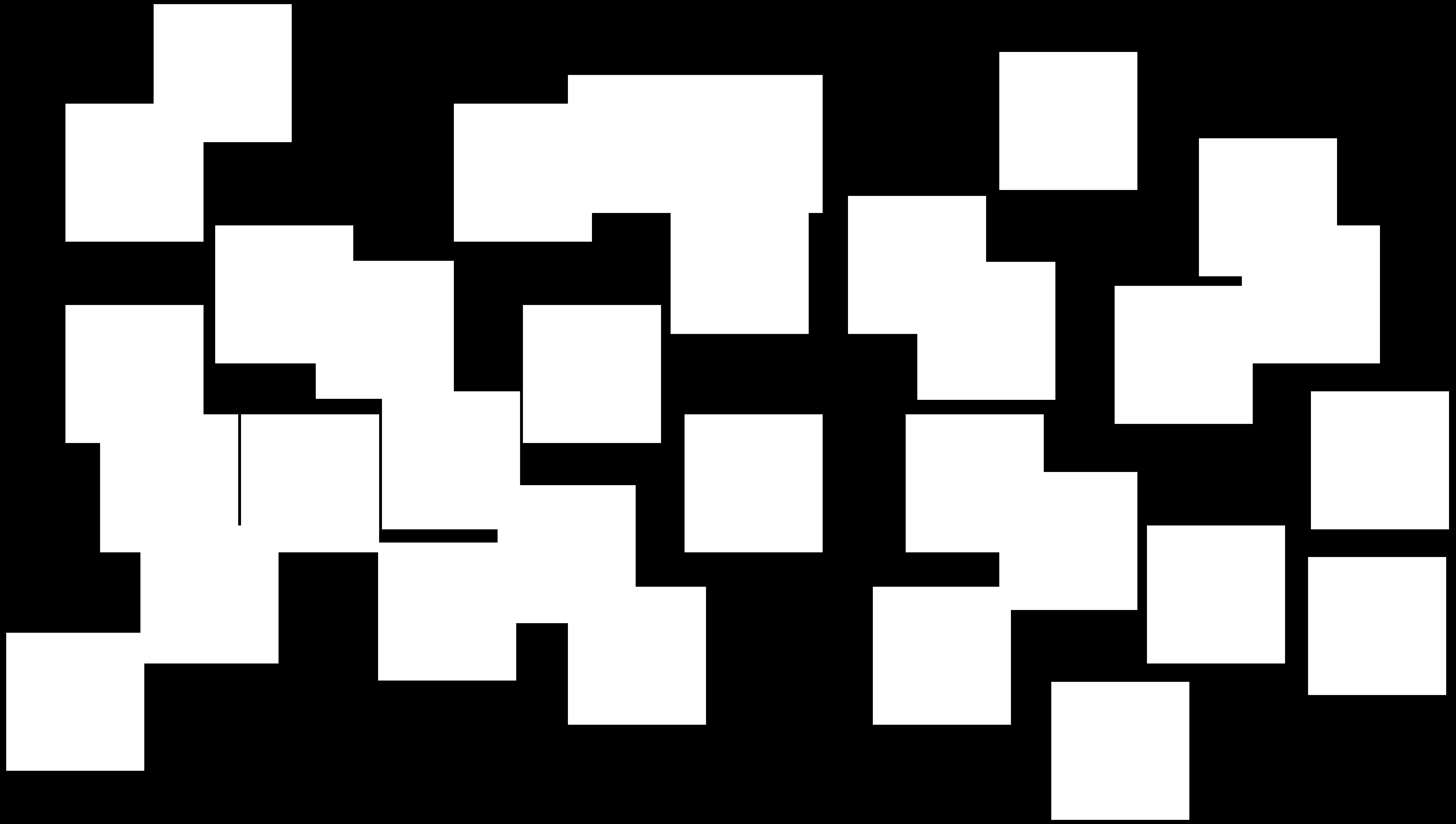
`CacheProxyFactoryBean`, `TransactionProxyFactoryBean`

```
public abstract class AbstractSingletonProxyFactoryBean
extends ProxyConfig
implements FactoryBean<Object >, BeanClassLoaderAware, InitializingBean
```

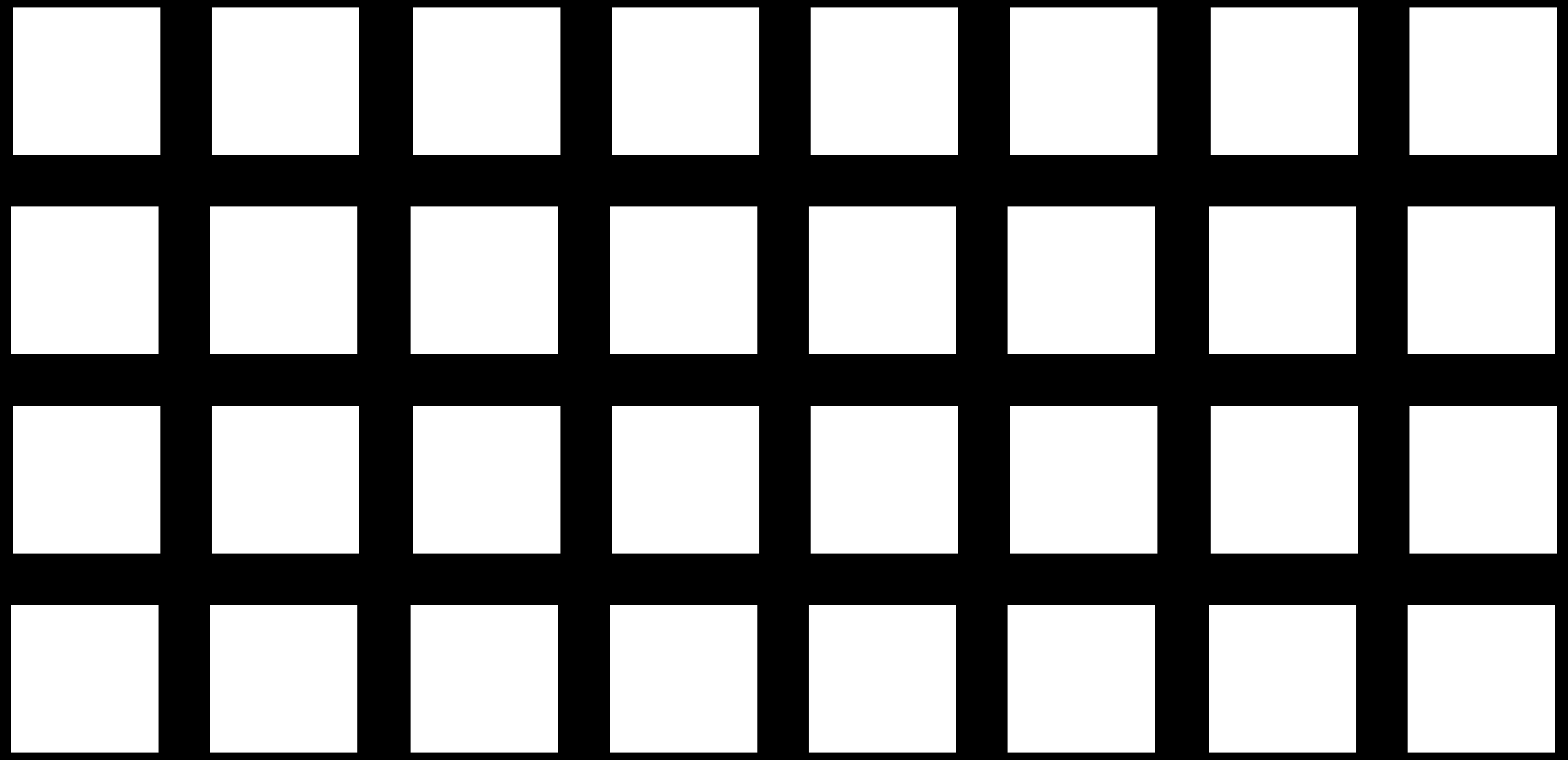
Convenient superclass for `FactoryBean` types that produce singleton-scoped proxy objects.

Manages pre- and post-interceptors (references, rather than interceptor names, as in `ProxyFactoryBean`) and provides consistent interface management.

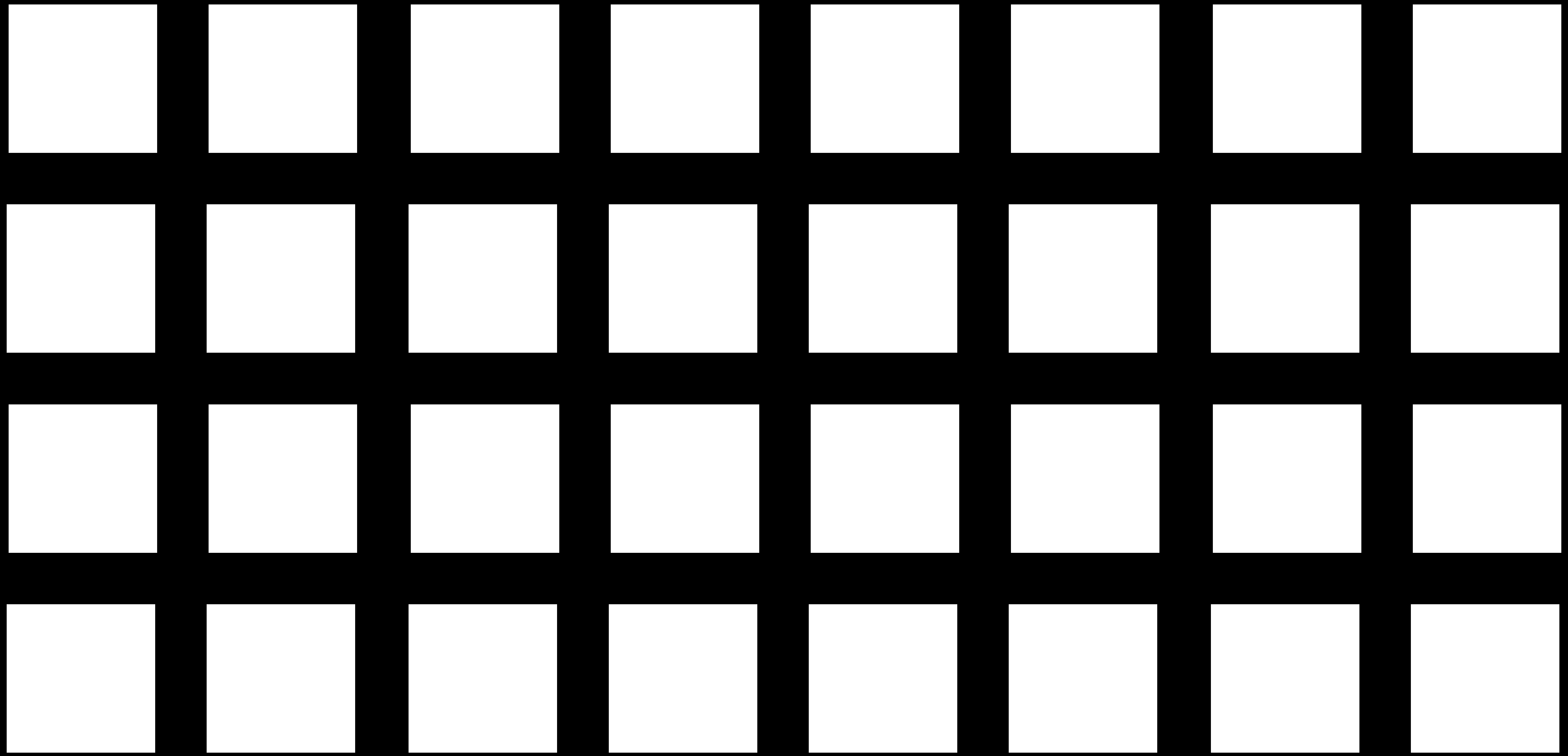
- Too much coupling
- Too many classes
- Code smells
- …



- Add indirection
- Use decorator pattern
- Refactor!



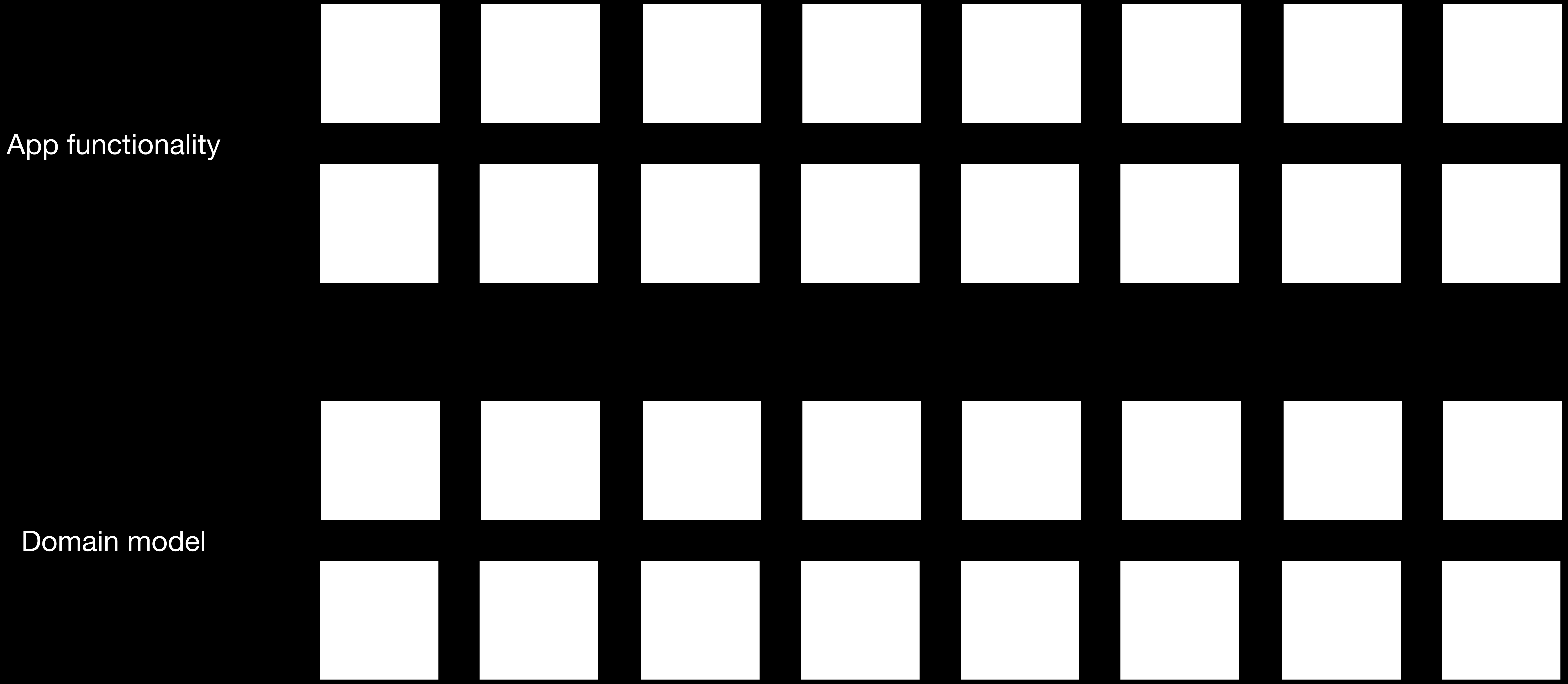
- Add indirection
- Use decorator pattern
- Refactor!



Sure, it’s less messy. But what about what it represents?

- Does the indirection correspond to anything in the real world?
- Does the decorator encode the possible states?
- Does the refactored code have a structure that better encodes the information about the world?

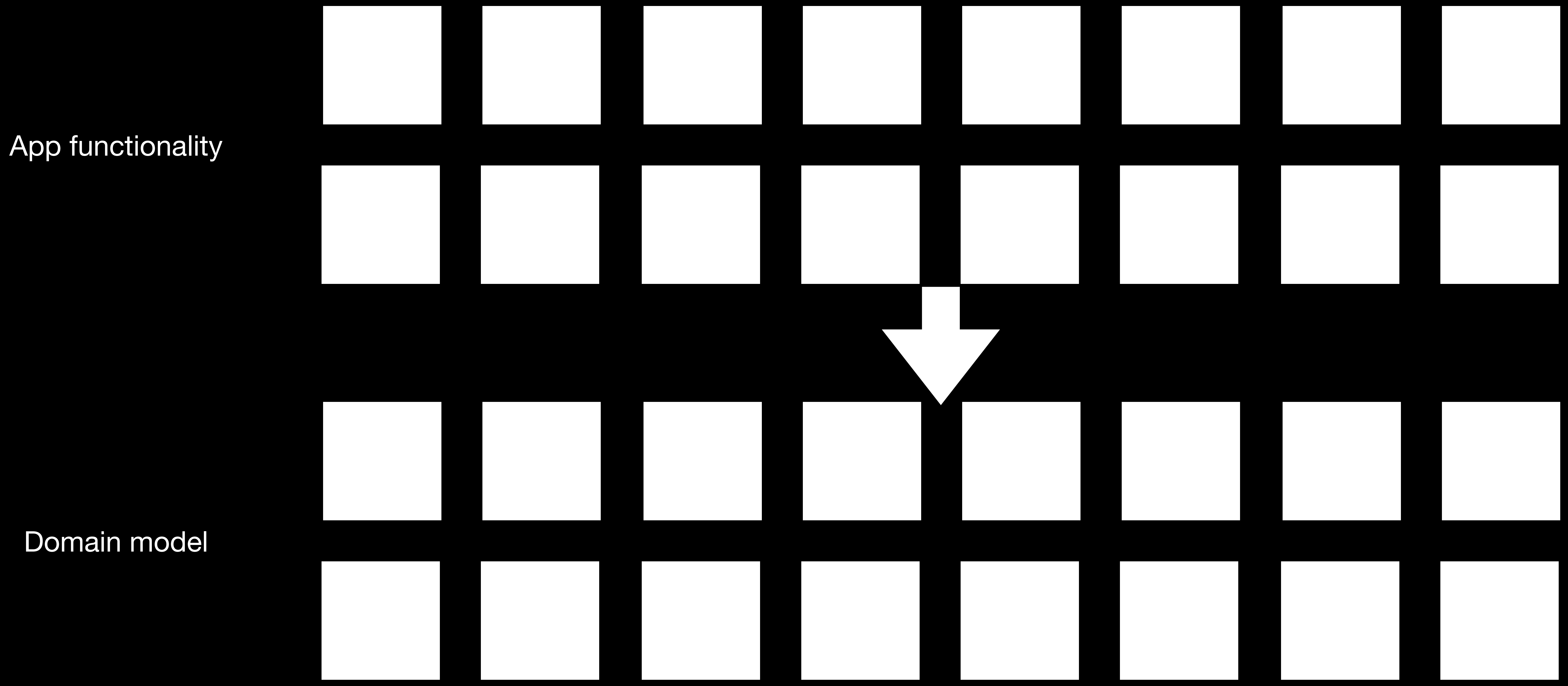
Domain modeling is a set of *skills and practices* we apply to encode our understanding of a domain separately from the software's explicit functionality.



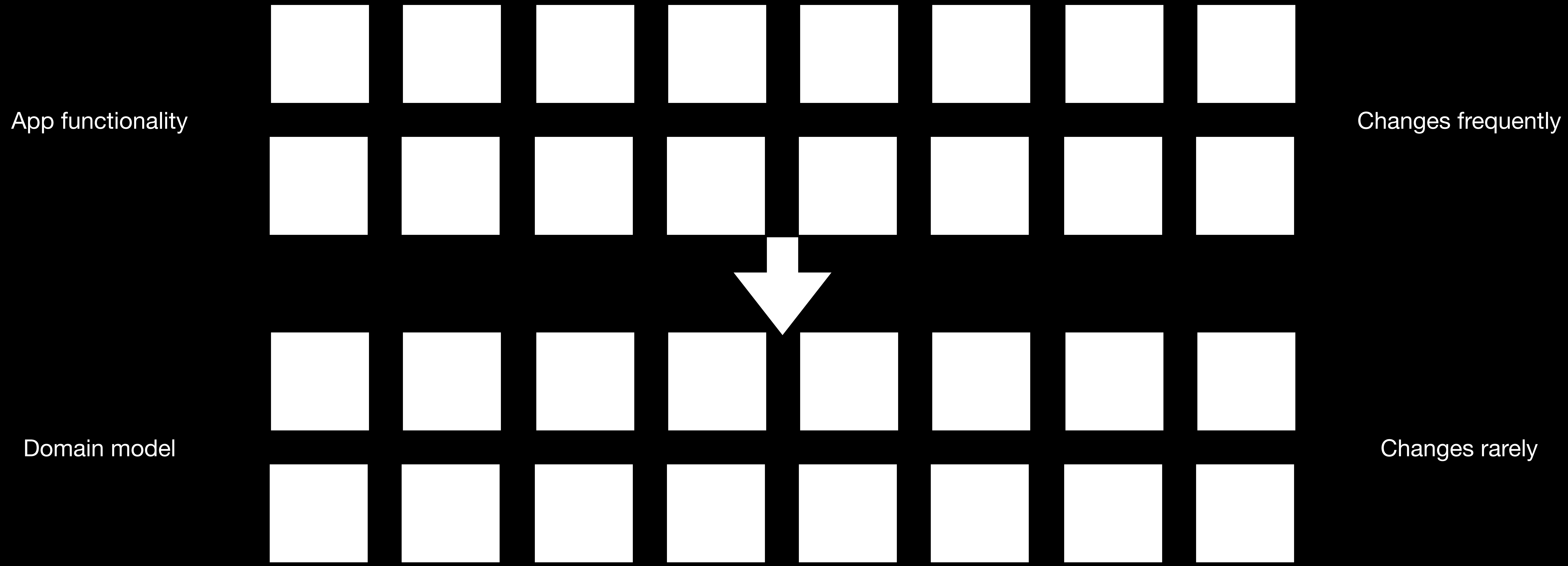
App functionality

Domain model

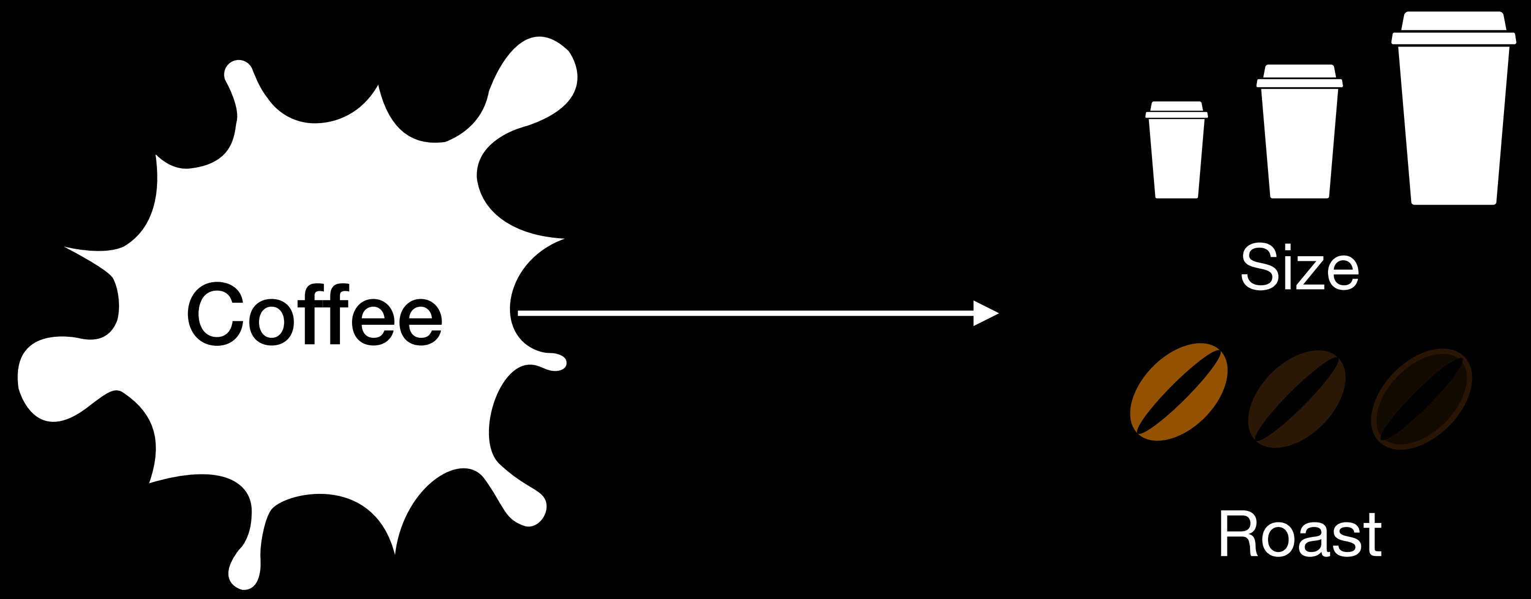
Domain modeling is a set of *skills and practices* we apply to encode our understanding of a domain separately from the software's explicit functionality.



Domain modeling is a set of *skills and practices* we apply to encode our understanding of a domain separately from the software's explicit functionality.



Teaching Challenges



Coffee

Size

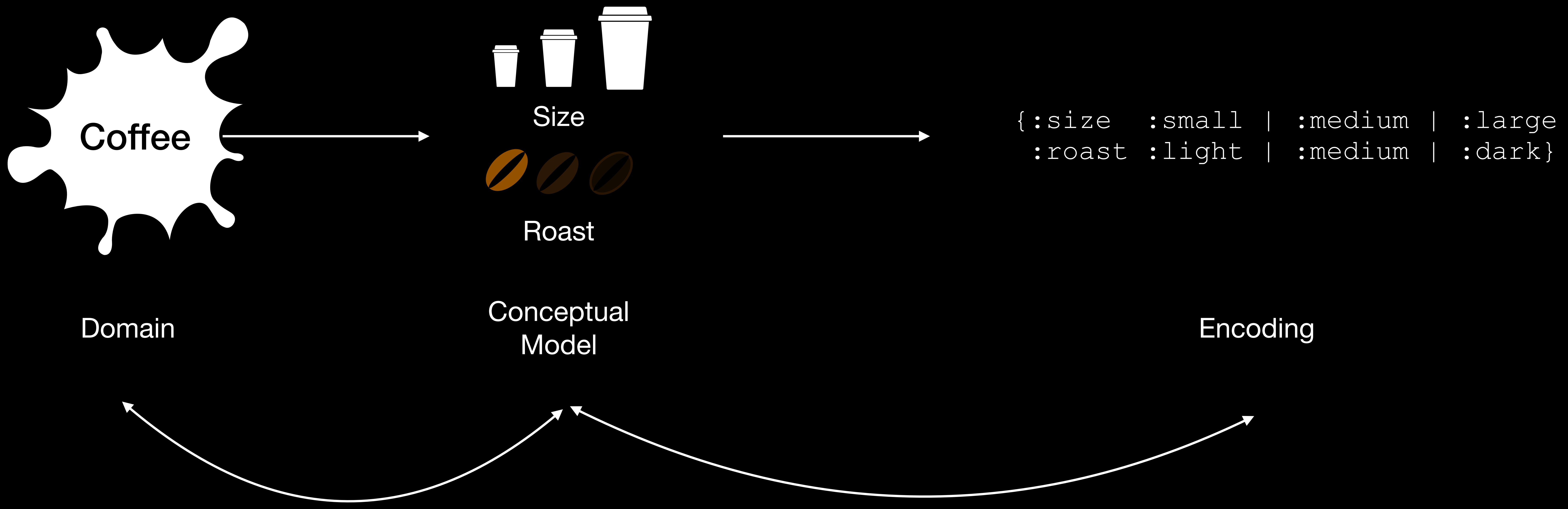
Roast

```
{:size  :small | :medium | :large
 :roast :light | :medium | :dark}
```

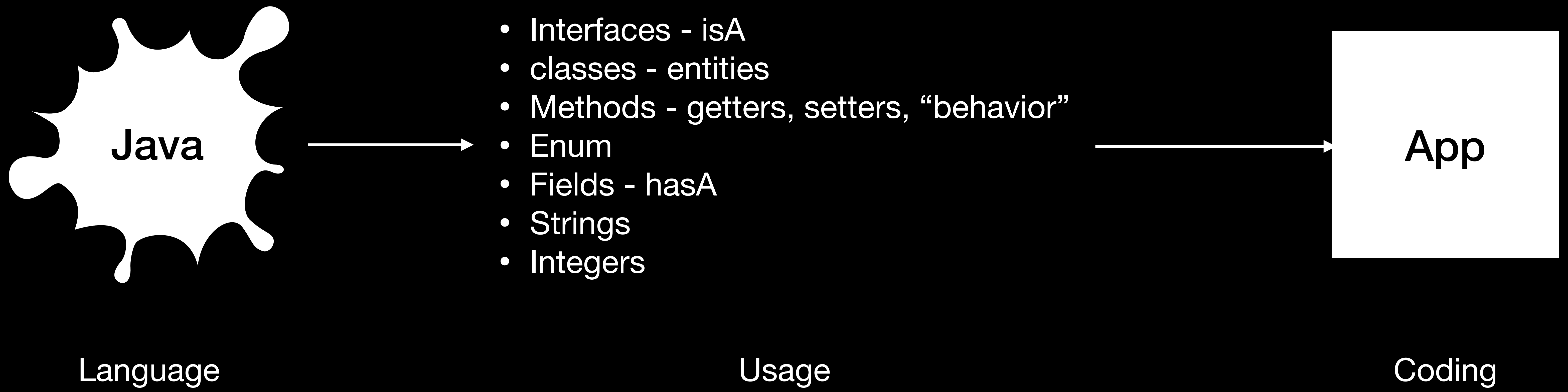
Domain

Conceptual Model

Encoding



```
{:size  :small | :medium | :large
 :roast :light | :medium | :dark}
```



Java

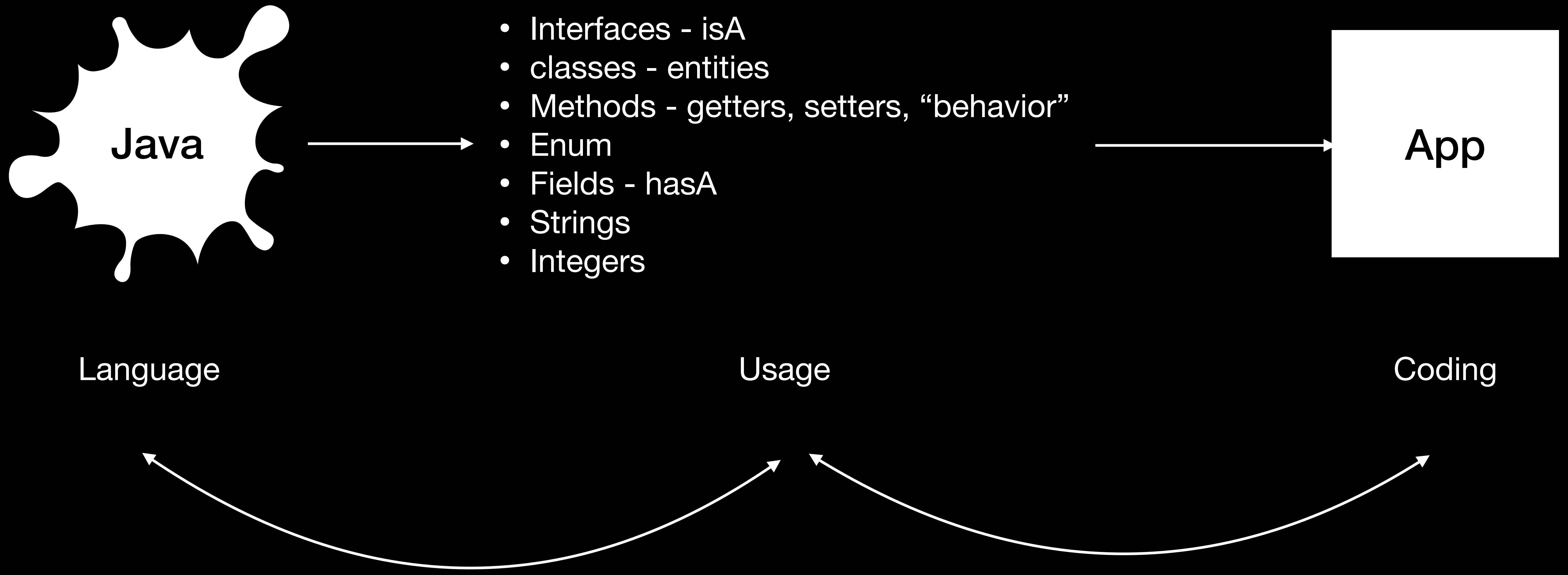
- Interfaces - isA
- classes - entities
- Methods - getters, setters, “behavior”
- Enum
- Fields - hasA
- Strings
- Integers

App

Language

Usage

Coding



Java

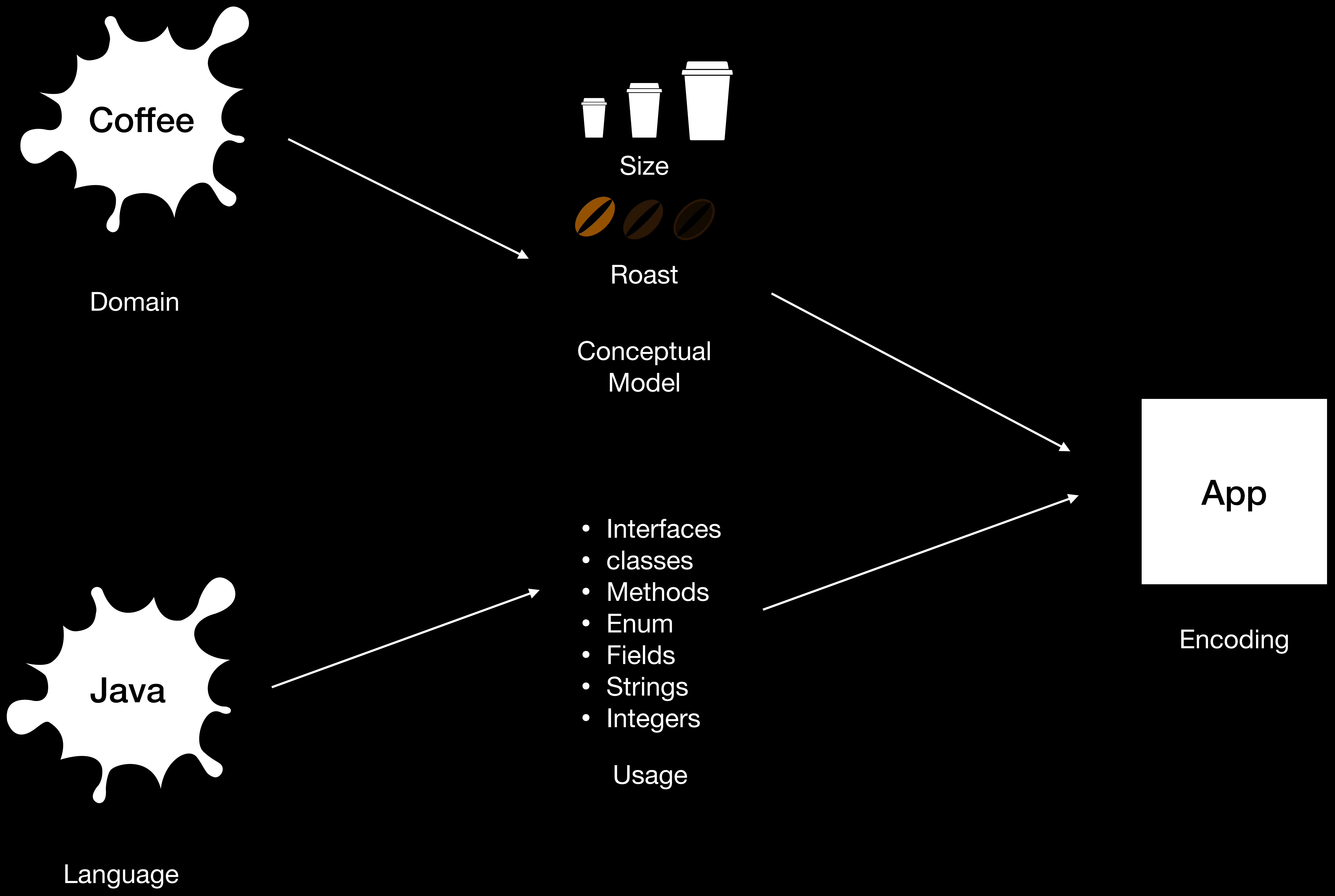
- Interfaces - isA
- classes - entities
- Methods - getters, setters, “behavior”
- Enum
- Fields - hasA
- Strings
- Integers

App

Language

Usage

Coding



Coffee

Domain

Size

Roast

Conceptual Model

Java

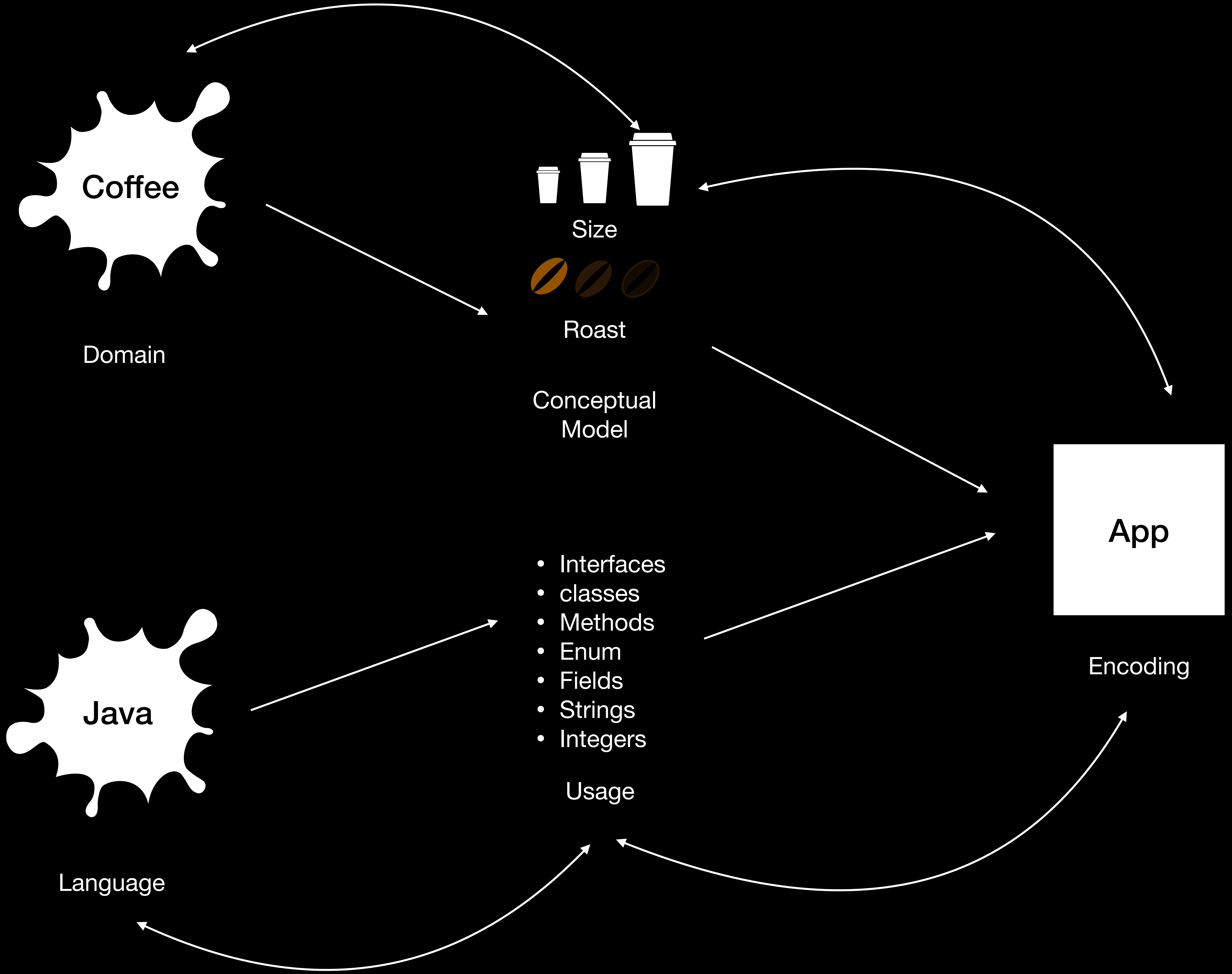
Language

- Interfaces
- classes
- Methods
- Enum
- Fields
- Strings
- Integers

Usage

App

Encoding



Coffee

Domain

Size

Roast

Conceptual Model

App

Encoding

Java

Language

- Interfaces
- classes
- Methods
- Enum
- Fields
- Strings
- Integers

Usage

Level 1: Data modeling

Goal: Encode and decode our conceptual model

Focus: Relationships among values

Level 2: Operation modeling

Goal: Support known use cases

Focus: Function signatures

Level 3: Algebraic modeling

Goal: Support unforeseen use cases

Focus: Composition of operations

Focus: Relationships among values

Level 1: Data modeling

Small

Medium

Large

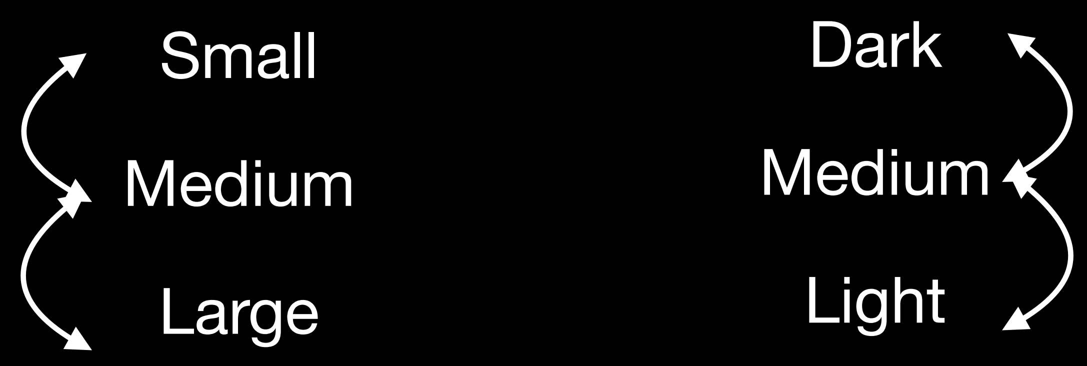
Dark

Medium

Light

Focus: Relationships among values

Level 1: Data modeling



Small

Medium

Large

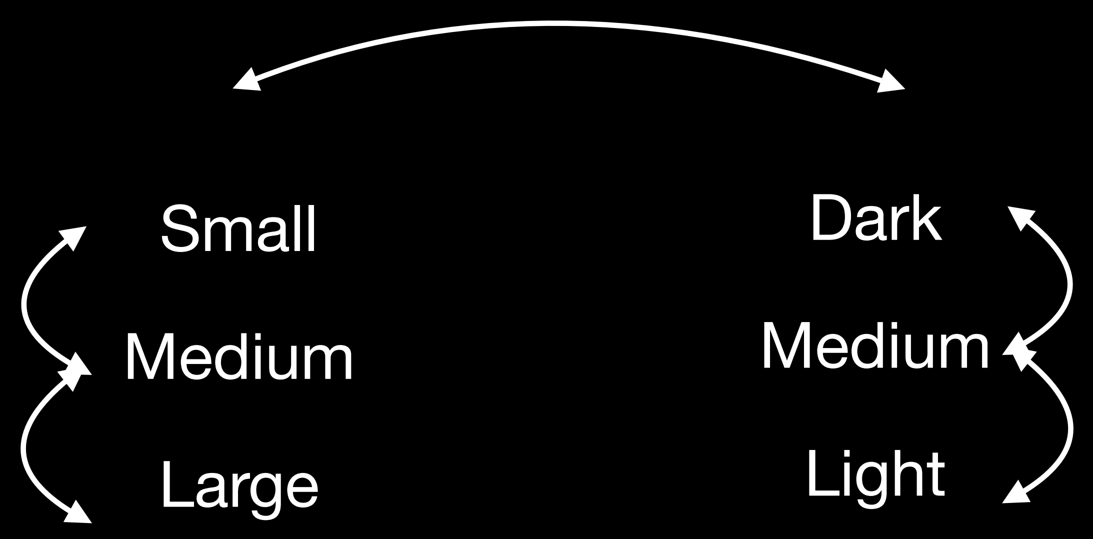
Dark

Medium

Light

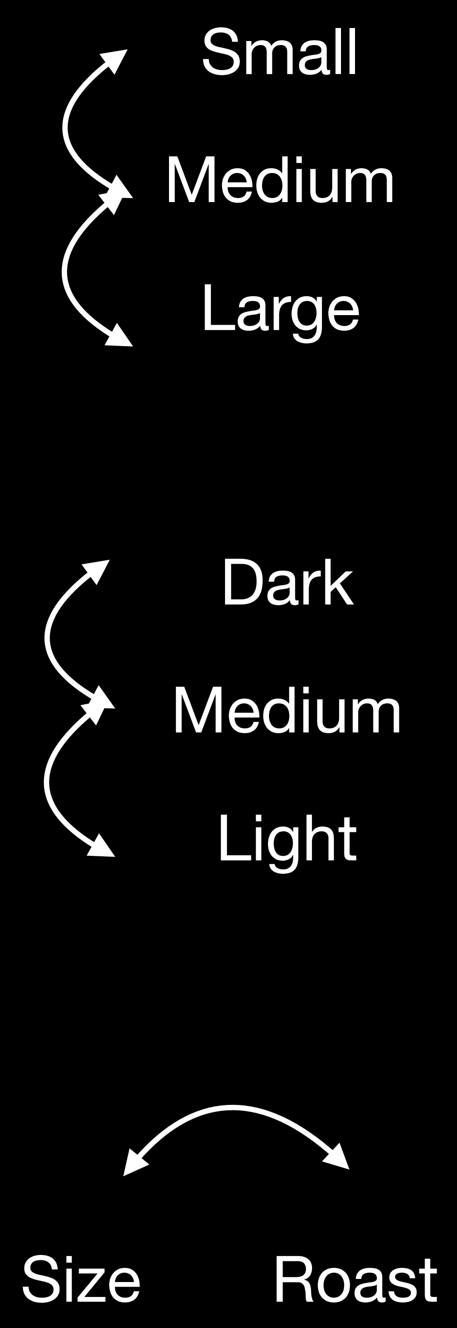
Focus: Relationships among values

Level 1: Data modeling



Focus: Relationships among values

Level 1: Data modeling



Small

Medium

Large

Dark

Medium

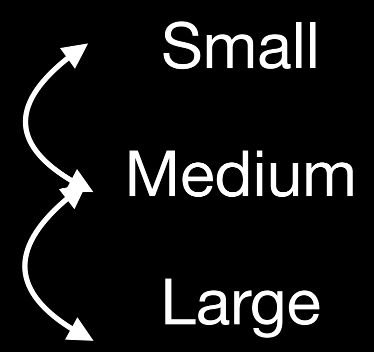
Light

Size

Roast

Focus: Relationships among values

Level 1: Data modeling

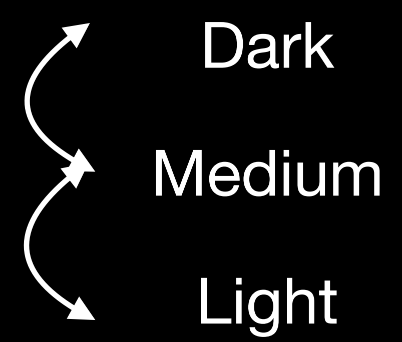


Small

Medium

Large

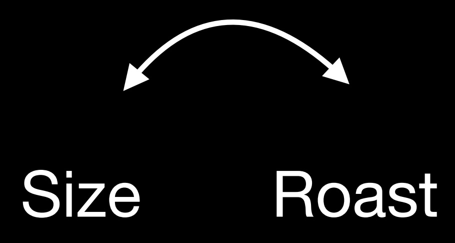
Choose one among many.



Dark

Medium

Light

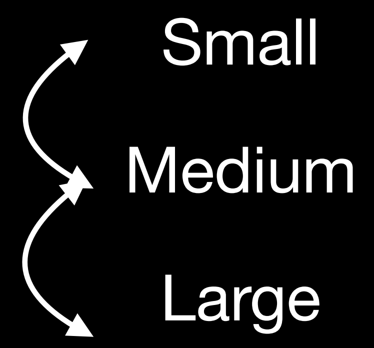


Size

Roast

Focus: Relationships among values

Level 1: Data modeling



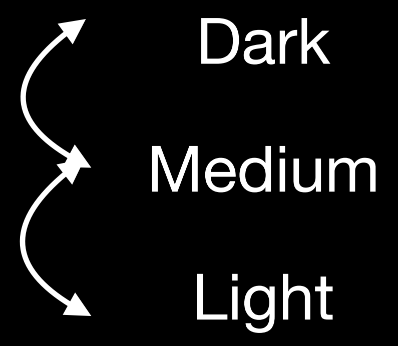
Small

Medium

Large

Choose one among many.

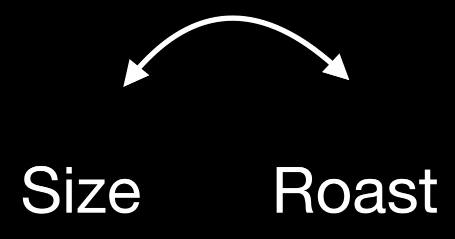
Small OR medium OR large



Dark

Medium

Light

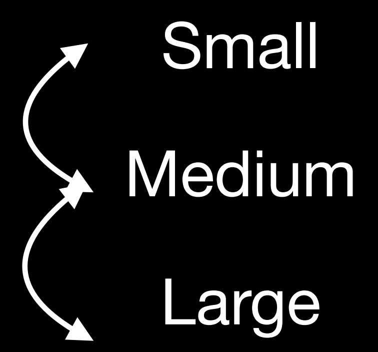


Size

Roast

Focus: Relationships among values

Level 1: Data modeling


Small

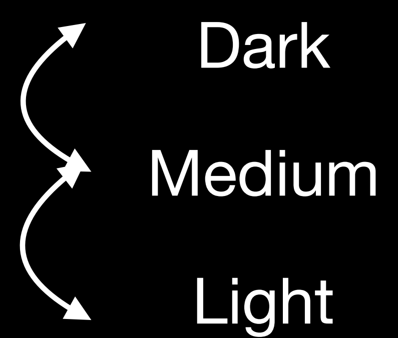
Medium

Large

Choose one among many.

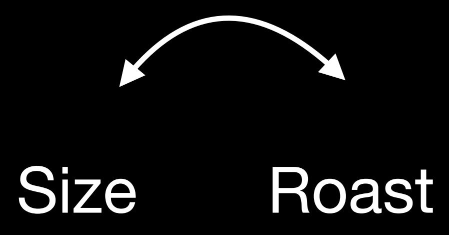
Small OR medium OR large

Alternative


Dark

Medium

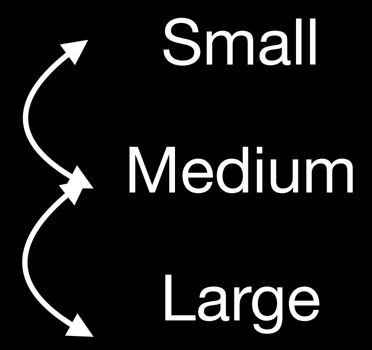
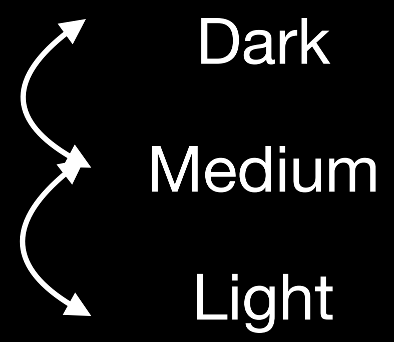
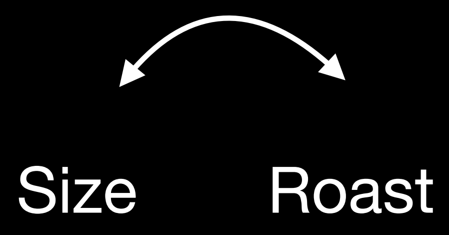
Light


Size

Roast

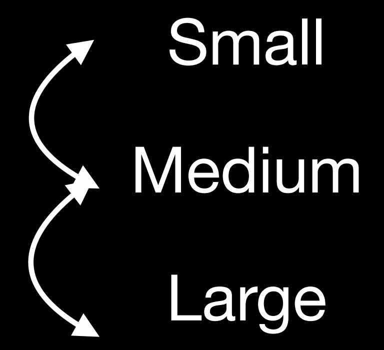
Focus: Relationships among values

Level 1: Data modeling

 Small Medium Large	Choose one among many.	Small OR medium OR large	Alternative
 Dark Medium Light	Choose one among many.	Dark OR medium OR light	Alternative
 Size Roast			

Focus: Relationships among values

Level 1: Data modeling



Small

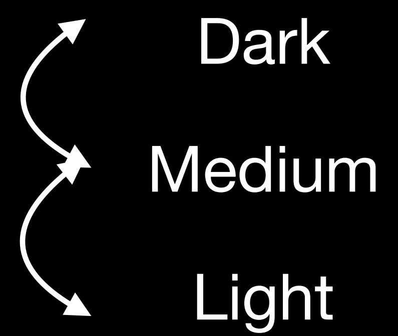
Medium

Large

Choose one among many.

Small OR medium OR large

Alternative



Dark

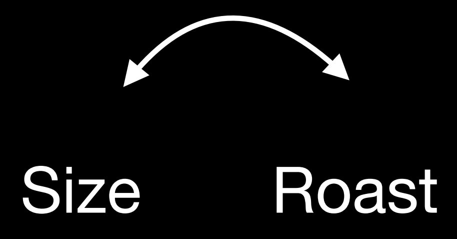
Medium

Light

Choose one among many.

Dark OR medium OR light

Alternative



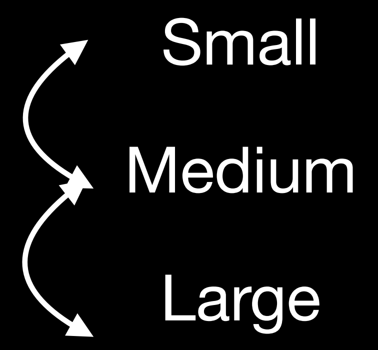
Size

Roast

Choose one of each.

Focus: Relationships among values

Level 1: Data modeling



Small

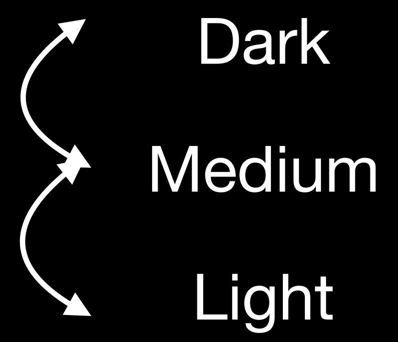
Medium

Large

Choose one among many.

Small OR medium OR large

Alternative



Dark

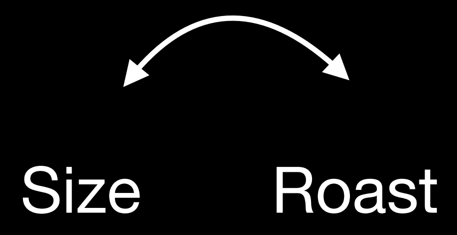
Medium

Light

Choose one among many.

Dark OR medium OR light

Alternative



Size

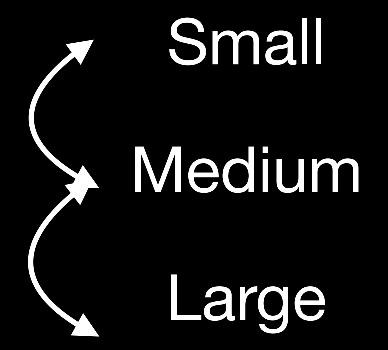
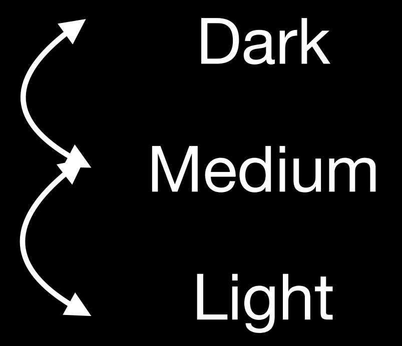
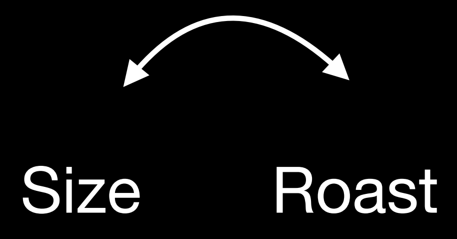
Roast

Choose one of each.

Size AND Roast

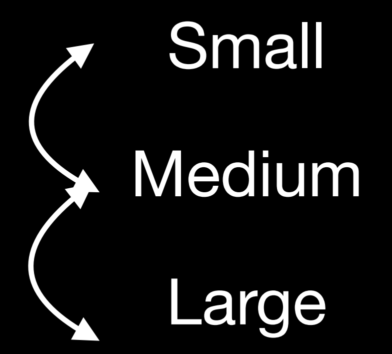
Focus: Relationships among values

Level 1: Data modeling

 Small Medium Large	Choose one among many.	Small OR medium OR large	Alternative
 Dark Medium Light	Choose one among many.	Dark OR medium OR light	Alternative
 Size Roast	Choose one of each.	Size AND Roast	Combination

Goal: Encode and decode our conceptual model

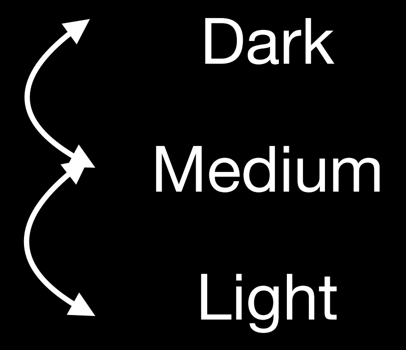
Level 1: Data modeling


Small

Medium

Large

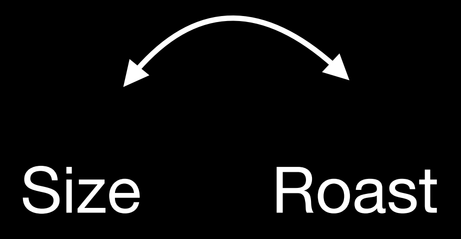
Alternative


Dark

Medium

Light

Alternative


Size Roast

Combination

Clojure

Keywords

Functions

Strings

Maps

Vectors

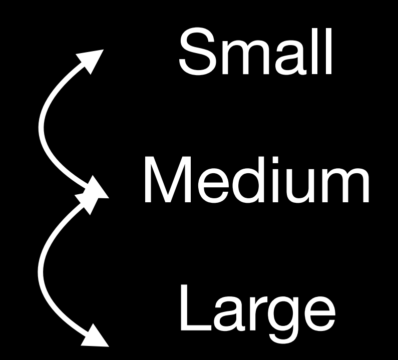
Protocols

Records

. . .

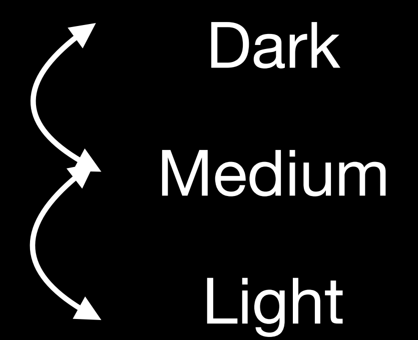
Goal: Encode and decode our conceptual model

Level 1: Data modeling

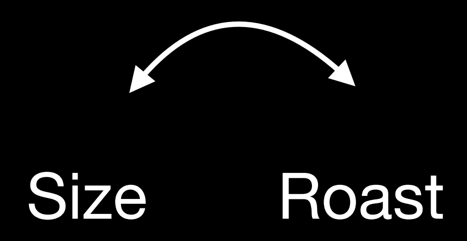


Alternative

```
:small, :medium, :large
```



Alternative



Combination

Clojure

← Keywords

Functions

Strings

Maps

Vectors

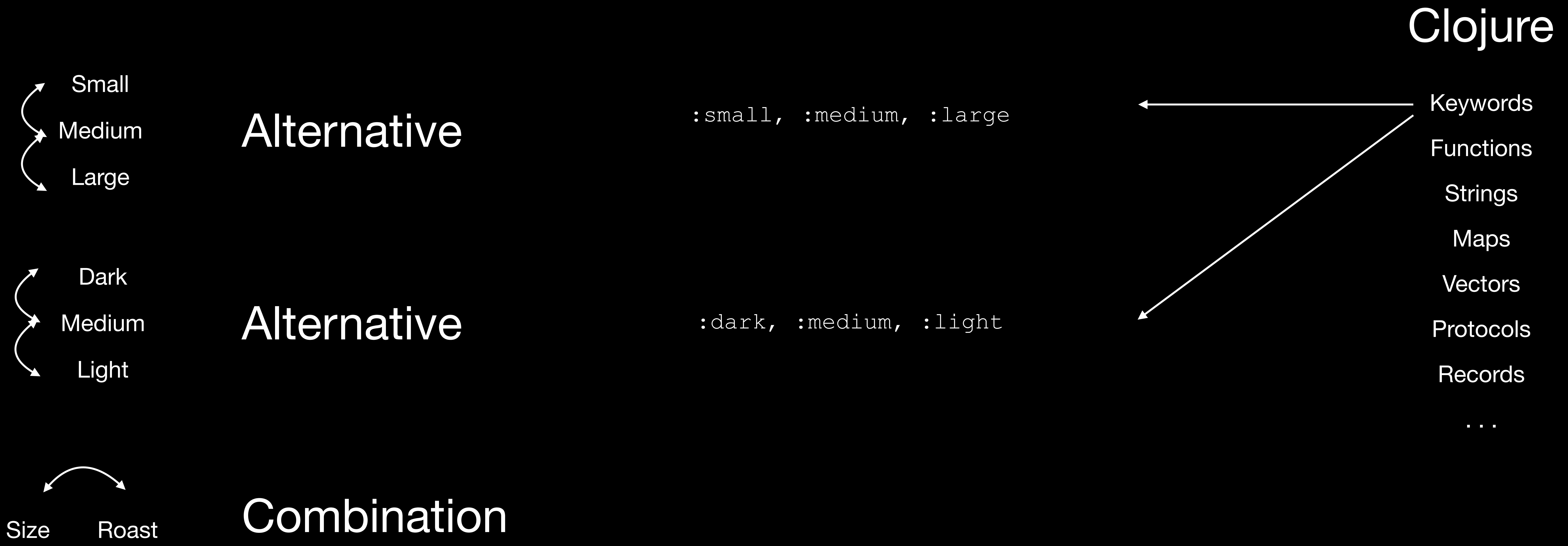
Protocols

Records

. . .

Goal: Encode and decode our conceptual model

Level 1: Data modeling



Small

Medium

Large

Alternative

```
:small, :medium, :large
```

Dark

Medium

Light

Alternative

```
:dark, :medium, :light
```

Size Roast

Combination

Clojure

Keywords

Functions

Strings

Maps

Vectors

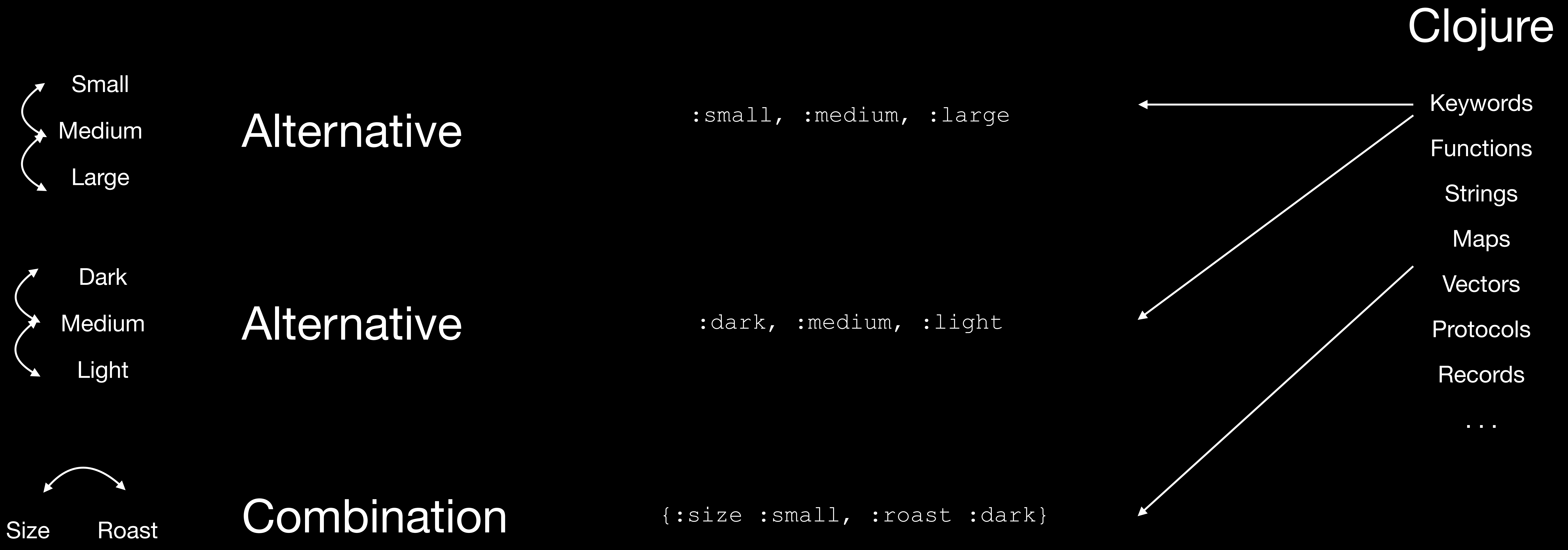
Protocols

Records

. . .

Goal: Encode and decode our conceptual model

Level 1: Data modeling



Clojure

Keywords

Functions

Strings

Maps

Vectors

Protocols

Records

. . .

Small

Medium

Large

Alternative

`:small, :medium, :large`

Dark

Medium

Light

Alternative

`:dark, :medium, :light`

Size

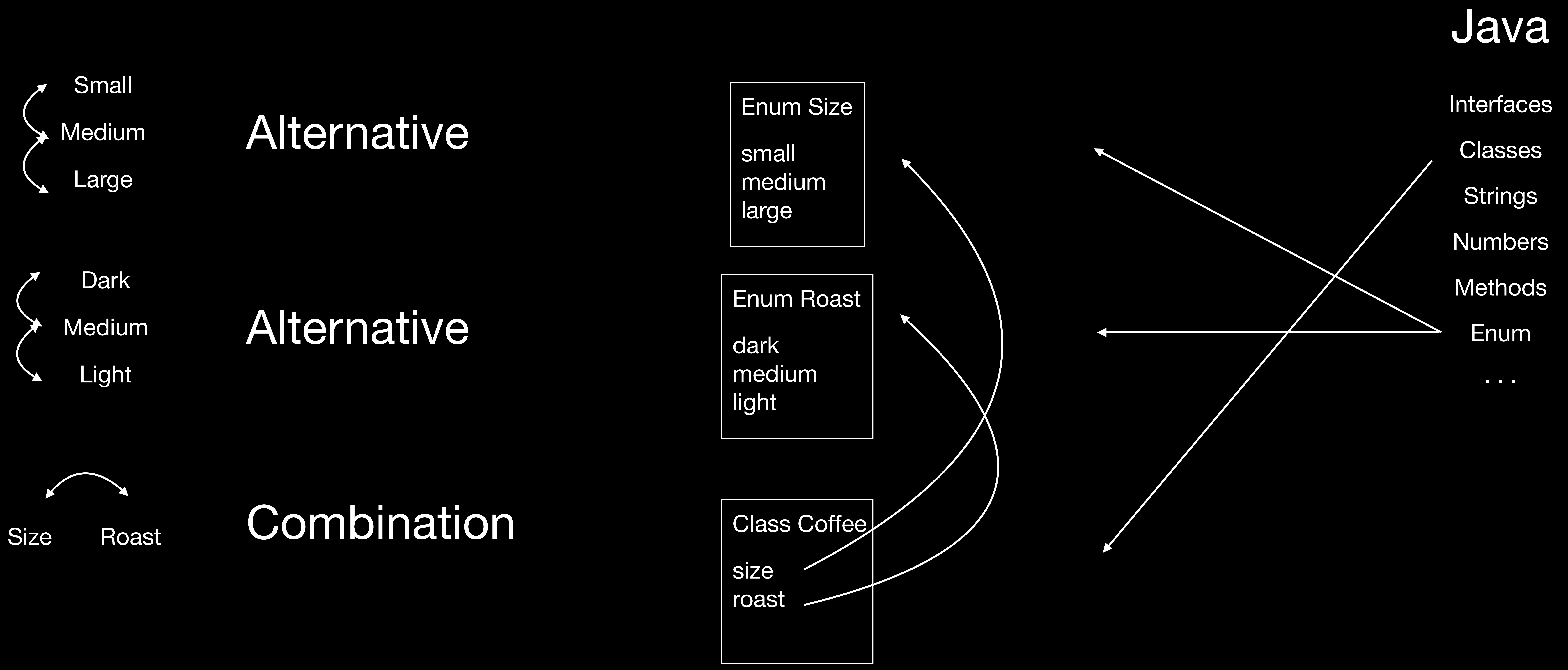
Roast

Combination

`{:size :small, :roast :dark}`

Goal: Encode and decode our conceptual model

Level 1: Data modeling



Small
Medium
Large

Alternative

Dark
Medium
Light

Alternative

Size
Roast

Combination

Enum Size

small
medium
large

Enum Roast

dark
medium
light

Class Coffee

size
roast

Java

Interfaces
Classes
Strings
Numbers
Methods
Enum
. . .

Data modeling elements

Atomic

- Identifier
- Count
- Measure
- Date
- Text

Composed

- Alternative
- Combination
- Collection
- Mapping
- Optional

Focus: Relationships among values

Level 1: Data modeling

Espresso shot

Almond

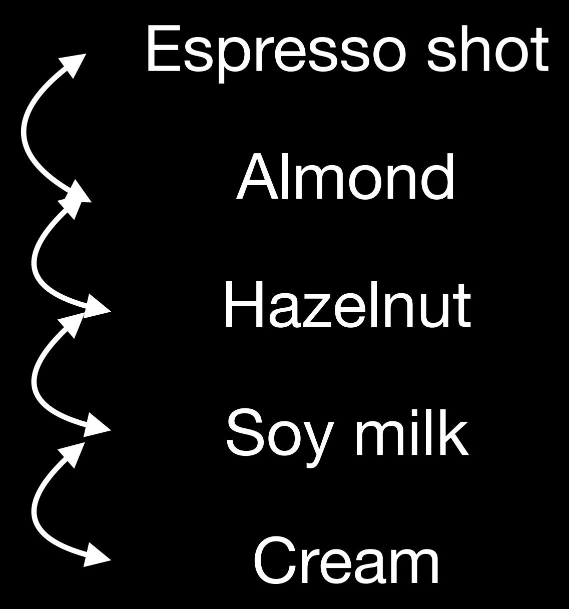
Hazelnut

Soy milk

Cream

Focus: Relationships among values

Level 1: Data modeling



Espresso shot

Almond

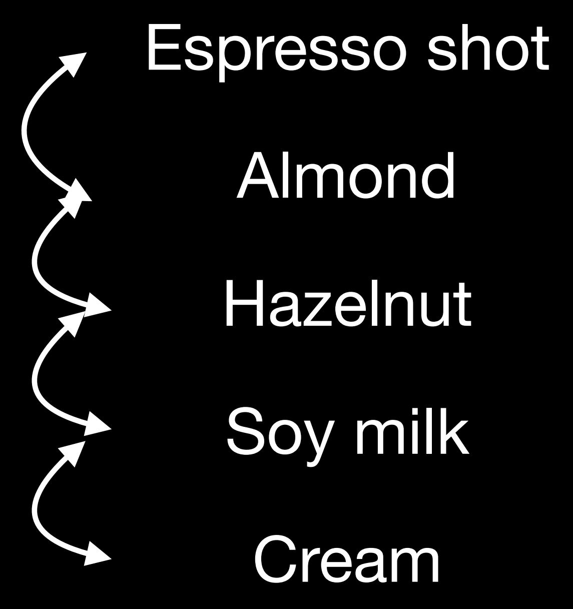
Hazelnut

Soy milk

Cream

Focus: Relationships among values

Level 1: Data modeling



Espresso shot

Almond

Hazelnut

Soy milk

Cream

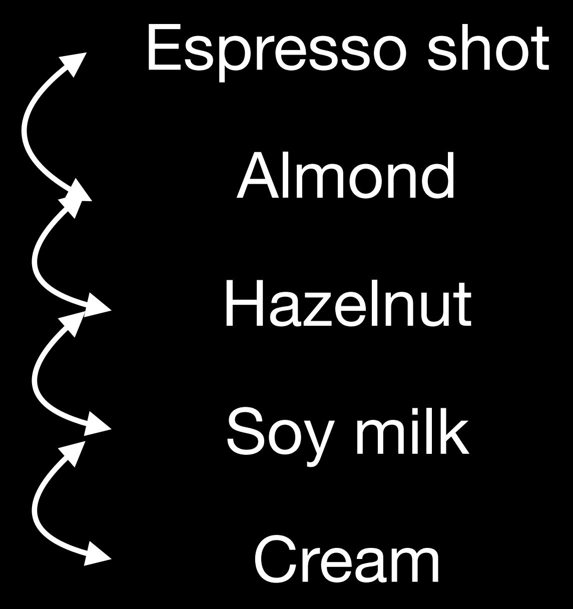
Choose one among many.

Small OR medium OR large

Alternative

Focus: Relationships among values

Level 1: Data modeling



Espresso shot

Almond

Hazelnut

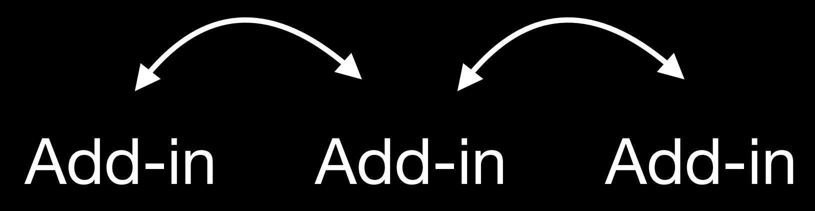
Soy milk

Cream

Choose one among many.

Small OR medium OR large

Alternative



Add-in Add-in Add-in

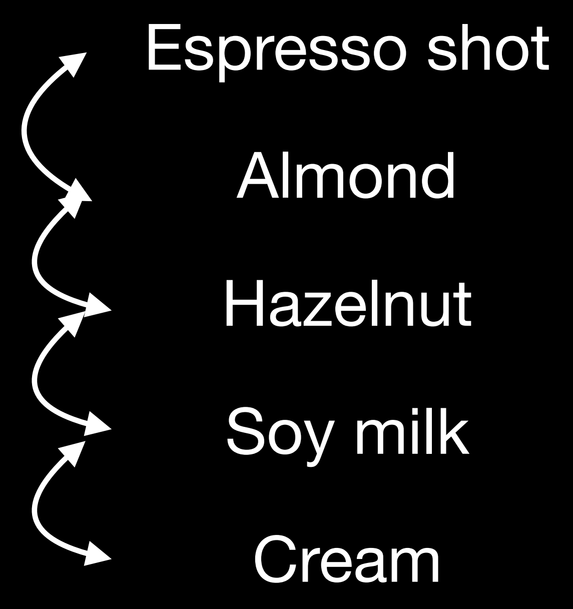
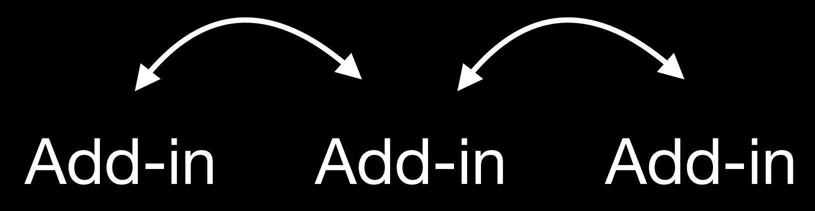
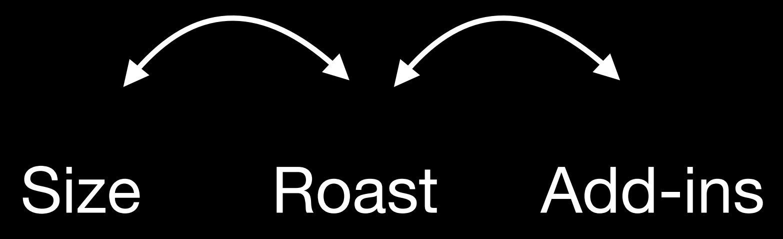
Choose 0-3.

Almond AND espresso

Collection

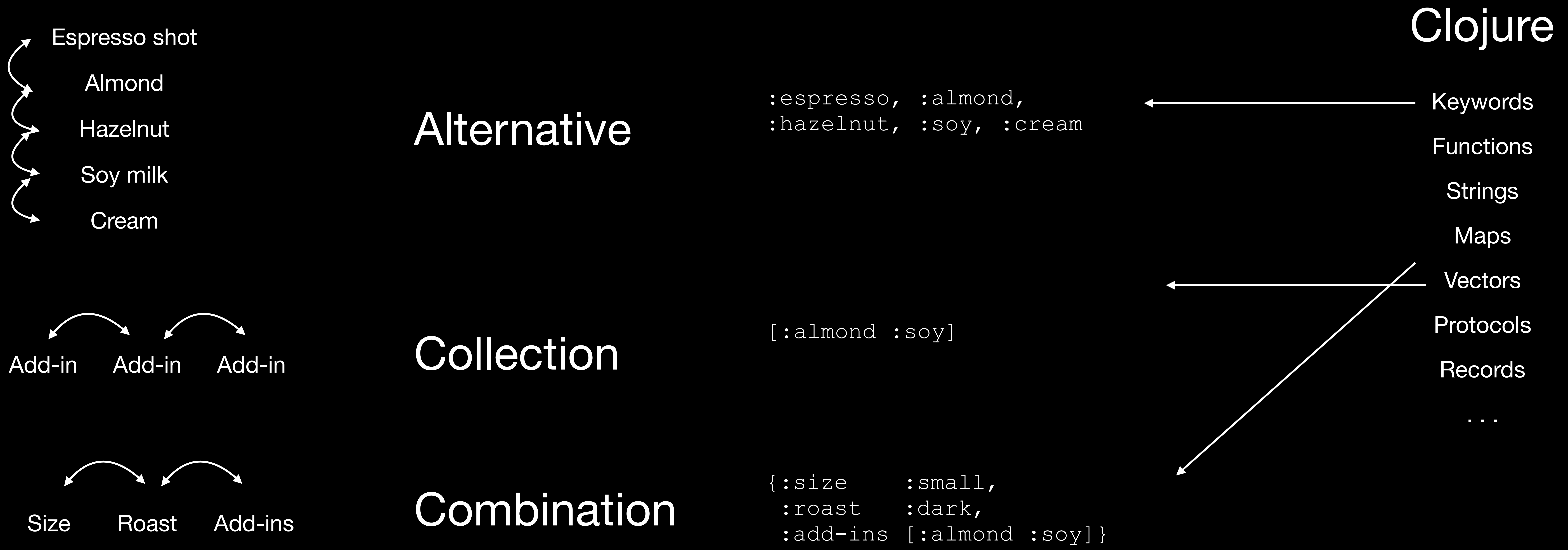
Focus: Relationships among values

Level 1: Data modeling

 Espresso shot Almond Hazelnut Soy milk Cream	Choose one among many.	Small OR medium OR large	Alternative
 Add-in Add-in Add-in	Choose 0-3.	Almond AND espresso	Collection
 Size Roast Add-ins	Choose one of each.	Size AND Roast	Combination

Focus: Relationships among values

Level 1: Data modeling



Espresso shot

Almond

Hazelnut

Soy milk

Cream

Alternative

```
:espresso, :almond,
:hazelnut, :soy, :cream
```

Add-in Add-in Add-in

Collection

```
[:almond :soy]
```

Size Roast Add-ins

Combination

```
{:size    :small,
 :roast   :dark,
 :add-ins [:almond :soy]}
```

Clojure

Keywords

Functions

Strings

Maps

Vectors

Protocols

Records

. . .

Espresso shot

Almond

Hazelnut

Soy milk

Cream

5 Add-ins

How many combinations of add-ins do we have? (up to 3)

Espresso shot

Almond

Hazelnut

Soy milk

Cream

5 Add-ins

How many combinations of add-ins do we have?
(up to 3)

[ ]

1

Espresso shot

Almond

Hazelnut

Soy milk

Cream

5 Add-ins

How many combinations of add-ins do we have?
(up to 3)

`[]`	1
`[a]`	5

Espresso shot

Almond

Hazelnut

Soy milk

Cream

5 Add-ins

How many combinations of add-ins do we have?
(up to 3)

`[]`	1
`[a]`	5
`[a b]`	5x5=25

Espresso shot
Almond
Hazelnut
Soy milk
Cream

5 Add-ins

How many combinations of add-ins do we have?
(up to 3)

[]	1
[a]	5
[a b]	5x5=25
[a b c]	5x5x5=125

Espresso shot

Almond

Hazelnut

Soy milk

Cream

5 Add-ins

How many combinations of add-ins do we have?
(up to 3)

`[]`	1
`[a]`	5
`[a b]`	5x5=25
`[a b c]`	5x5x5=125

156 combinations

156 combinations

but we counted some twice (or thrice)

```
[:almond :soy]        [:soy :almond]
```

```
[:almond :soy :soy]   [:soy :almond :soy]   [:soy :soy :almond]
```

Only 56 unique combinations

What do we do?

- Live with it

```
[:soy :almond :soy]
```

- Find a new representation

```
#OrderedList [:almond :soy :soy]
```

- Change the conceptual model
  - Ex: Collection => Mapping of identifiers to counts

```
{:soy 2 :almond 1}
```

- Revisit the domain
  - Ex: No duplicates allowed

```
#{:almond :soy}
```

Class Coffee

Int size

Int roast

Int espresso
Int soy
Int almond
Int hazelnut
Int cream

- Number of states in encoding vs in conceptual model vs in reality
- Complexity of normalize function
- Complexity of validate function

Focus: Functional Signatures

Level 2: Operation model

Are two coffees equal?

```
(defn coffee= [coffee-a coffee-b]) ;=> boolean
```

How many espresso shots does a coffee have?

```
(defn how-many? [coffee add-in]) ;=> natural-number
```

Maximum number of add-ins.

```
(defn within-limit? [coffee min max]) ;=> boolean
```

Add add-in

```
(defn add [coffee add-in]) ;=> coffee
```

Remove add-in

```
(defn remove [coffee add-in]) ;=> coffee
```

Vector

```
{:size :small :roast :dark
 :add-ins [:soy :almond :soy]}
```

Map

```
{:size :small :roast :dark
 :add-ins {:soy 2 :almond 1}}
```

Vector

```
{:size :small :roast :dark
 :add-ins [:soy :almond :soy]}
```

```
(defn coffee= [coffee-a coffee-b] ;=> boolean
  (= coffee-a coffee-b))
```

Map

```
{:size :small :roast :dark
 :add-ins {:soy 2 :almond 1}}
```

```
(defn coffee= [coffee-a coffee-b] ;=> boolean
  (= coffee-a coffee-b))
```

Vector

```
{:size :small :roast :dark
 :add-ins [:soy :almond :soy]}
```

```
(defn coffee= [coffee-a coffee-b] ;=> boolean
  (= coffee-a coffee-b))
```

```
(defn how-many? [coffee add-in] ;=> natural-number
  (count (filter #{add-in} (:add-ins coffee))))
```

Map

```
{:size :small :roast :dark
 :add-ins {:soy 2 :almond 1}}
```

```
(defn coffee= [coffee-a coffee-b] ;=> boolean
  (= coffee-a coffee-b))
```

```
(defn how-many? [coffee add-in] ;=> natural-number
  (count (filter #{add-in} (:add-ins coffee))))
```

Vector

```
{:size :small :roast :dark
 :add-ins [:soy :almond :soy]}
```

```
(defn coffee= [coffee-a coffee-b] ;=> boolean
  (= coffee-a coffee-b))
```

```
(defn how-many? [coffee add-in] ;=> natural-number
  (count (filter #{add-in} (:add-ins coffee))))
```

```
(defn within-limit? [coffee min max] ;=> boolean
  (>= min (count (:add-ins coffee)) max))
```

Map

```
{:size :small :roast :dark
 :add-ins {:soy 2 :almond 1}}
```

```
(defn coffee= [coffee-a coffee-b] ;=> boolean
  (= coffee-a coffee-b))
```

```
(defn how-many? [coffee add-in] ;=> natural-number
  (get (:add-ins coffee) add-in 0)))
```

```
(defn within-limit? [coffee min max] ;=> boolean
  (>= min (reduce + 0 (vals (:add-ins coffee))) max))
```

Vector

```
{:size :small :roast :dark
 :add-ins [:soy :almond :soy]}
```

```
(defn coffee= [coffee-a coffee-b] ;=> boolean
  (= coffee-a coffee-b))
```

```
(defn how-many? [coffee add-in] ;=> natural-number
  (count (filter #{add-in} (:add-ins coffee))))
```

```
(defn within-limit? [coffee min max] ;=> boolean
  (>= min (count (:add-ins coffee)) max))
```

```
(defn add [coffee add-in] ;=> coffee
  (update coffee :add-ins (comp vec sort conj) add-in))
```

Map

```
{:size :small :roast :dark
 :add-ins {:soy 2 :almond 1}}
```

```
(defn coffee= [coffee-a coffee-b] ;=> boolean
  (= coffee-a coffee-b))
```

```
(defn how-many? [coffee add-in] ;=> natural-number
  (count (filter #{add-in} (:add-ins coffee))))
```

```
(defn within-limit? [coffee min max] ;=> boolean
  (>= min (count (:add-ins coffee)) max))
```

```
(defn add [coffee add-in] ;=> coffee
  (update-in coffee [:add-ins add-in] (fnil inc 0)))
```

Vector

```
{:size :small :roast :dark
 :add-ins [:soy :almond :soy]}
```

```
(defn coffee= [coffee-a coffee-b] ;=> boolean
  (= coffee-a coffee-b))
```

```
(defn how-many? [coffee add-in] ;=> natural-number
  (count (filter #{add-in} (:add-ins coffee))))
```

```
(defn within-limit? [coffee min max] ;=> boolean
  (>= min (count (:add-ins coffee)) max))
```

```
(defn add [coffee add-in] ;=> coffee
  (update coffee :add-ins (comp vec sort conj) add-in))
```

```
(defn remove [coffee add-in] ;=> coffee
  (assoc coffee :add-ins
    (loop [add-ins add-ins acc []]
      (cond
        (empty? add-ins)
        acc
        (= add-in (first add-ins))
        (into acc (rest add-ins))
        :else
        (recur (rest add-ins) (conj acc (first add-ins)))))))
```

Map

```
{:size :small :roast :dark
 :add-ins {:soy 2 :almond 1}}
```

```
(defn coffee= [coffee-a coffee-b] ;=> boolean
  (= coffee-a coffee-b))
```

```
(defn how-many? [coffee add-in] ;=> natural-number
  (count (filter #{add-in} (:add-ins coffee))))
```

```
(defn within-limit? [coffee min max] ;=> boolean
  (>= min (count (:add-ins coffee)) max))
```

```
(defn add [coffee add-in] ;=> coffee
  (update-in coffee [:add-ins add-in] (fnil inc 0)))
```

```
(defn remove [coffee add-in] ;=> coffee
  (if (<= 1 (get-in coffee [:add-ins add-in] 0))
    (update coffee :add-ins dissoc add-in)
    (update-in coffee [:add-ins add-in] dec)))
```

Vector

```
{:size :small :roast :dark
 :add-ins [:soy :almond :soy]}
```

```
(defn coffee= [coffee-a coffee-b]) ;=> boolean

(defn how-many? [coffee add-in]) ;=> natural-number

(defn within-limit? [coffee min max]) ;=> boolean

(defn add [coffee add-in]) ;=> coffee

(defn remove [coffee add-in]) ;=> coffee
```

Map

```
{:size :small :roast :dark
 :add-ins {:soy 2 :almond 1}}
```

```
(defn coffee= [coffee-a coffee-b]) ;=> boolean

(defn how-many? [coffee add-in]) ;=> natural-number

(defn within-limit? [coffee min max]) ;=> boolean

(defn add [coffee add-in]) ;=> coffee

(defn remove [coffee add-in]) ;=> coffee
```

Vector

```
{:size :small :roast :dark
 :add-ins [:soy :almond :soy]}
```

```
(defn coffee= [coffee-a coffee-b]) ;=> boolean

(defn how-many? [coffee add-in]) ;=> natural-number
```

Linear search????

```
(defn within-limit? [coffee min max]) ;=> boolean

(defn add [coffee add-in]) ;=> coffee

(defn remove [coffee add-in]) ;=> coffee
```

Linear search????

Map

```
{:size :small :roast :dark
 :add-ins {:soy 2 :almond 1}}
```

```
(defn coffee= [coffee-a coffee-b]) ;=> boolean

(defn how-many? [coffee add-in]) ;=> natural-number
```

Linear sum????

```
(defn within-limit? [coffee min max]) ;=> boolean

(defn add [coffee add-in]) ;=> coffee

(defn remove [coffee add-in]) ;=> coffee
```

Total functions

Level 2: Operation modeling

A total function is a function that is defined for all valid arguments.

```
(defn remove [coffee add-in]) ;=> coffee
```

```
(remove {:size :small :roast :medium :add-ins []} :soy)
```

3 options:

1. Restrict the arguments
2. Augment the return value
3. Change the meaning

1. Restrict the arguments

Making a function total

```
(defn remove [coffee add-in]) ;=> coffee

(remove {:size :small :roast :medium :add-ins []} :soy)❌
```

1. Restrict the arguments

Making a function total

```
(defn remove [coffee add-in] ;=> coffee
  {:pre [(pos? (how-many? coffee add-in))]})

(remove {:size :small :roast :medium :add-ins []} :soy)❌
```

- Make some combination of arguments invalid.
- Force the caller to check the arguments before calling.
- By changing the definition of “valid arguments”, I have made the function total.

2. Augment the return

Making a function total

```
(defn remove [coffee add-in]) ;=> coffee | nil

(remove {:size :small :roast :medium :add-ins []} :soy)
```

- Augment the return value with an extra state indicating failure.
- Force the caller to deal with the return value after calling.

2. Change the meaning

Making a function total

```
(defn remove [coffee add-in]) ;=> coffee

(remove {:size :small :roast :medium :add-ins []} :soy)
```

- Change meaning to *remove if it exists*.
- Some combinations of arguments return an unchanged coffee.
- All checks are contained in the function.

HTTP Client Example

Total functions

- With HTTP, you will get errors (timeouts, 500s, etc).
- How to make a request function total?

1. Restrict the arguments? NO
2. Augment the return? YES
3. Change the meaning? NO

HTTP Client Example

Total functions

```
{:status :success
 :value {..JSON..}}

|

{:status :error
 :code 500
 :message “Server error”}

(defn value-or-error [response]
  (case (:status response)
    :success (:value response)
    :error (throw (ex-info (:message response) response))))
```

- Precise set of meanings
- Complete set of meanings
- Minimal set of meanings (nothing unnecessary).
- Totality of functions
- Possibility/complexity of your functions (revisit data model)

Focus: Composition of functions

Level 3: Algebraic modeling

```
(let [coffee {:size :small :roast :light :add-ins []}]

  (assert (= coffee
             (-> coffee
                 (add    :espresso)
                 (remove :espresso)))))
```

Focus: Composition of functions

Level 3: Algebraic modeling

```
(let [coffee {:size :small :roast :light :add-ins []}
      [add-in (random-nth
               [:espresso :soy :almond :hazelnut :cream])]

  (assert (= coffee
             (-> coffee
                 (add    add-in)
                 (remove add-in)))))
```

Focus: Composition of functions

Level 3: Algebraic modeling

```
(let [coffee {:size    (random-size)
              :roast   (random-roast)
              :add-ins (random-add-ins)}
      add-in (random-nth
               [:espresso :soy :almond :hazelnut :cream])]

  (assert (= coffee
             (-> coffee
                 (add    add-in)
                 (remove add-in)))))
```

Focus: Composition of functions

Level 3: Algebraic modeling

```
(let [coffee {:size    (random-size)
              :roast   (random-roast)
              :add-ins (random-add-ins)}
      add-ins (random-add-ins)

      coffee-with    (reduce add    coffee      add-ins)
      coffee-without (reduce remove coffee-with add-ins)]

  (assert (= coffee coffee-without)))
```

Focus: Composition of functions

Level 3: Algebraic modeling

```
(let [coffee {:size    (random-size)
              :roast   (random-roast)
              :add-ins (random-add-ins)}
      add-ins  (random-add-ins)
      add-ins’ (shuffle add-ins)

      coffee-with    (reduce add    coffee      add-ins)
      coffee-without (reduce remove coffee-with add-ins’)]

  (assert (= coffee coffee-without)))
```

Focus: Composition of functions

Level 3: Algebraic modeling

```
Relationship between add, remove, and how-many?

(< (how-many? coffee add-in)
   (how-many? (add coffee add-in) add-in))
(>= (how-many? coffee add-in)
    (how-many? (remove coffee add-in) add-in))

Relationship of add with itself?

(= (-> coffee (add a) (add b))
   (-> coffee (add b) (add a)))

Relationship of remove with itself?

(= (-> coffee (remove a) (remove b))
   (-> coffee (remove b) (remove a)))
```