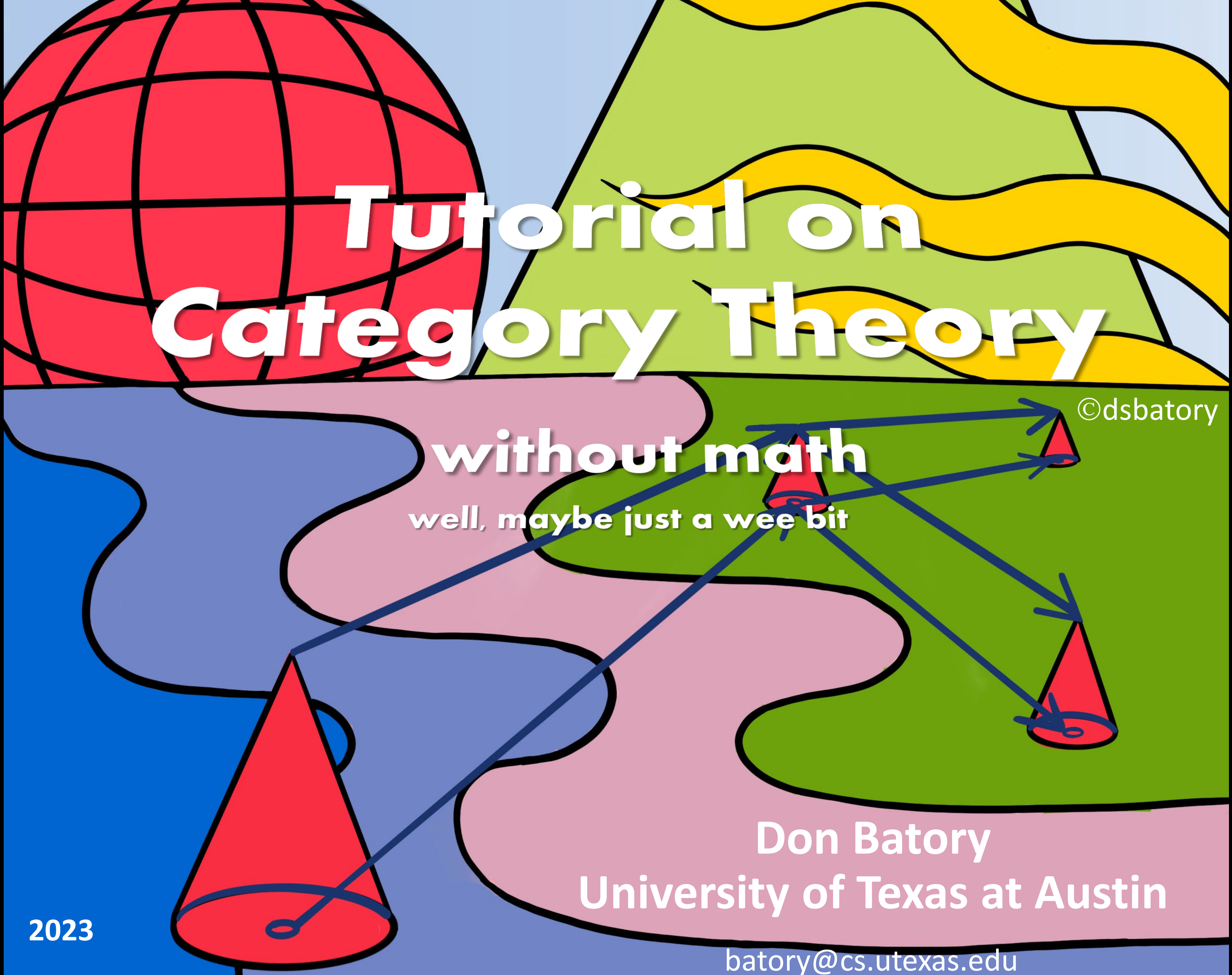


Tutorial on Category Theory

without math

well, maybe just a wee bit

Don Batory
University of Texas at Austin
batory@cs.utexas.edu

2023

Good Morning From Austin, Texas!

1. Why am I in Austin?
   - I'm ancient with health problems

2. My limitations were clearly stated in my proposal, but it was accepted anyway ☺

3. The proposal is 2 one-hour lectures. I was given 90 minutes, as all 2-hour tutorials. I pause the presentation at 90 mins, finish 14 minutes later ☺

My Background

- I studied relational databases from 1975-1990s and
  - early on was interested in the design of DBMS software
  - I was a closet **Software Engineer (SE)** but didn't know it

- Like many SE researchers back then, I saw little connection and relevance of mathematics to software design. Seemed a waste of time
  - studied small problems, worked really hard, for small solutions that didn't scale

- Eventually I realized I needed a language that would allow me to express design concepts in Software Product Lines, ***Model Driven Engineering***, and Dataflow Programming. **Category Theory (*CT*)** basics fit the bill. ***CT*** is *not* "abstract nonsense". It is profound and immensely practical.

My Caveat

- I am NOT a mathematician; I am a Software Engineer with DBMS background
- Category Theory is vast; I know some basics

  - **DBMS History**: When E.F. Codd proposed Relational Model in 1970, mathematicians rejoiced that Set Theory 1880s was its foundation
    - within 5-10 years, only the first few pages of a set theory text was ever used
    - 50+ years later, not much has changed IMO

- I argue the same holds for Category Theory for MDE
    - ***CT*** 1945 is a/the mathematical foundation of MDE 1995

Your Take Aways from this Lecture: ***CT*** is…

1. An essential counterpart to UML class diagrams

2. Should be taught in grad MDE courses ***and***
   in intro SE grad courses ***and***
   some ideas appropriate at undergrad level

3. An elegant and practical modeling language and way of thinking

- I will pause after a slide with this symbol which means "any questions?"

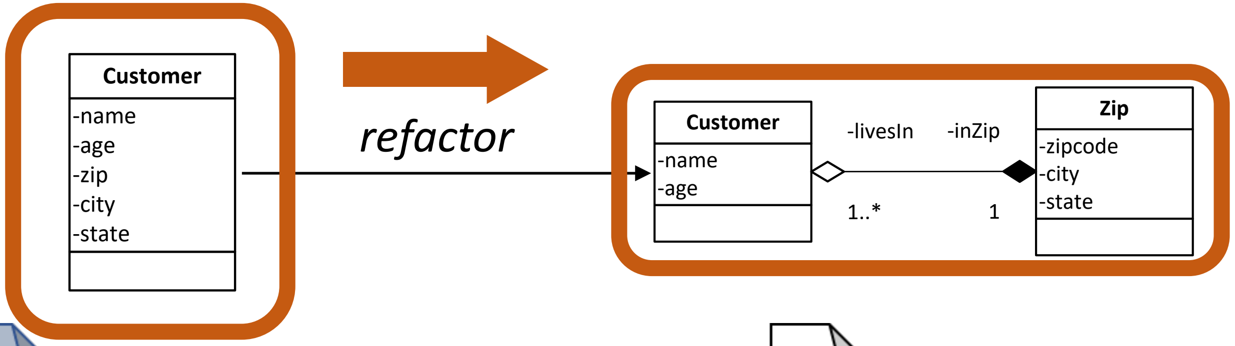
Fundamentals of Information Organization

As computer scientists and software engineers,
we collect information, process information,
and produce new information… we do this all the time.
CT is the foundation of this Universe

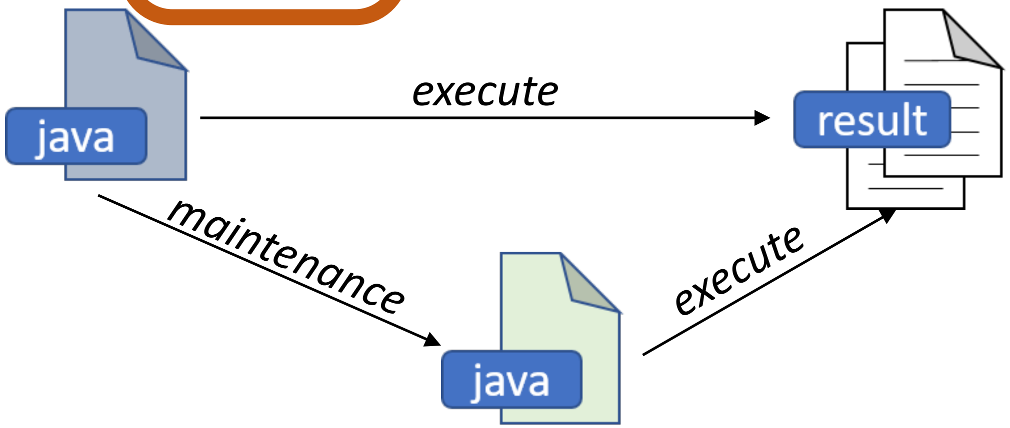
Fundamentals of Information Organization

- Four ideas:
  1. information **structures** (aka objects, things, models)
  2. **computation** on a structure produces another structure (aka arrow, transformation)

- Ex: refactoring a class diagram

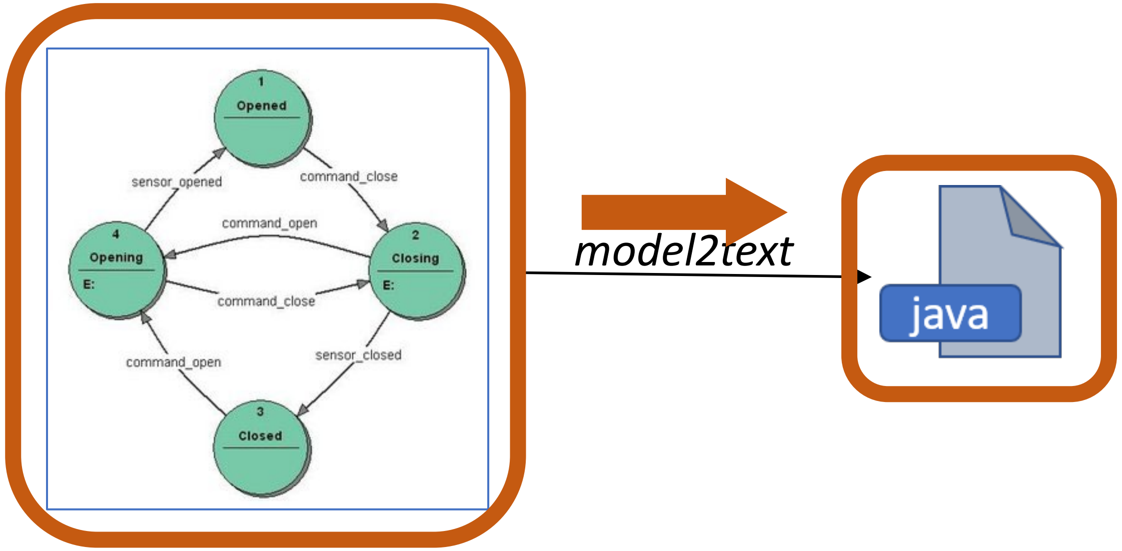


- Regression testing



Fundamentals of Information Organization

- Model Driven Engineering
  Define a Finite State Machine of elevator door and convert it into Java



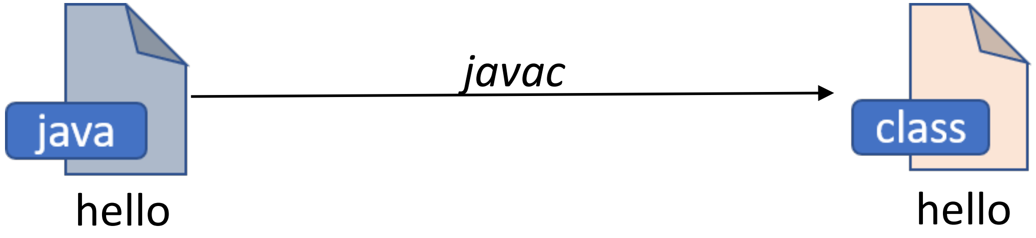
- SQL Query
  for a given DB

```
select name, city
from CustomerDelivery
where deliveryDate = "May 1"
```

execute

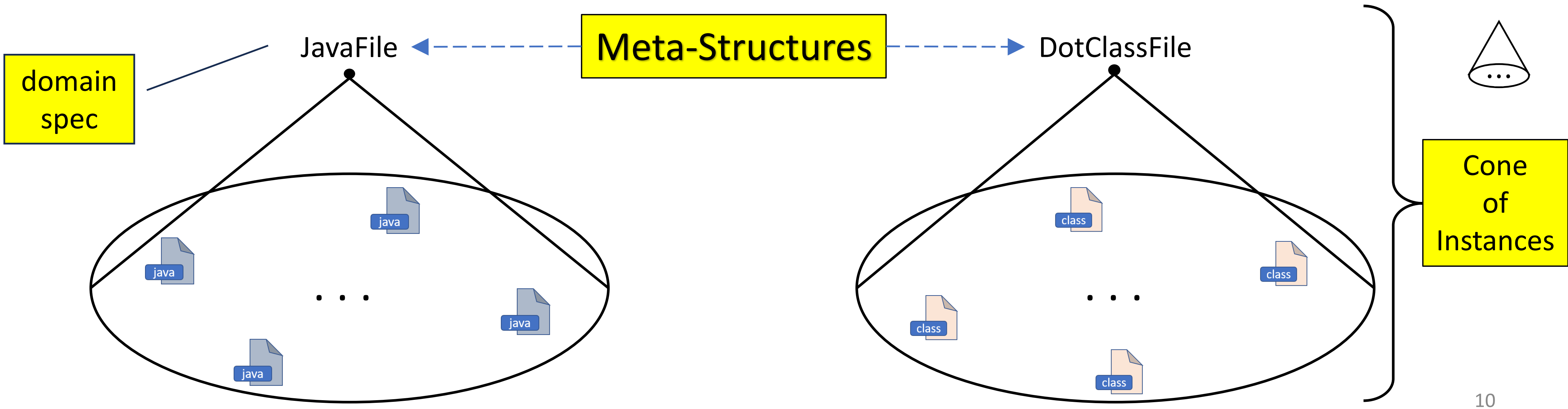
name	city
Einstein	Princeton
Gates	Seattle
Murray	Chicago
Carson	Los Angeles

- Java program



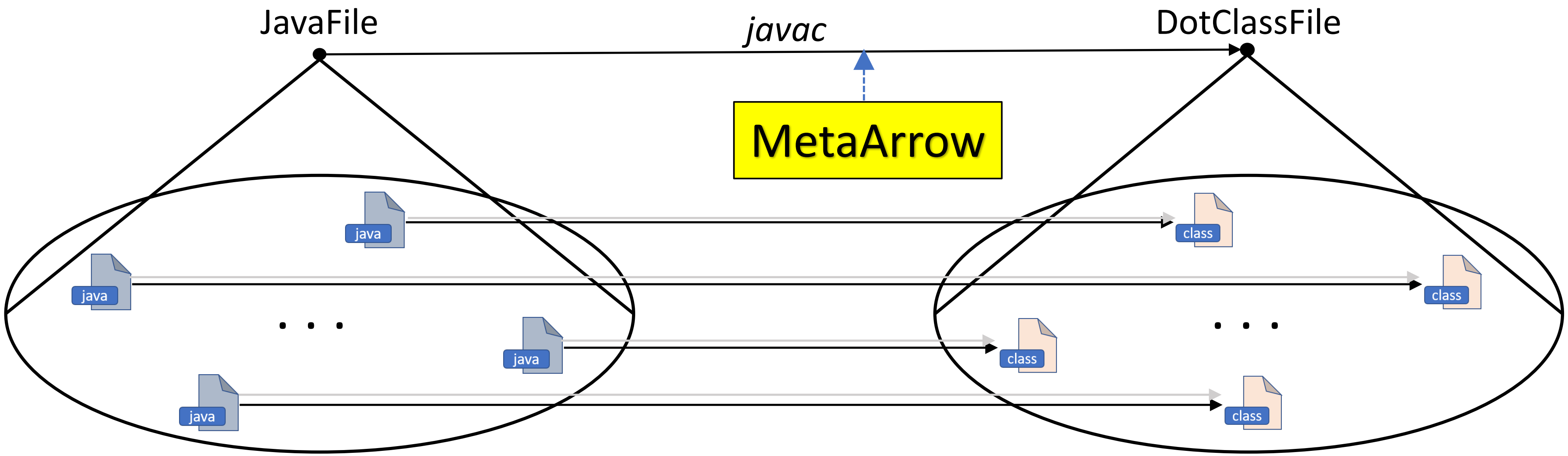
Fundamentals of Information Organization

- Four ideas:
    1. information structures (aka objects, things, models)
    2. computation on a structure produces another structure (aka arrow, transformation)
    3. structures are instances of **meta-structures** (types or meta-models)



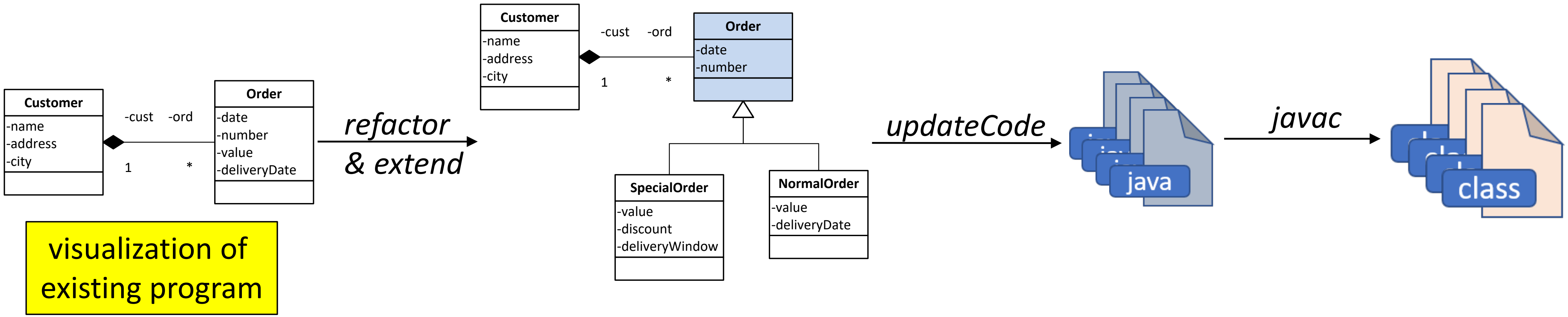
Fundamentals of Information Organization

- Four ideas:
  1. information structures (aka objects, things, models)
  2. computation on a structure produces another structure (aka arrow, transformation)
  3. structures are instances of meta-structures (types or meta-models)
  4. computations are instances of a **meta-computation** (meta-arrow)



How are Arrows Useful? Answer #1

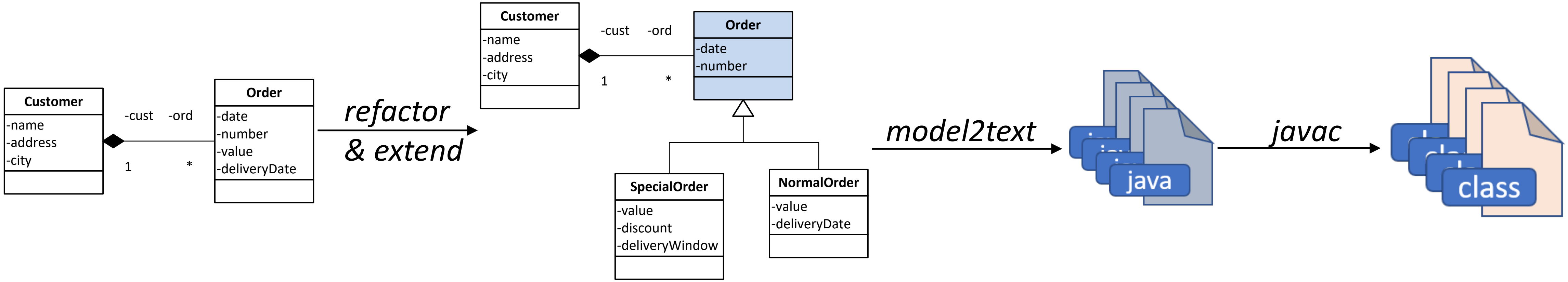
- We need to define **computational relationships** between structures
- UML class diagrams are pre-eminent ways to express structures in UML
- Arrows tell us what our programs do with structures – is a natural fit



- This is a trivial category diagram, but it is still useful

How are Arrows Useful? Answer #2

- UML class diagrams are **declarative** – abstract away implementation detail
  - Forces designers to focus on essential, not artificial or low-level details
- Same for arrows – abstract away implementation detail



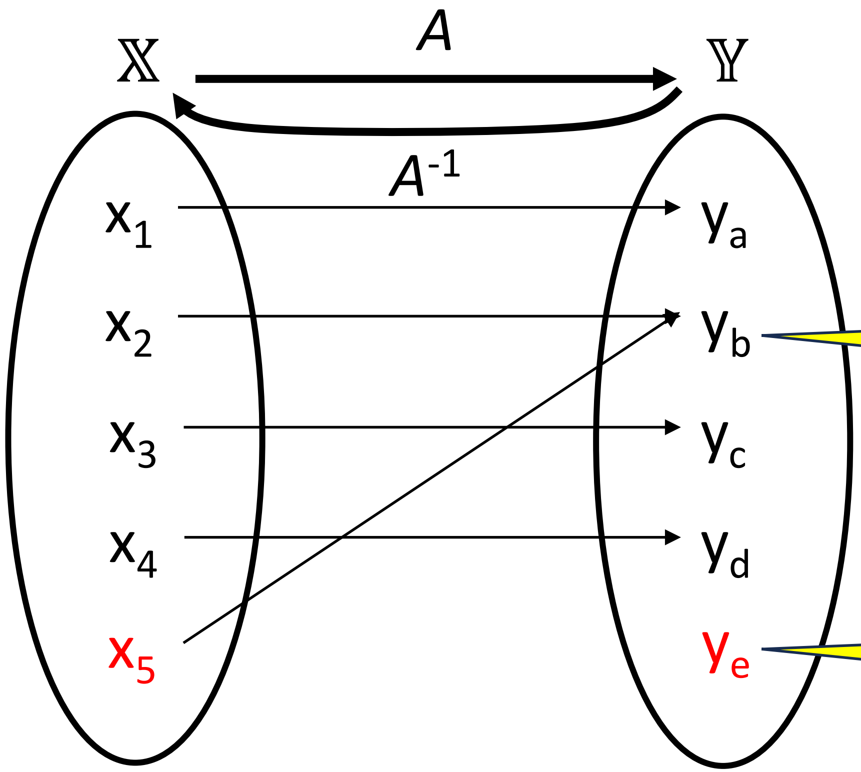
- ***CT*** is essential counterpart to UML diagrams and other SE representations

Meta-Arrows

- Are **total functions**: every element of the input domain is paired with produces an element of the output domain.

$$A\colon \mathbb{X} \to \mathbb{Y}$$

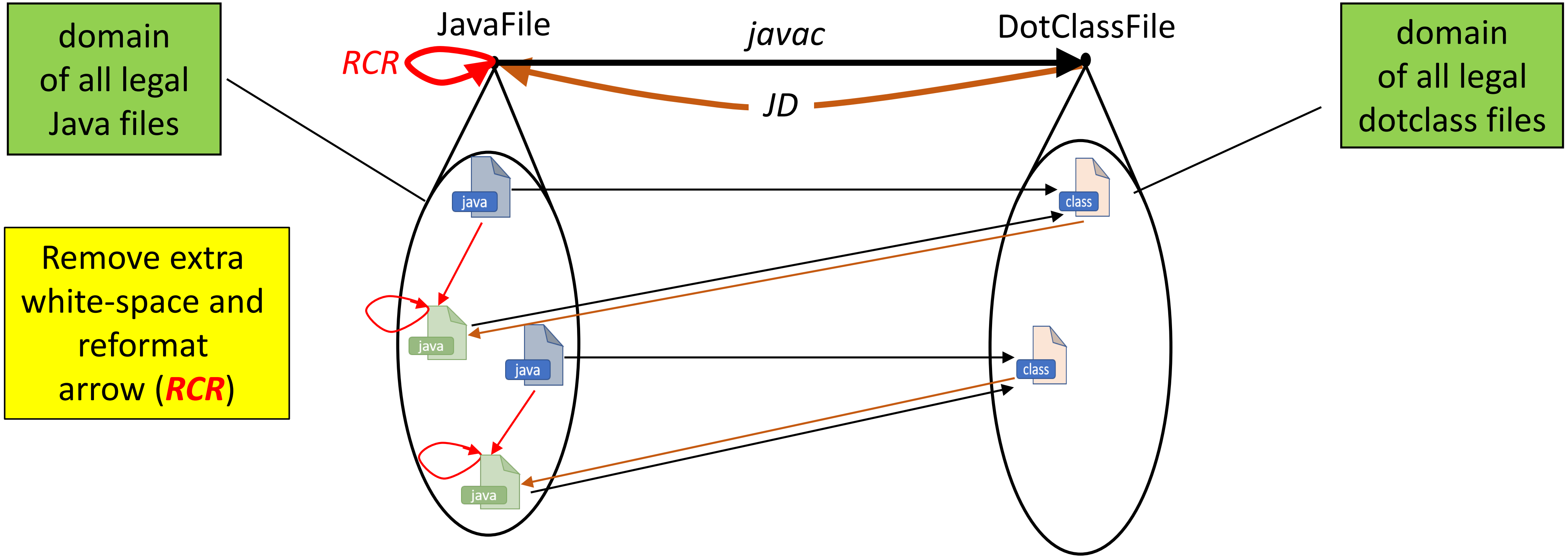
a 1:1 correspondence between $\mathbb{X}$ and $\mathbb{Y}$ elements means *A* has an inverse (A^{-1})



OK if some $\mathbb{X}$ values are paired with same $\mathbb{Y}$ value

OK if some $\mathbb{Y}$ values are not paired with an $\mathbb{X}$ value

Non-Trivial Example

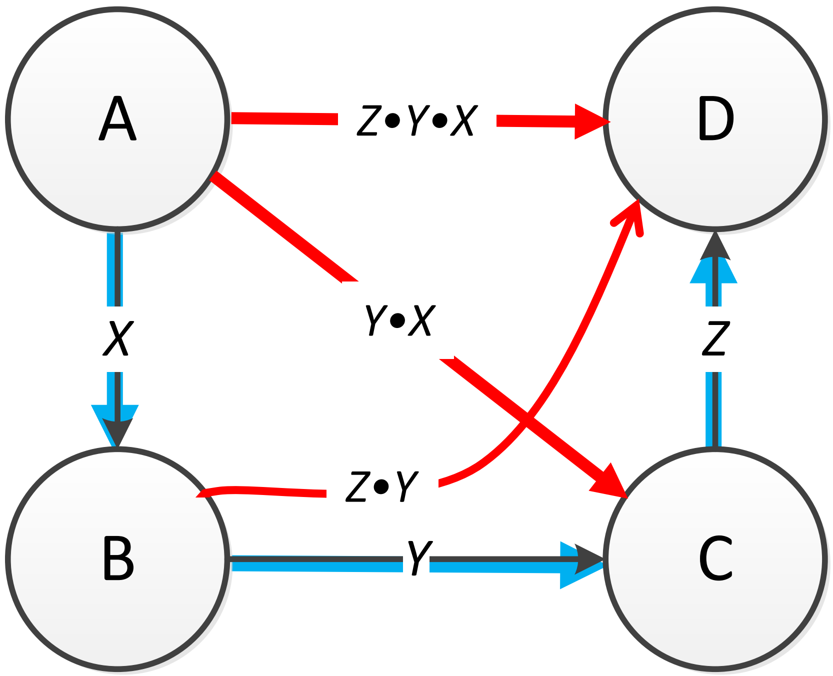


- Ignoring errors: is ***javac*** a total function? Yes!
- JavaDecompiler (***JD***) translates DotClassFile into a JavaFile: is it a total function? Yes!
- Is there an arrow that connects a blue Java file to its green Java? Yes!
- Arrow ***RCR*** removes comments and reformats a blue Java file → a green Java file
- How does ***RCR*** map green files? How does ***javac*** map green files?

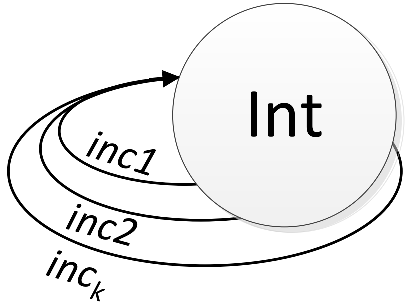
Axioms of ***CT***

1. Arrows compose: $X: \mathrm{A} \to \mathrm{B}$ and $Y: \mathrm{B} \to \mathrm{C}$ then $Y \cdot X: \mathrm{A} \to \mathrm{C}$

function composition



Like transitive closure but... closure can yield an non-finite category

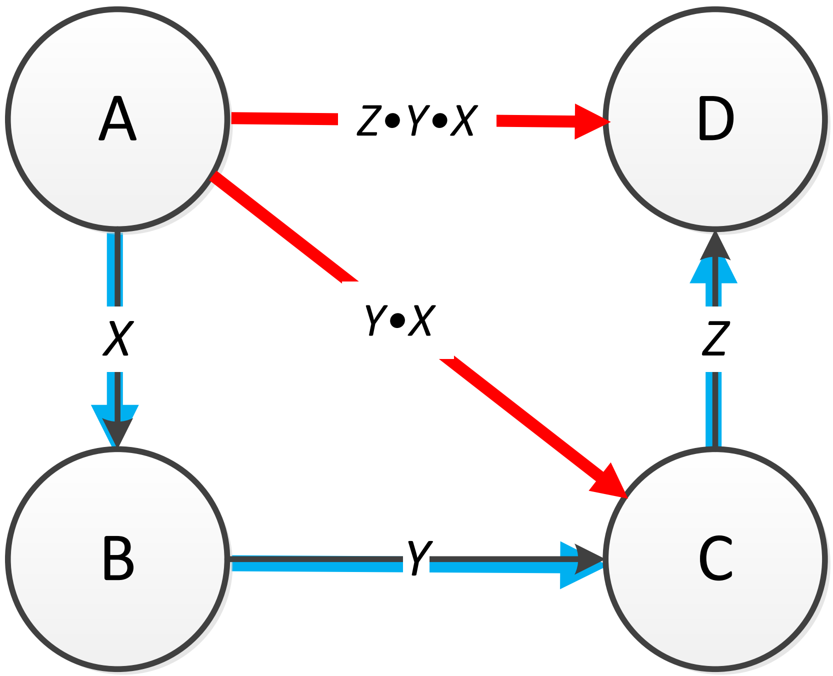


Axioms of ***CT***

1. Arrows compose: $X\!: \mathrm{A} \rightarrow \mathrm{B}$ and $Y\!: \mathrm{B} \rightarrow \mathrm{C}$ then $Y \cdot X\!: \mathrm{A} \rightarrow \mathrm{C}$
2. Composition is associative: $Z \cdot (Y \cdot X) = (Z \cdot Y) \cdot X$

function composition

function composition is associative



Axioms of ***CT***

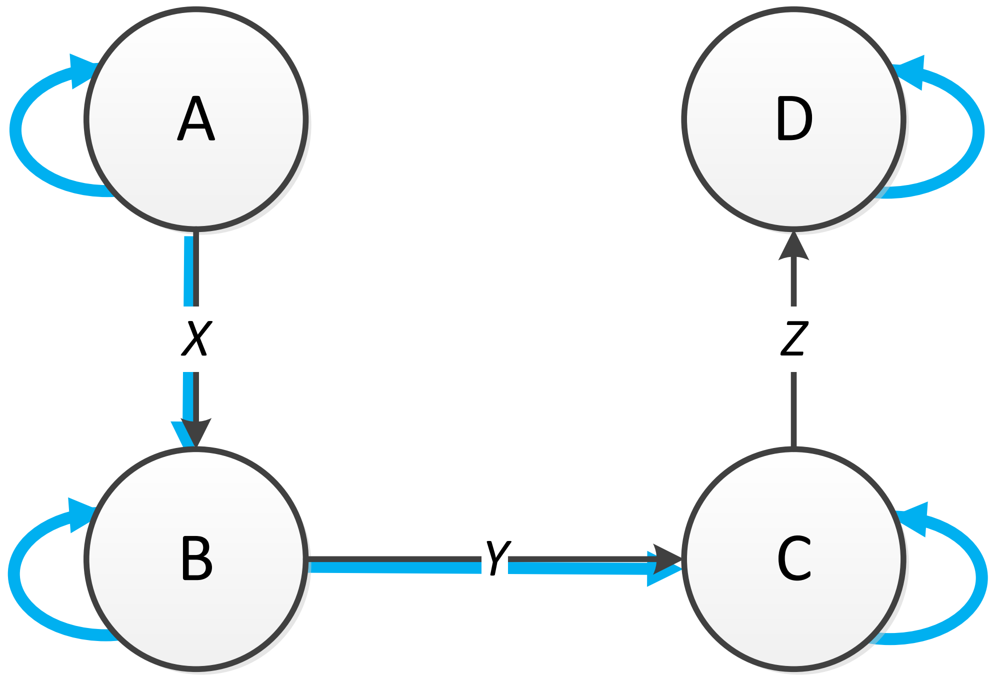
1. Arrows compose: $X\!: \mathrm{A} \to \mathrm{B}$ and $Y\!: \mathrm{B} \to \mathrm{C}$ then $Y \cdot X\!: \mathrm{A} \to \mathrm{C}$

function composition

2. Composition is associative: $Z \cdot (Y \cdot X) = (Z \cdot Y) \cdot X$

function composition is associative

3. Every domain structure has an identity arrow:



$\forall\, \mathrm{b} \in \mathrm{B}\!: \quad \mathrm{I_B(b)} = \mathrm{b} \quad \wedge$

$\mathrm{I_B} \cdot \mathrm{X} = \mathrm{X} \quad \wedge$

$\mathrm{Y} \cdot \mathrm{I_B} = \mathrm{Y}$

Axioms of ***CT***

1. Arrows compose: $X: \mathrm{A} \to \mathrm{B}$ and $Y: \mathrm{B} \to \mathrm{C}$ then $Y \cdot X: \mathrm{A} \to \mathrm{C}$
2. Composition is associative: $Z \cdot (Y \cdot X) = (Z \cdot Y) \cdot X$
3. Every domain structure has an identity arrow:

function composition

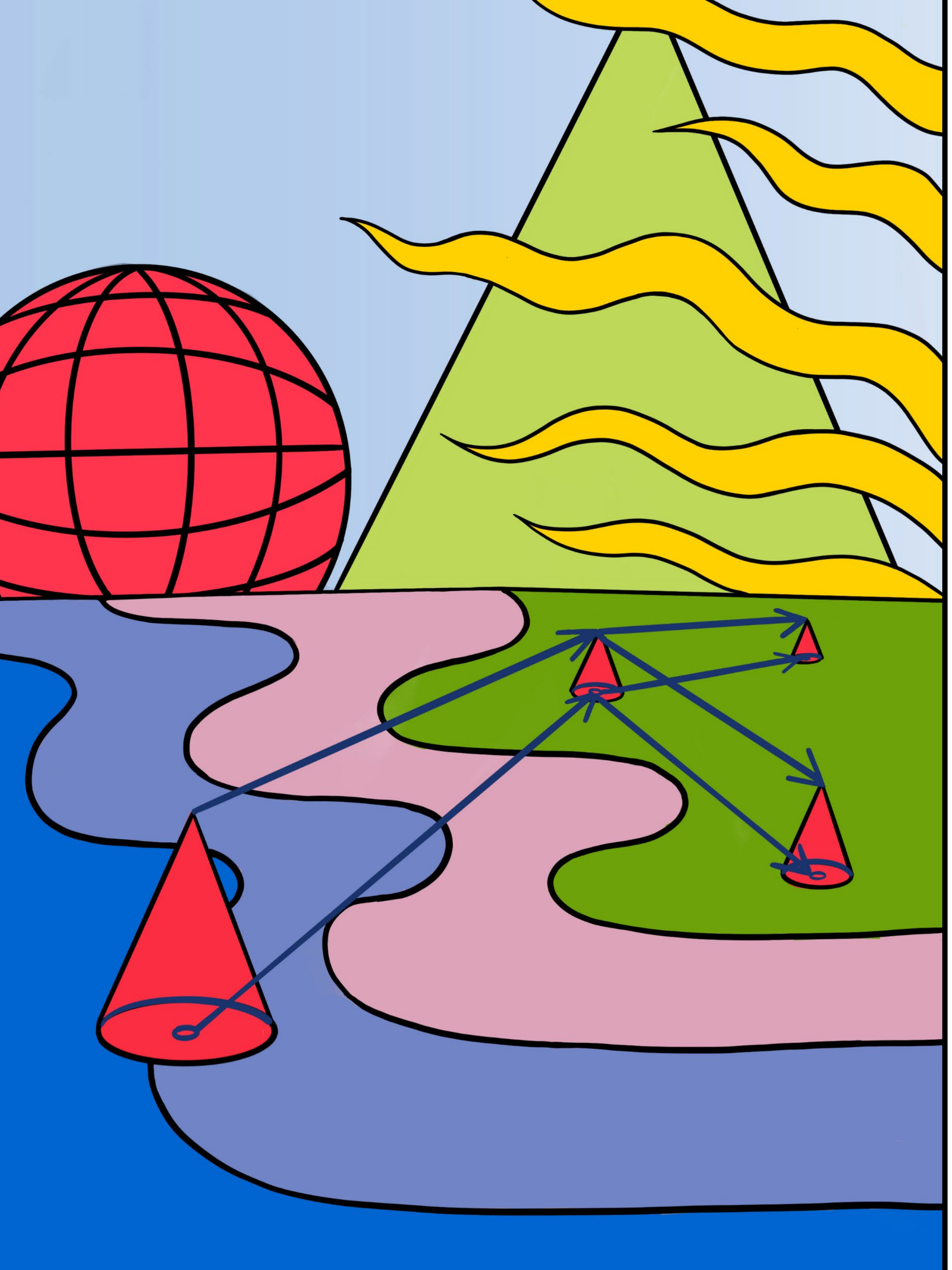
function composition is associative

These are the 3 axioms of Category Theory

$\forall\, \mathrm{b} \in \mathrm{B}: \quad \mathrm{I_B(b) = b} \quad \wedge$

$\mathrm{I_B \cdot X = X} \quad \wedge$

$\mathrm{Y \cdot I_S = Y}$

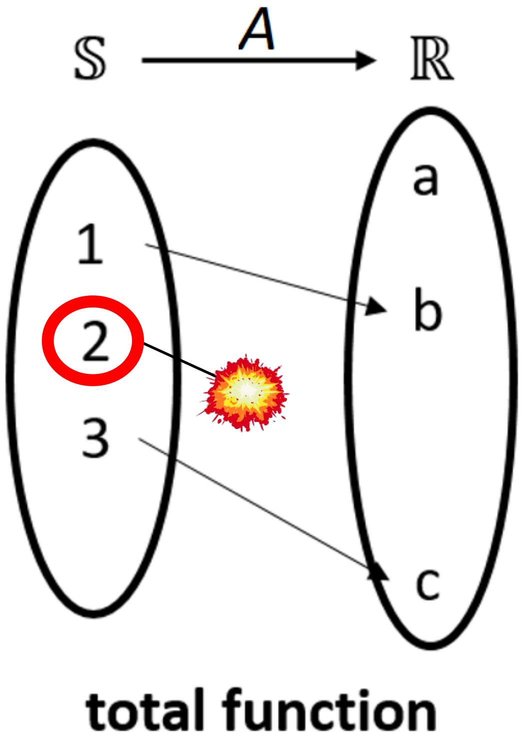


More Terminology and Facts about *CT*

Errors Occur!

- How does ***CT*** handle errors?

- When an error occurs, computation stops

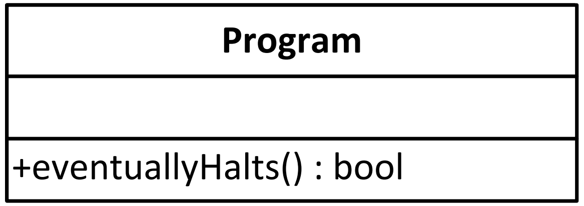


“Arrows are not Constructive”

- Meaning you can draw arrows and not know how to implement them

- This is the “Halting Problem” for which no general algorithm exists Turing 1936

- Ans: Why create this UML class diagram?



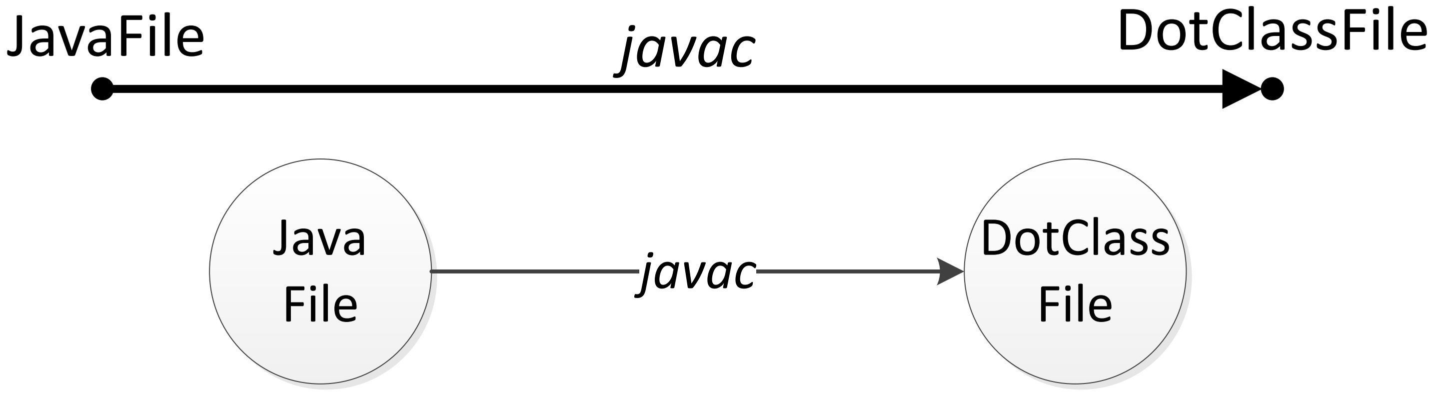
UML class diagrams are not constructive either!

Both are Declarative

Seems silly – why would anyone need this generality?

Category Diagram

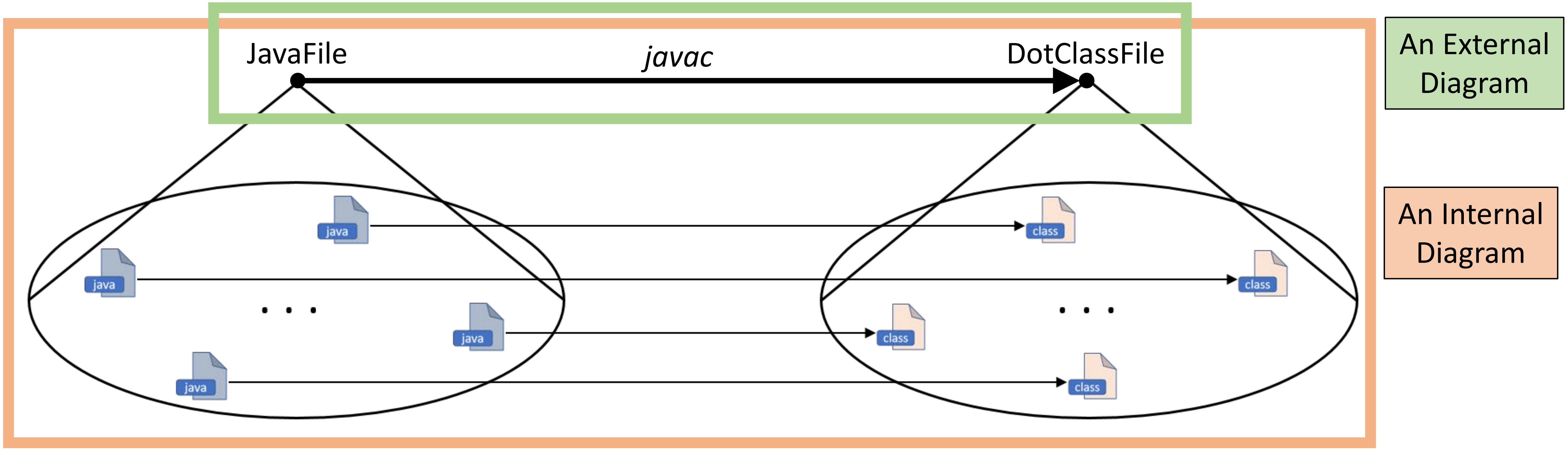
- A **category** or an **external diagram** is a directed multi-graph of labeled nodes domains and labeled edges arrow from input domain to output domain



javac: JavaFile → DotClassFile

Internal Category Diagram

- A **category** or an **external diagram** is a directed multi-graph of labeled nodes domains and labeled edges arrow from input domain to output domain

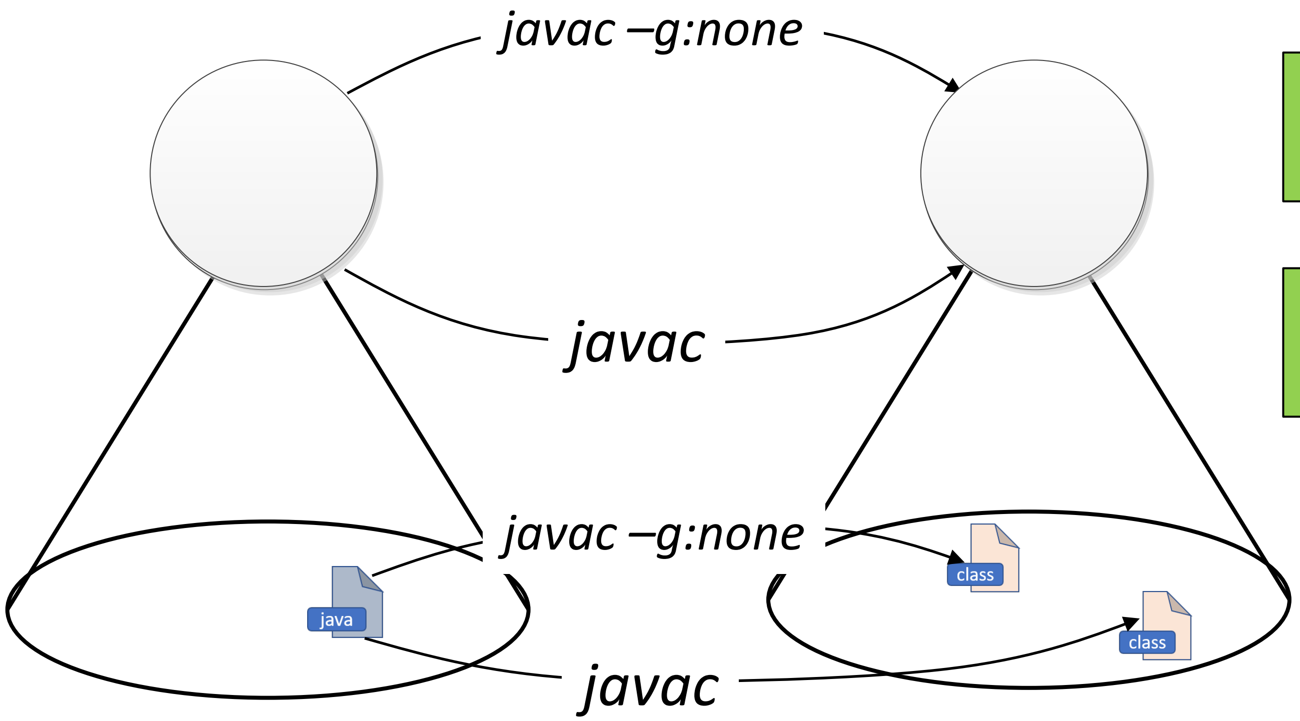


- An **internal diagram** shows the cone-of-instances for each domain
  - and arrow instances consistent with meta-arrow(s) shown

What is a Directed Multigraph?

- An external or internal diagram (**directed multigraph**) can have many arrows from source to target

What tool invocation is "?"



Hint: gcc compiler has optimizer option

Java w/w.o. debugging information

Every Arrow in ***CT*** has 1 domain and 1 codomain: *That's too restrictive!*

- Earlier example: SQL Query *for a given DB*

```
select name, city
from CustomerDelivery
where deliveryDate = "May 1"
```

execute →

name	city
Einstein	Princeton
Gates	Seattle
Murray	Chicago
Carson	Los Angeles

- Makes more sense if execute() has 2 arguments: a DB ***and*** a query

execute ( ,

```
select name, city
from CustomerDelivery
where deliveryDate = "May 1"
```

) =

name	city
Einstein	Princeton
Gates	Seattle
Murray	Chicago
Carson	Los Angeles

execute ( ,

```
select name, city
from CustomerDelivery
where deliveryDate = "May 1"
```

) =

name	city
Feinman	Pasadena
Zwicky	Pasadena

execute ( ,

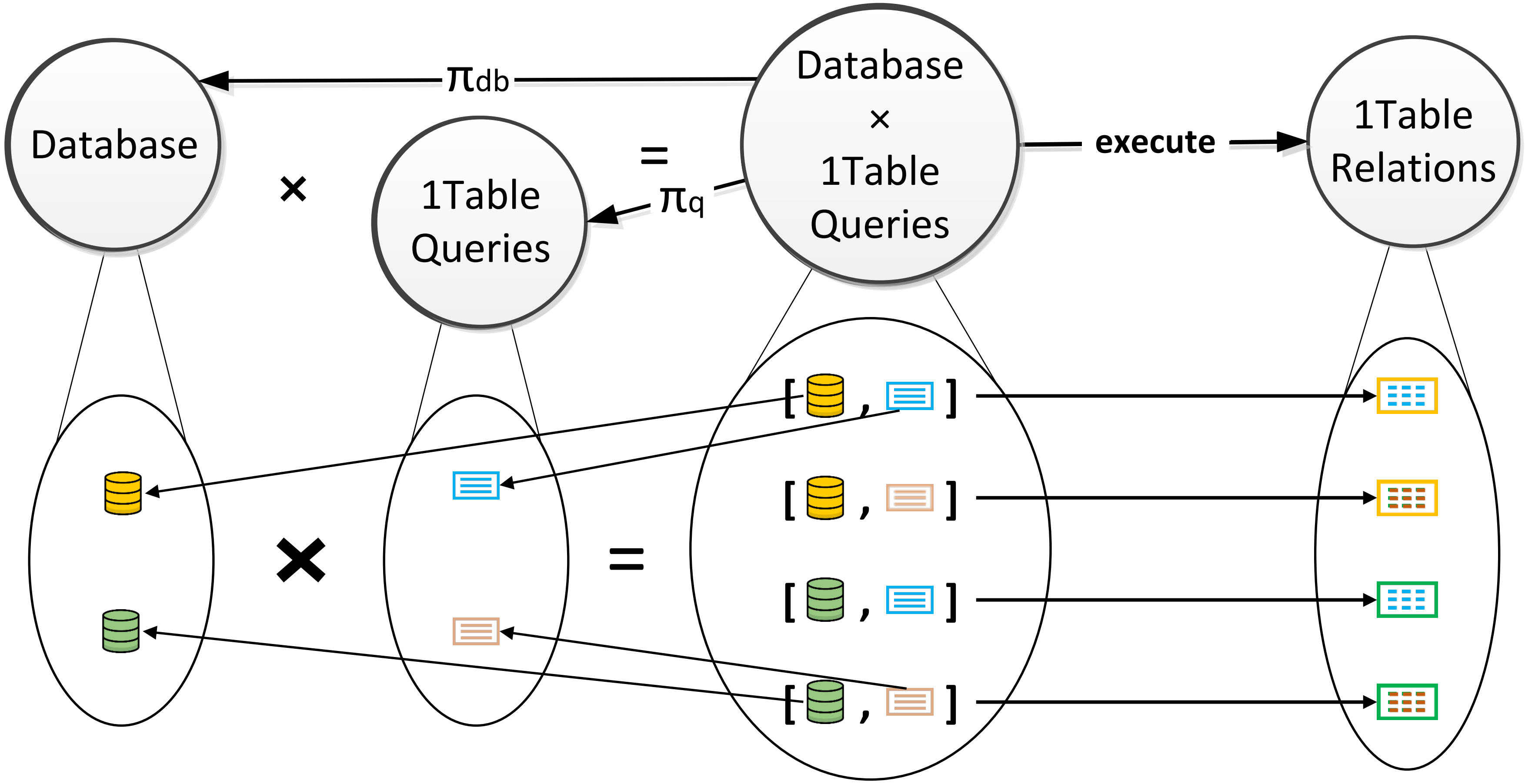
```
select *
from Customers
where birthdate=9/21
```

) =

name	city
Batory	Austin
Murray	Chicago
Wells	London
Cohen	Montreal

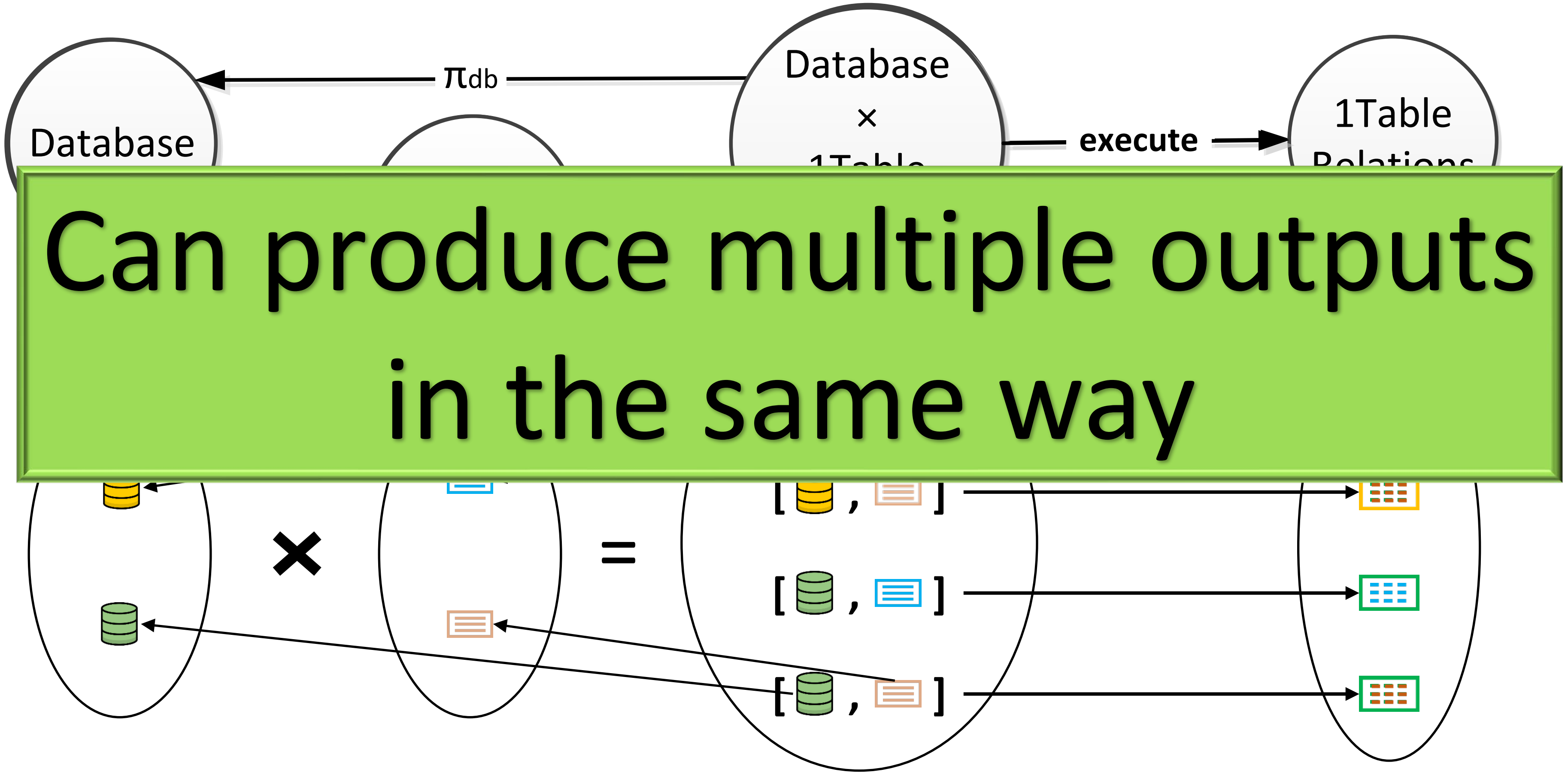
Solution to multiple input/output problem

- Take **cross-product of domains** to produce an input domain of multiple arguments

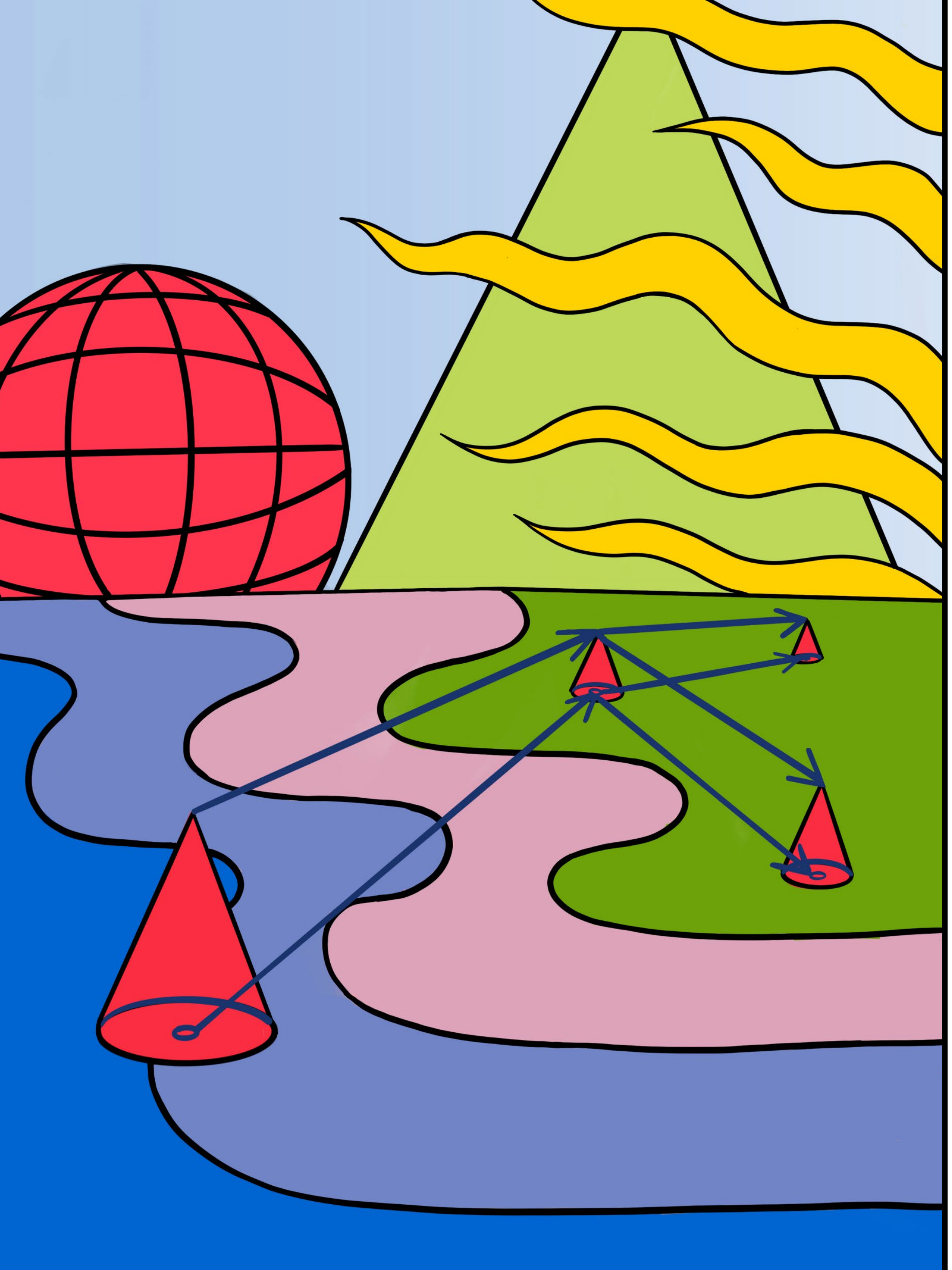


Solution to multiple input/output problem

- Take **cross-product of domains** to produce an input domain of multiple arguments



Can produce multiple outputs in the same way



Type Recursion

Fundamentals of Information Organization

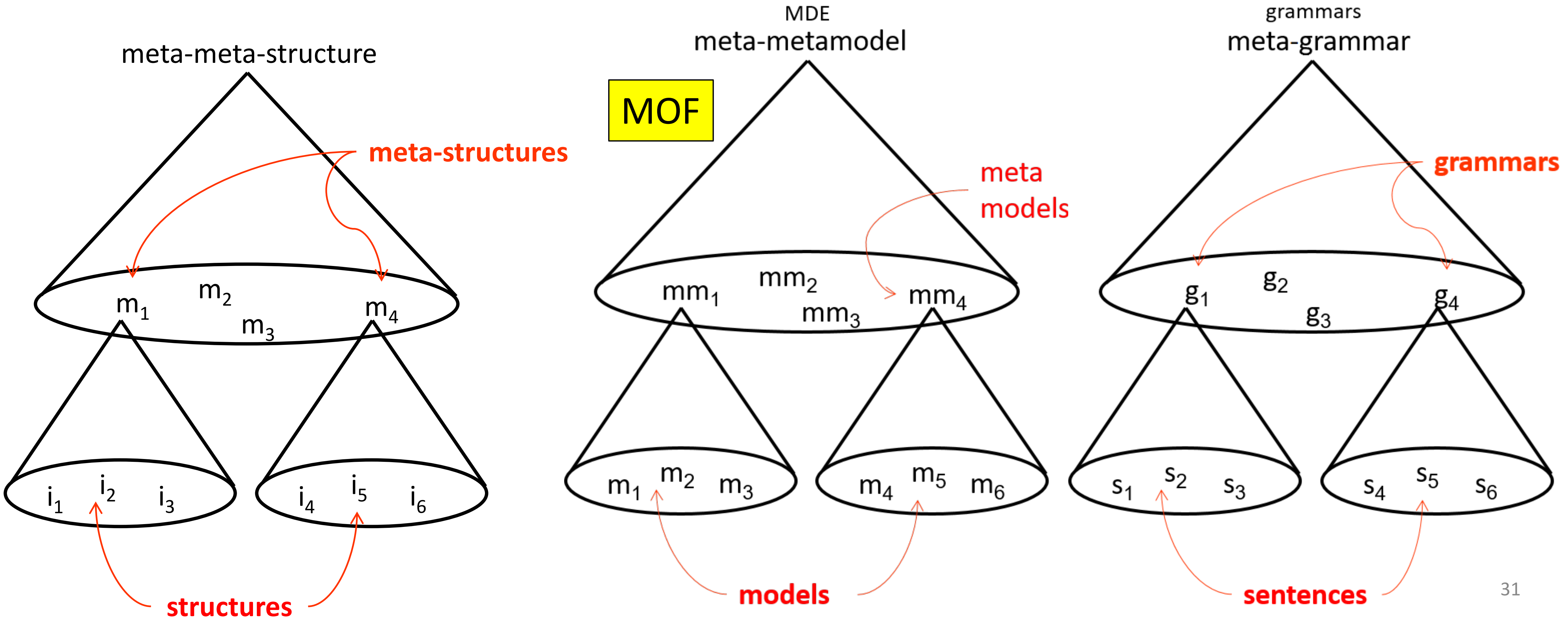
- Four ideas:
  1. information structures (aka objects, things, models)
  2. computation on a structure produces another structure (arrow)
  3. structures are instances of a meta-structure (type or meta-model)
  4. computations are instances of a meta-computation (meta-arrow)

Recursion (items 3–4)

Every level of abstraction in *CT* is governed by the same principles/ideas

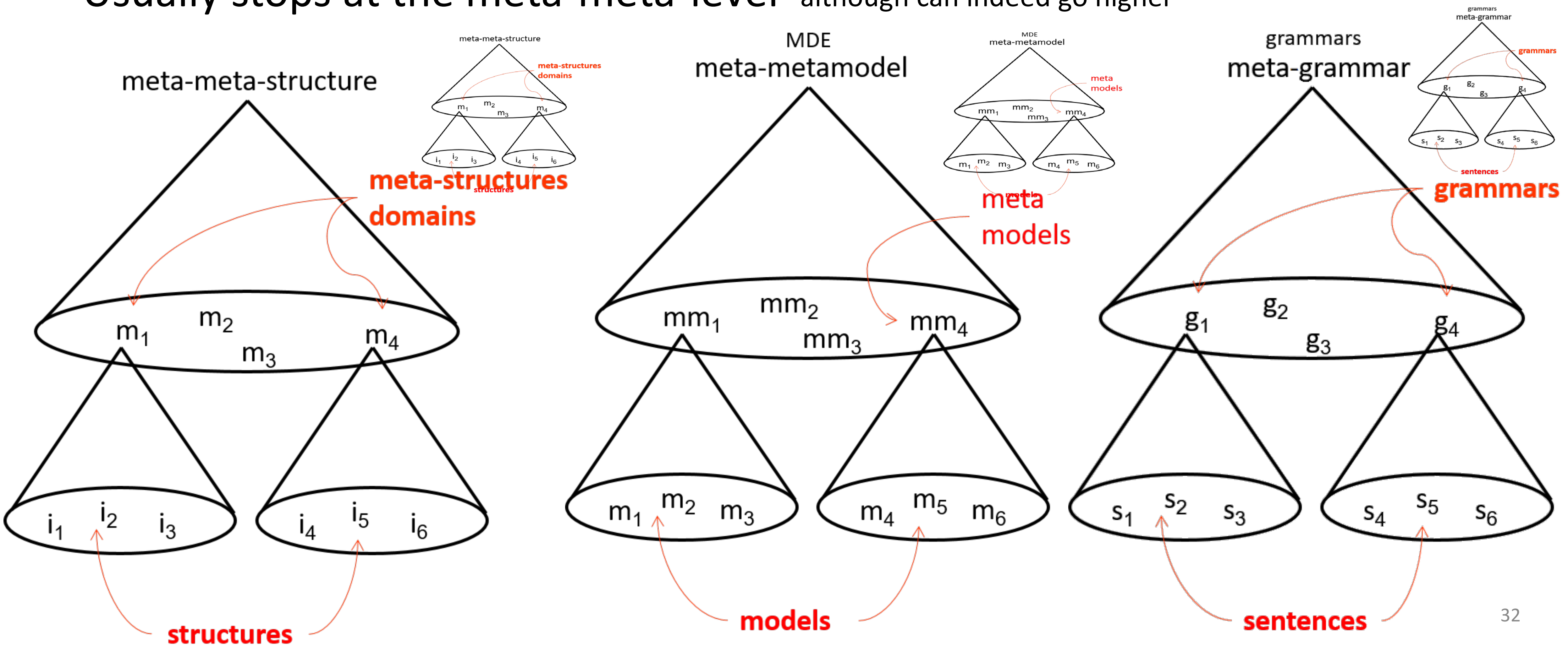
Examples of Type Recursion

- Usually stops at the meta-meta-level although can indeed go higher



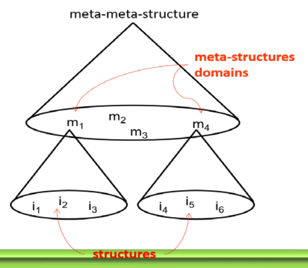
Examples of Type Recursion

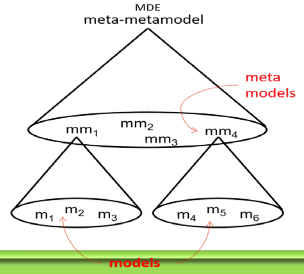
- Usually stops at the meta-meta-level although can indeed go higher

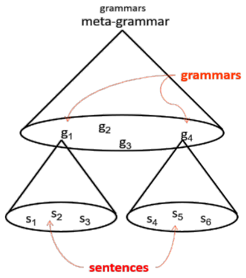


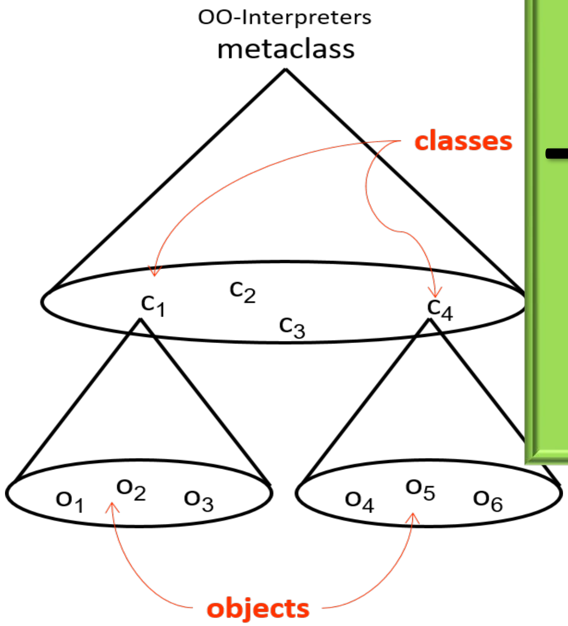
Examples of Type Recursion

- Usually stops at the meta-meta-level although can indeed go higher

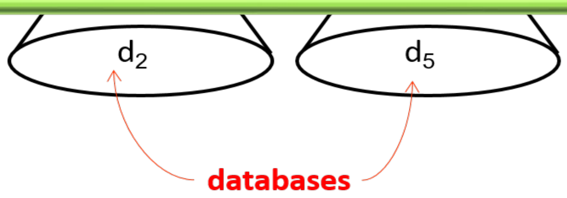


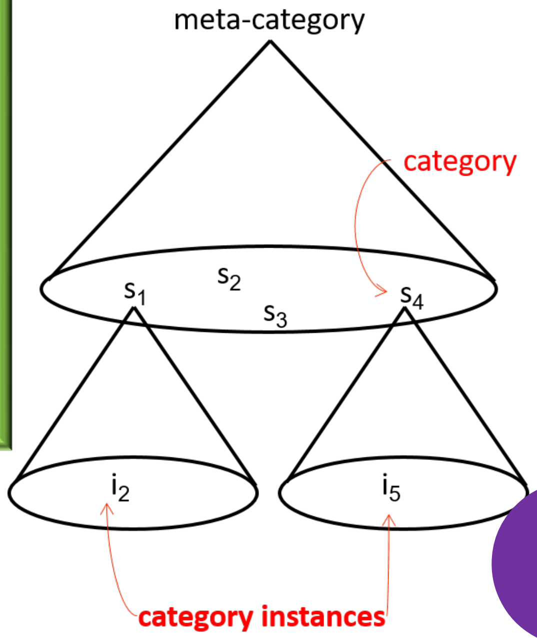


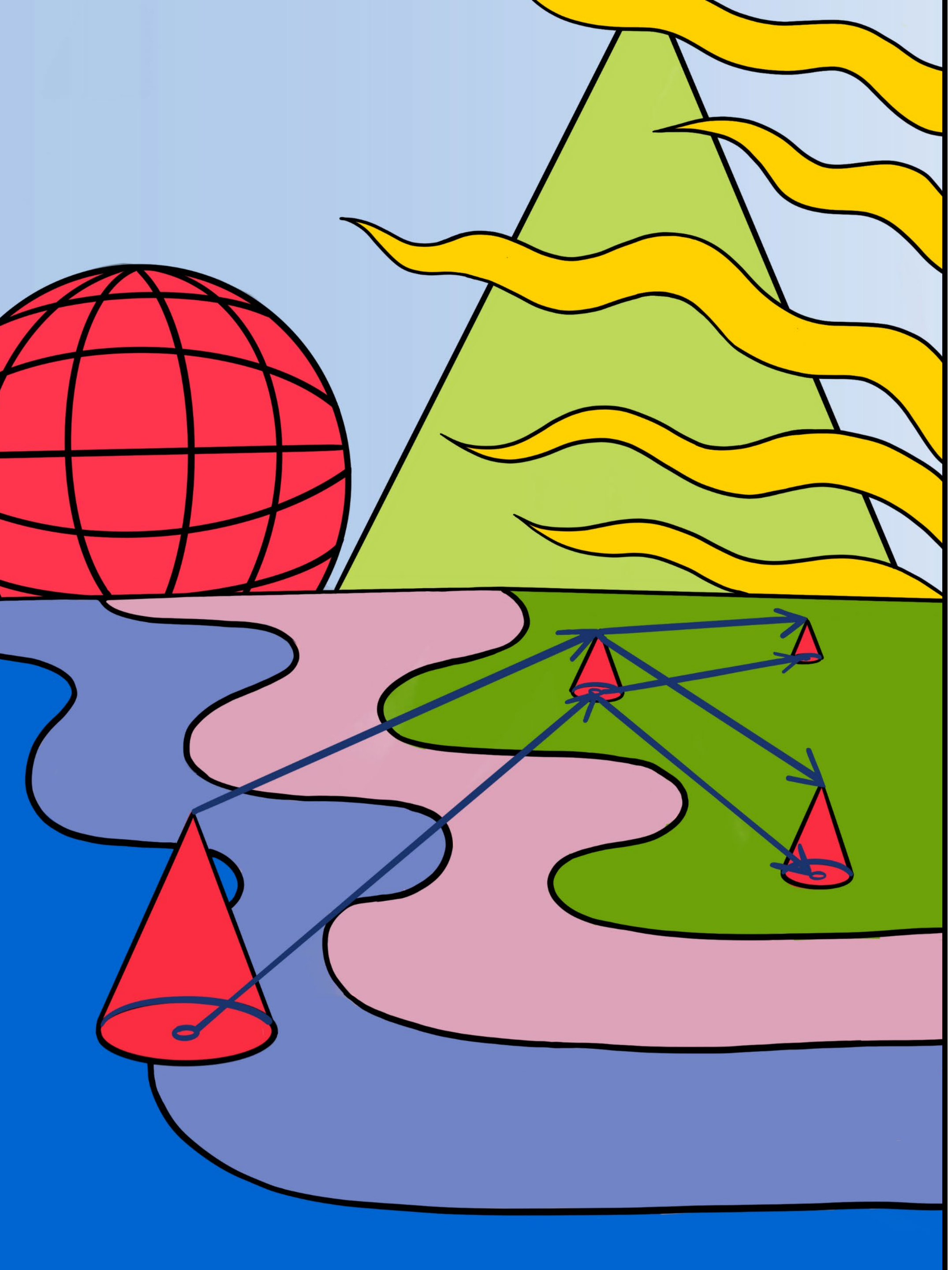




Not accidental!
This is the Structure of Information





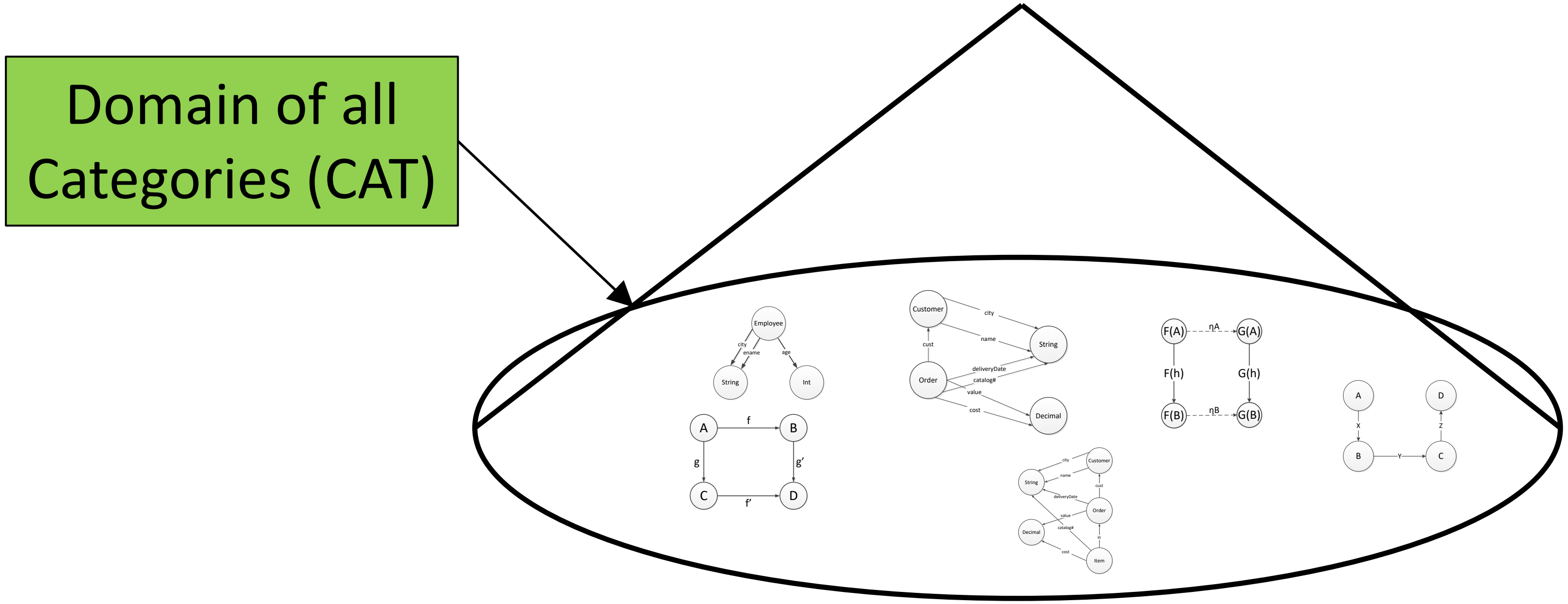


Containment Recursion

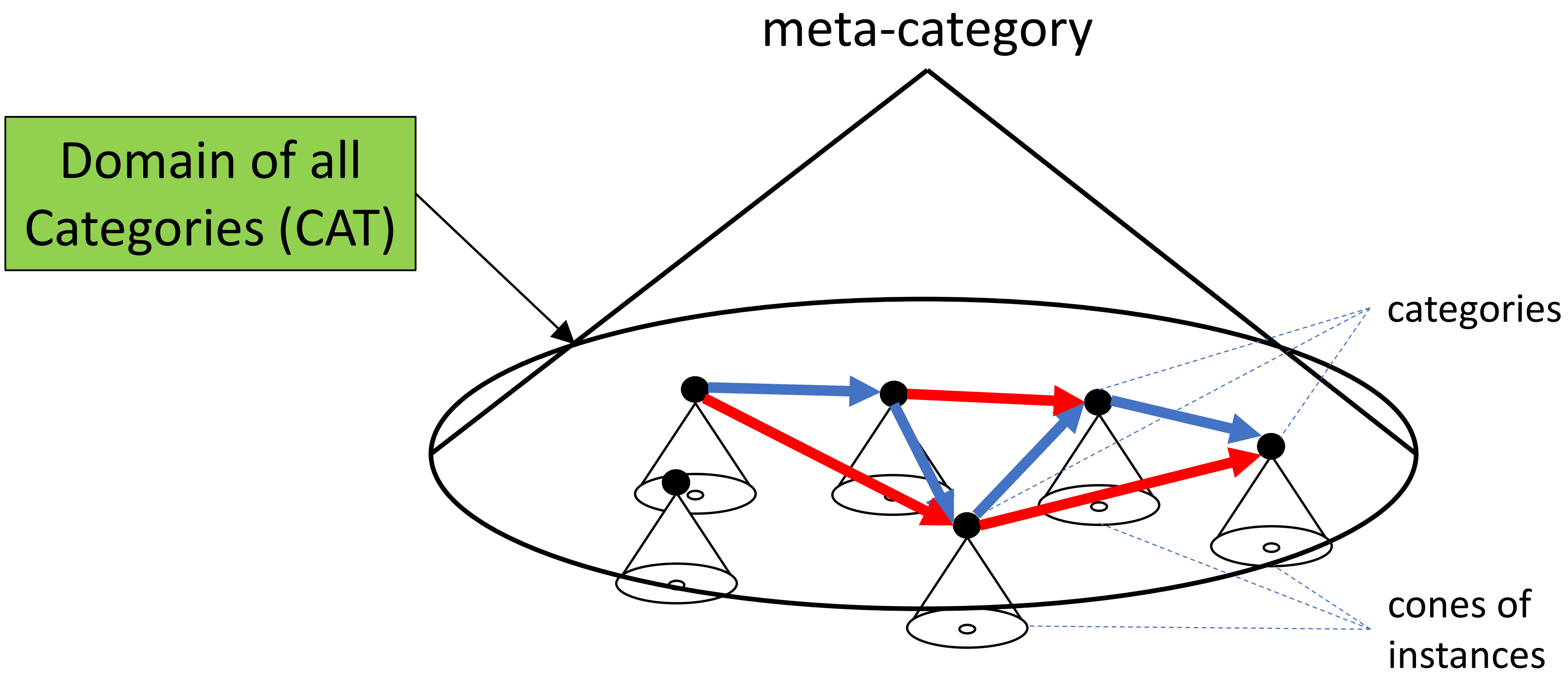
different from type recursion

Containment Recursion

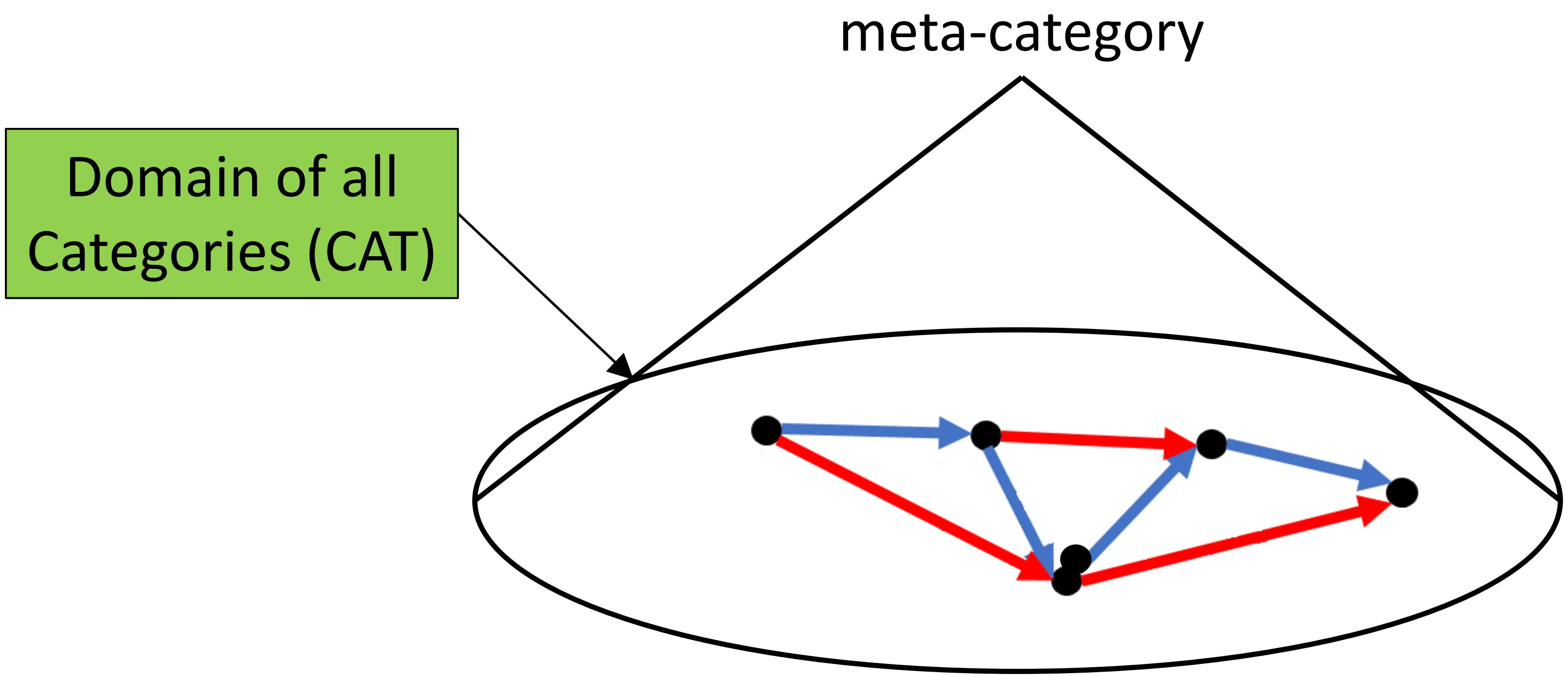
meta-category



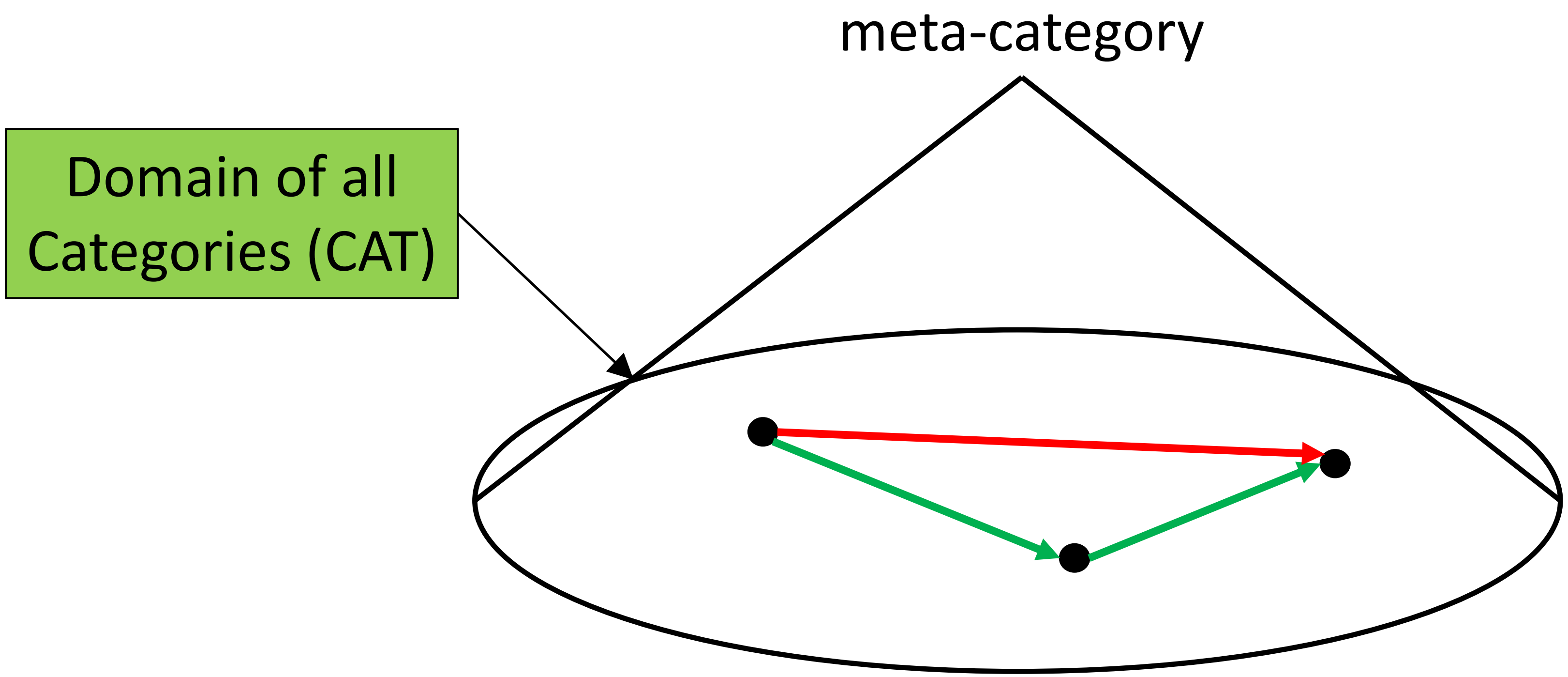
Containment Recursion



Containment Recursion

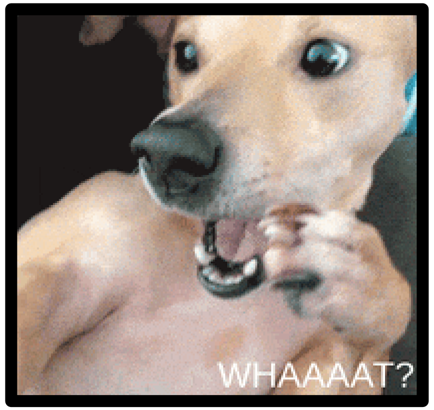


Containment Recursion

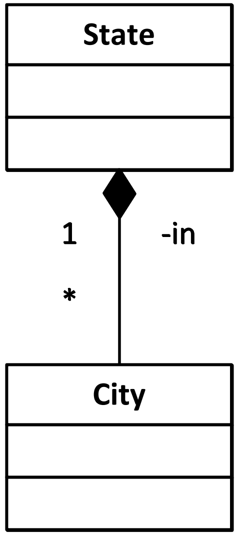


Every level of recursion abstraction in ***CT*** is governed by the same principles/ideas

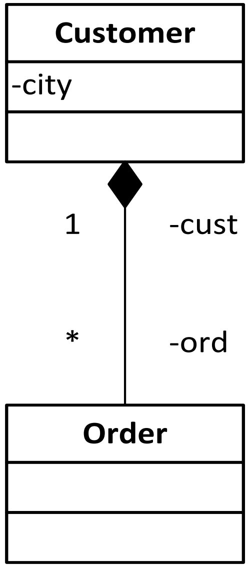
This occurs???



- Yes! Stacked layer architectures is an example…
- Aside: Containment recursion does not exist in UML class diagrams
- Who does this?? Java since 1997 allows class nesting… I used it in building software product lines



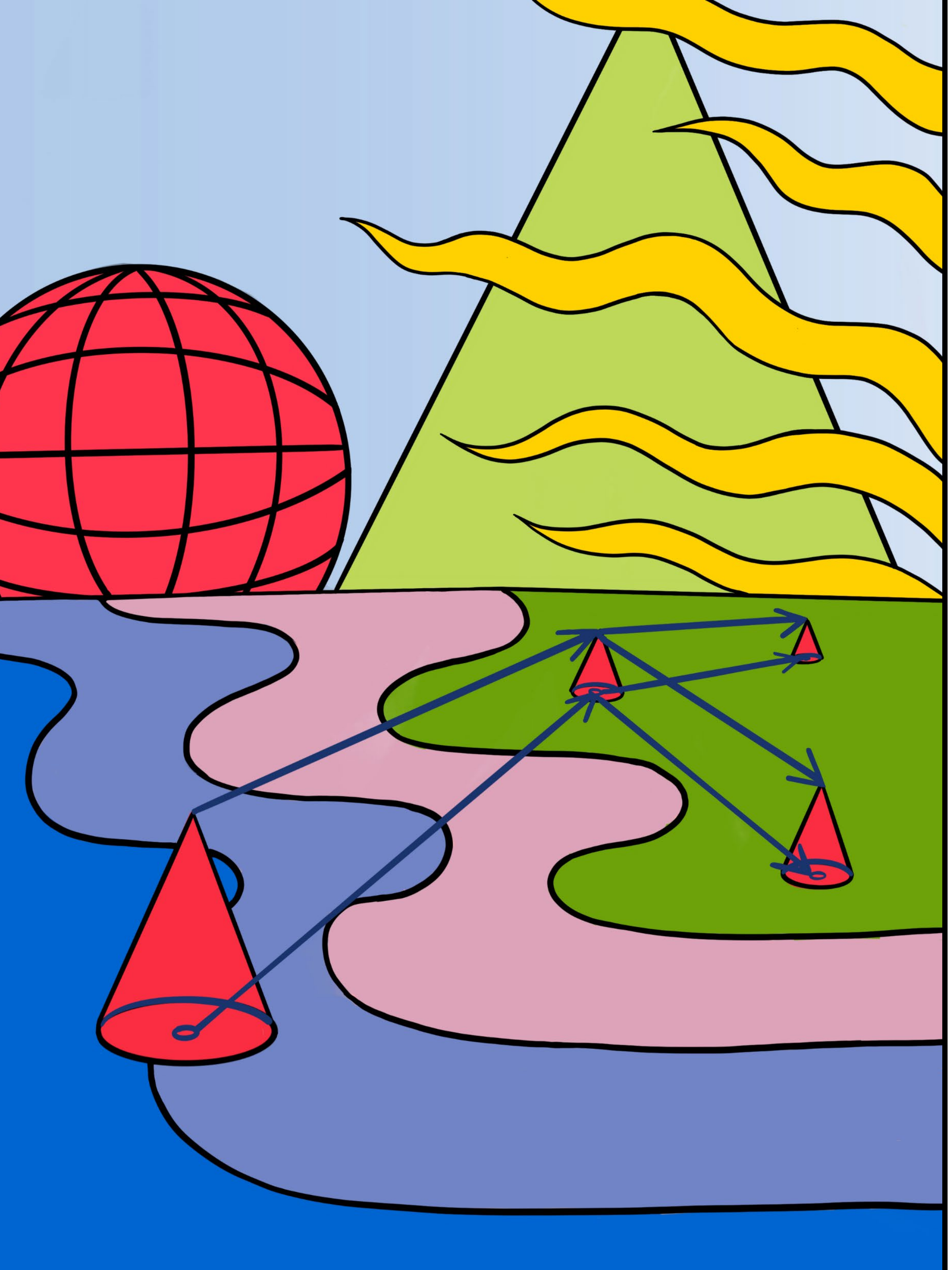
is inside →




```
class Customer {
   static class State { ... }

   static class City {
      State in; ... // association
   }

   // Customer members and methods
}
```

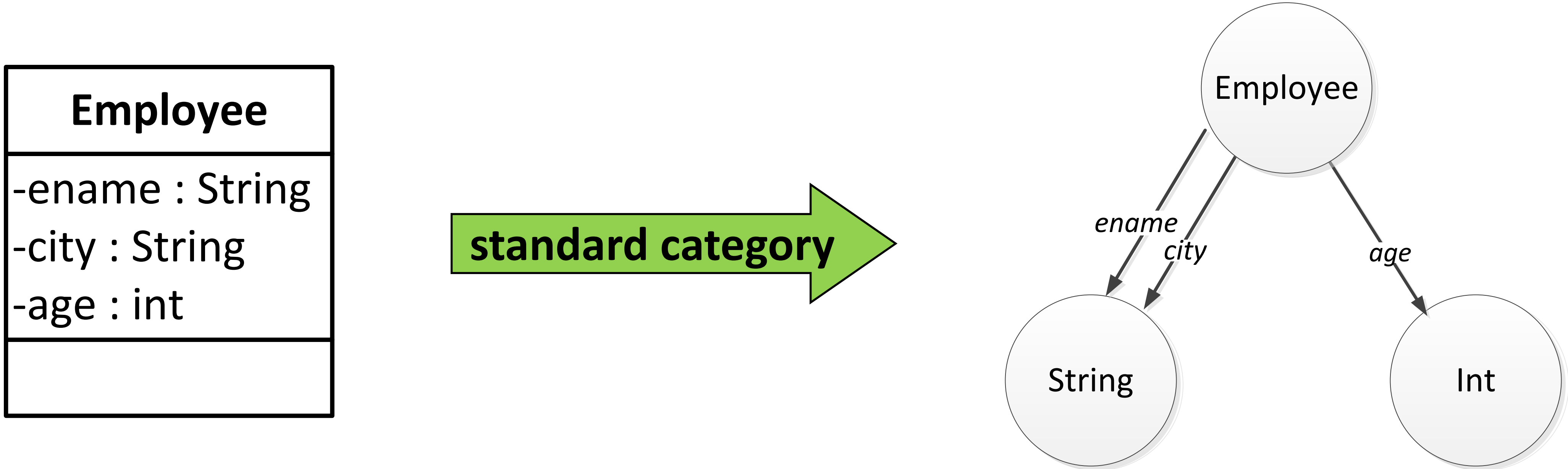


Structures that don't seem to have computations but really do

Example

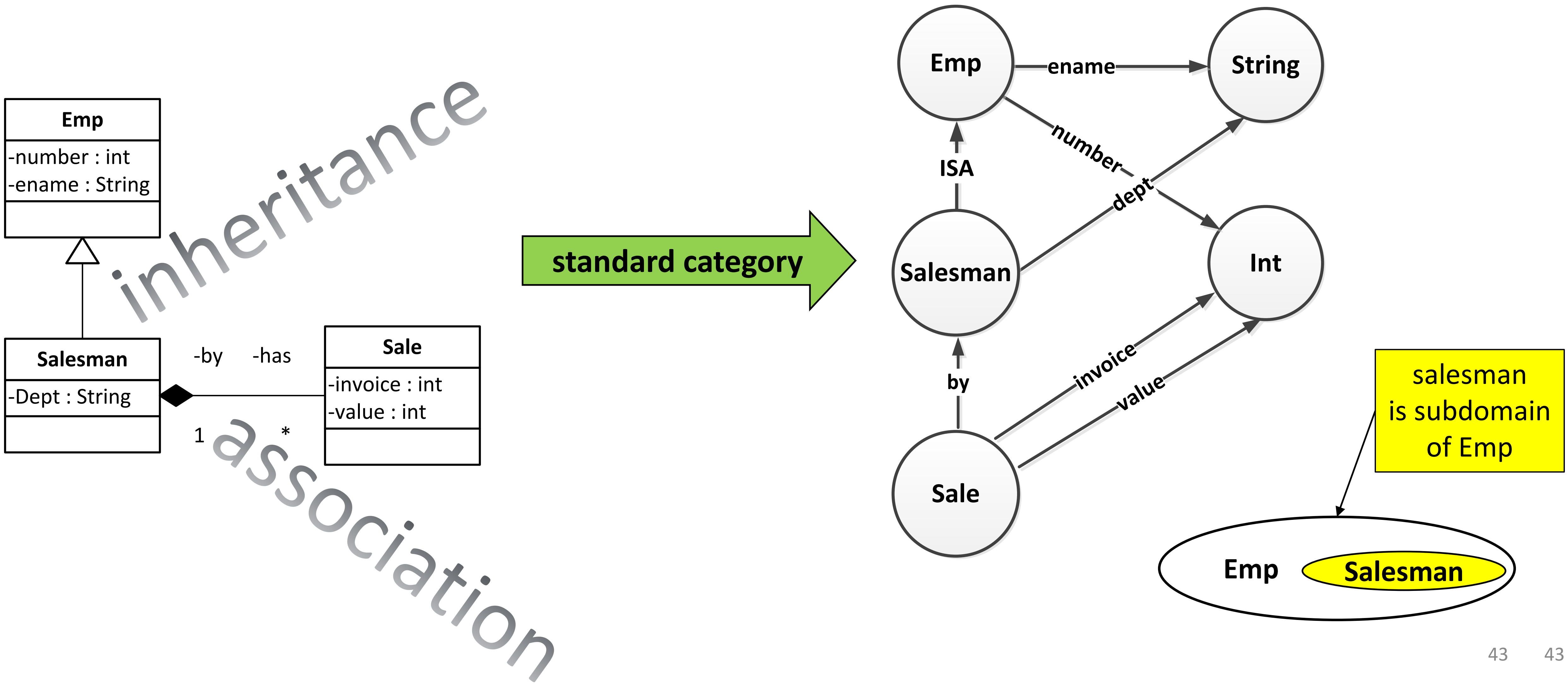
- Consider a trivial UML class diagram:
  - Employee has a name, age and lives in a city
  - Given an employee instance, what is his/her age?
  - Then what is its category?

This is a simple lookup!
aka a computation!



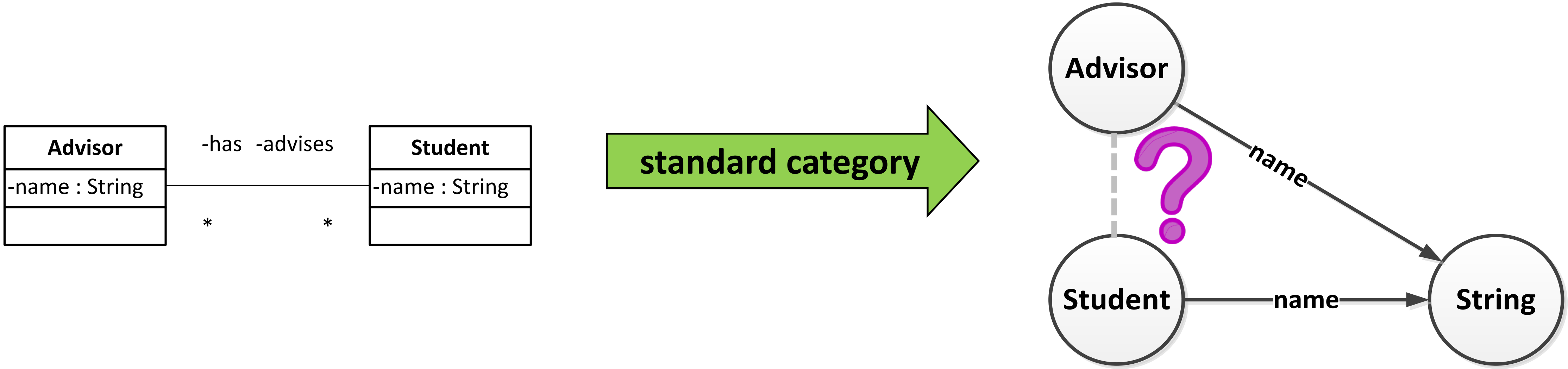
answer

Every Class Diagram has a Standard Category



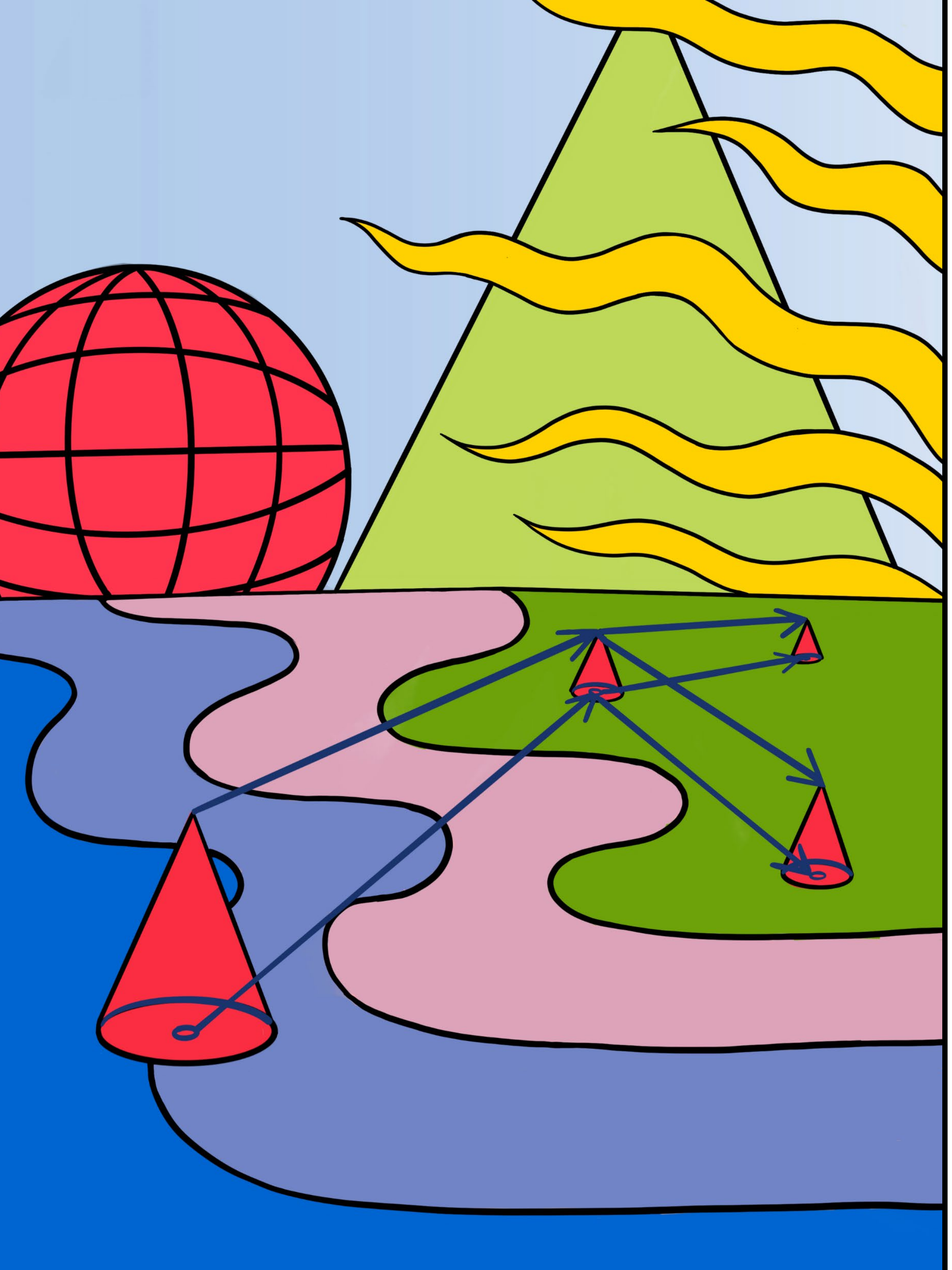
43

Every Class Diagram has a Category



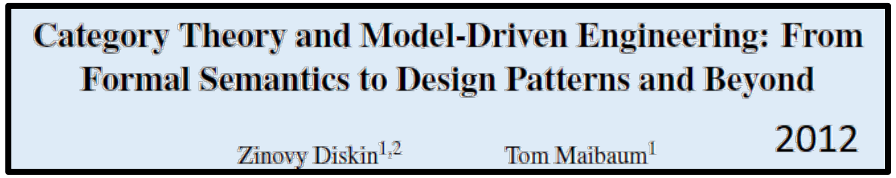
Answer in Lecture #2 ☺

 44



Some Well-Known Category Constructions

aka Design Patterns – common constructions in information designs…


Category Theory and Model-Driven Engineering: From Formal Semantics to Design Patterns and Beyond

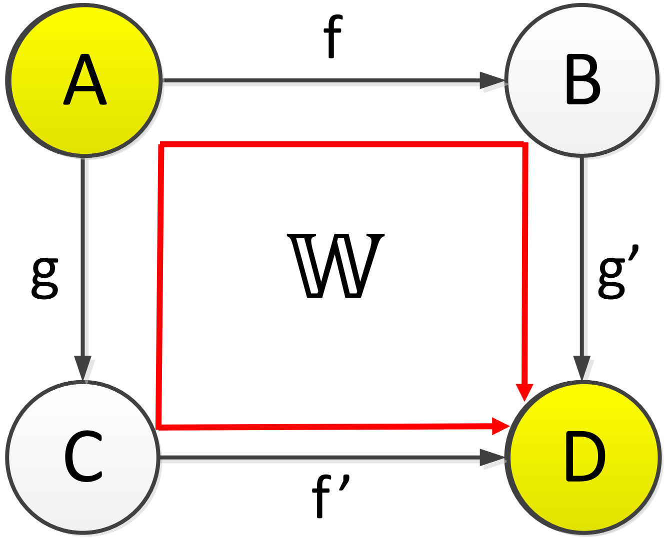
Zinovy Diskin[1,2] Tom Maibaum[1]

2012

most important Commuting Diagrams

- An external diagram 𝕎 **commutes** if for every pair of domains in 𝕎, say A and D, all directed paths from A to D yield the same result

To say a diagram commutes is a strong constraint!



$g' \cdot f \; = \; f' \cdot g$

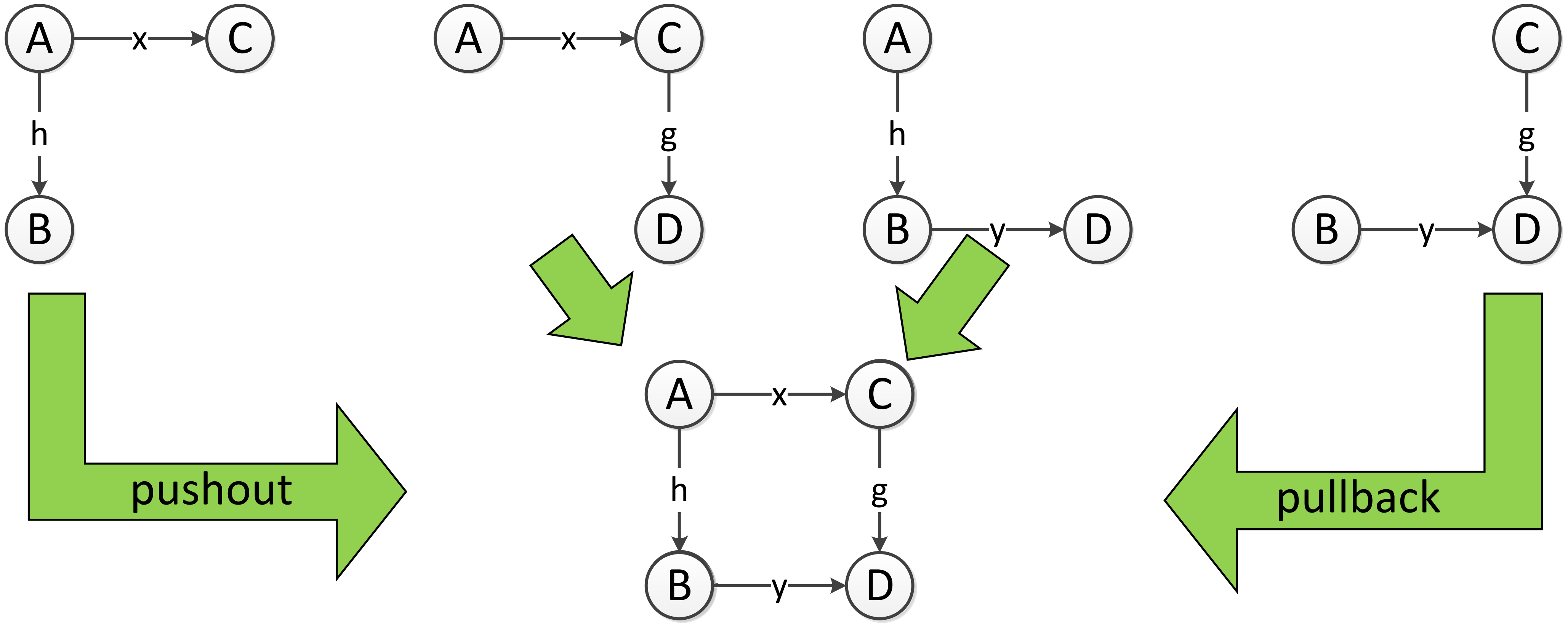
$g' \cdot f \; \neq \; f' \cdot g$

- Constellation of domains does NOT IMPLY its diagram commutes

You must decide if 𝕎 commutes

Hint about Commuting Diagrams

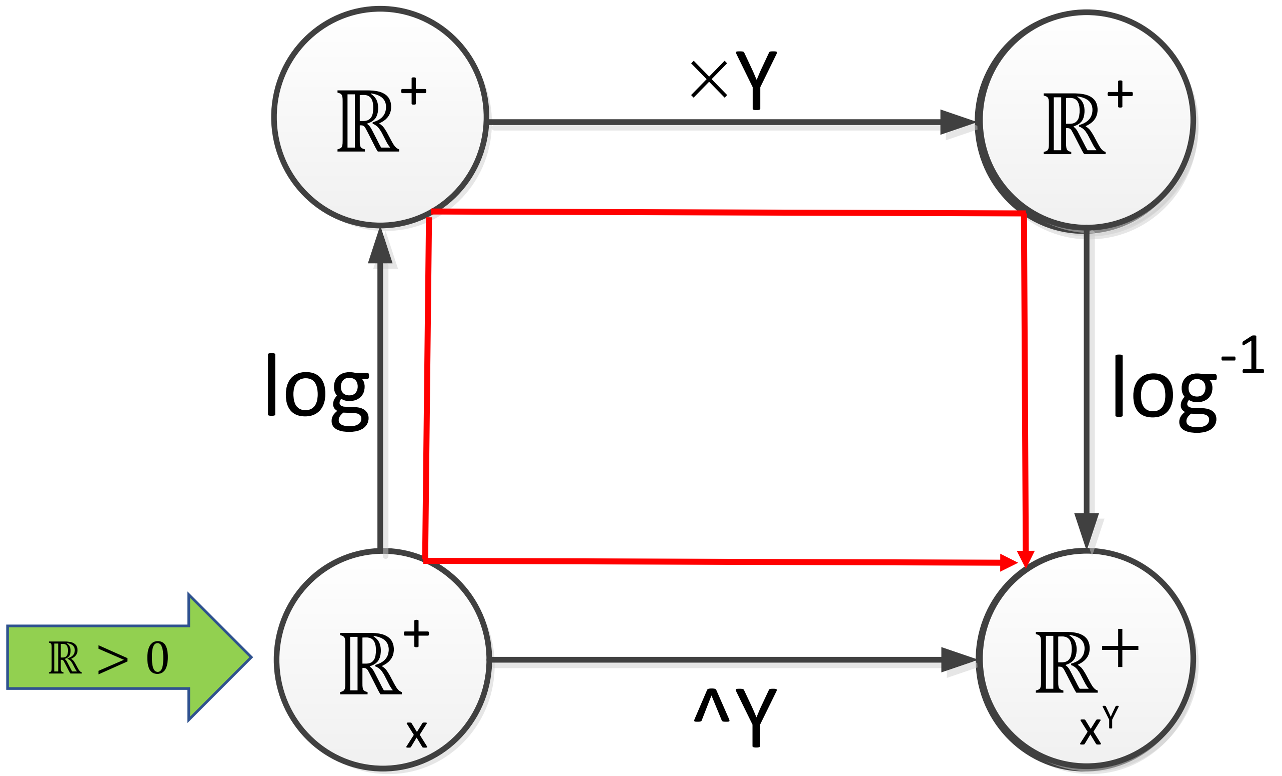
- If you see any of these configurations:



- Complete the diagram to make it commute

Example #1 of Commuting Diagrams: Logarithms

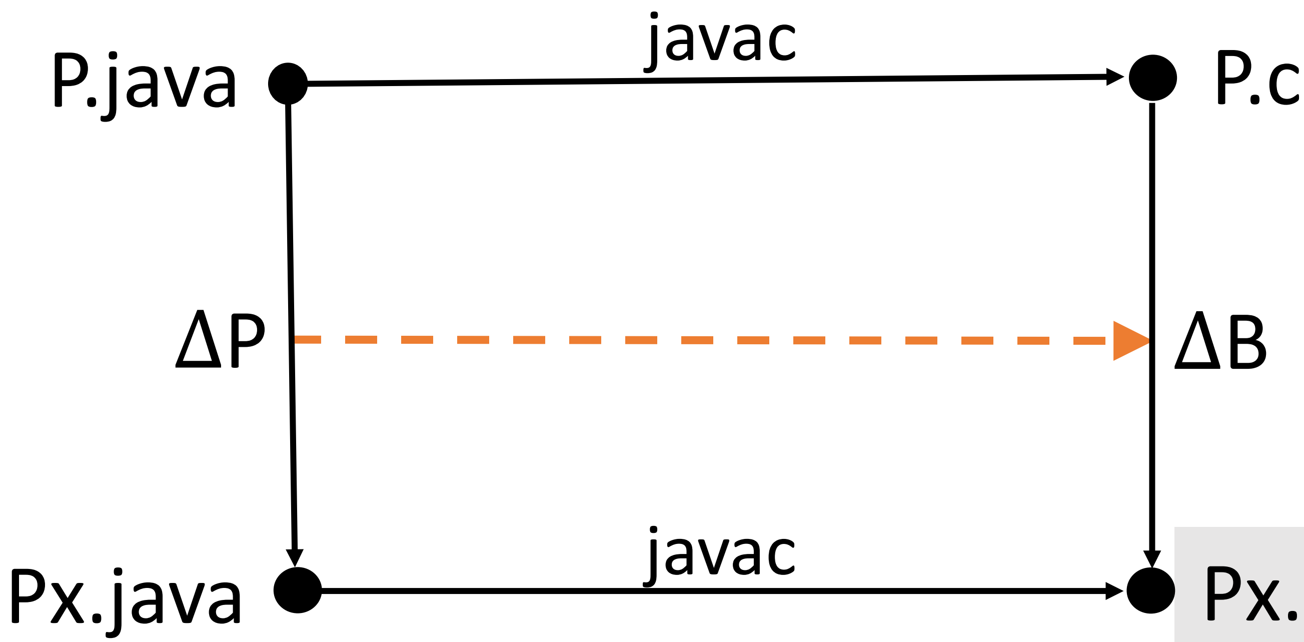
- From high-school mathematics and using slide rules… compute X^Y:



$$X^Y = 10\text{^}(\mathrm{Y} * \log_{10} X)$$

Example #2 of Commuting Diagrams: Javac and Extension

inject source code ΔP



compile ΔP inject bytecode ΔB into P.class

Px.class = ΔB • P.class

Compositional Compilation for Java-like Languages through Polymorphic Bytecode

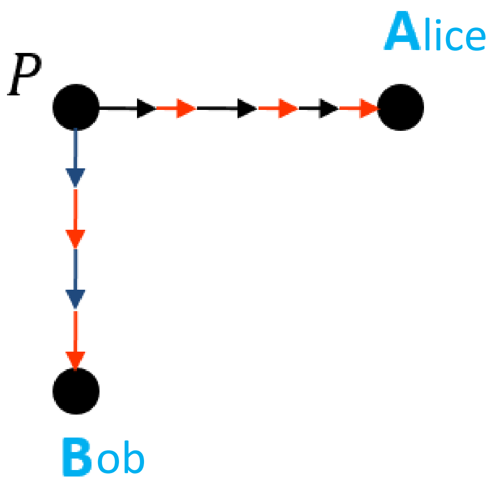
extended version of the paper appearing at POPL 2005

Davide Ancona
Ferruccio Damiani
Sophia Drossopoulou
Elena Zucca

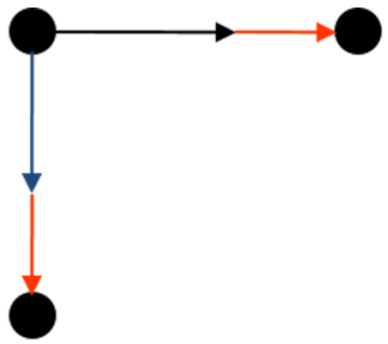
2005

Example #3 of Commuting Diagrams: Version Control

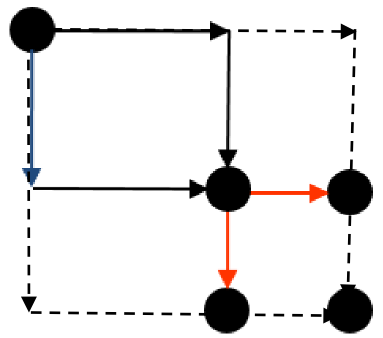
conflicting edit → nonconflicting edit



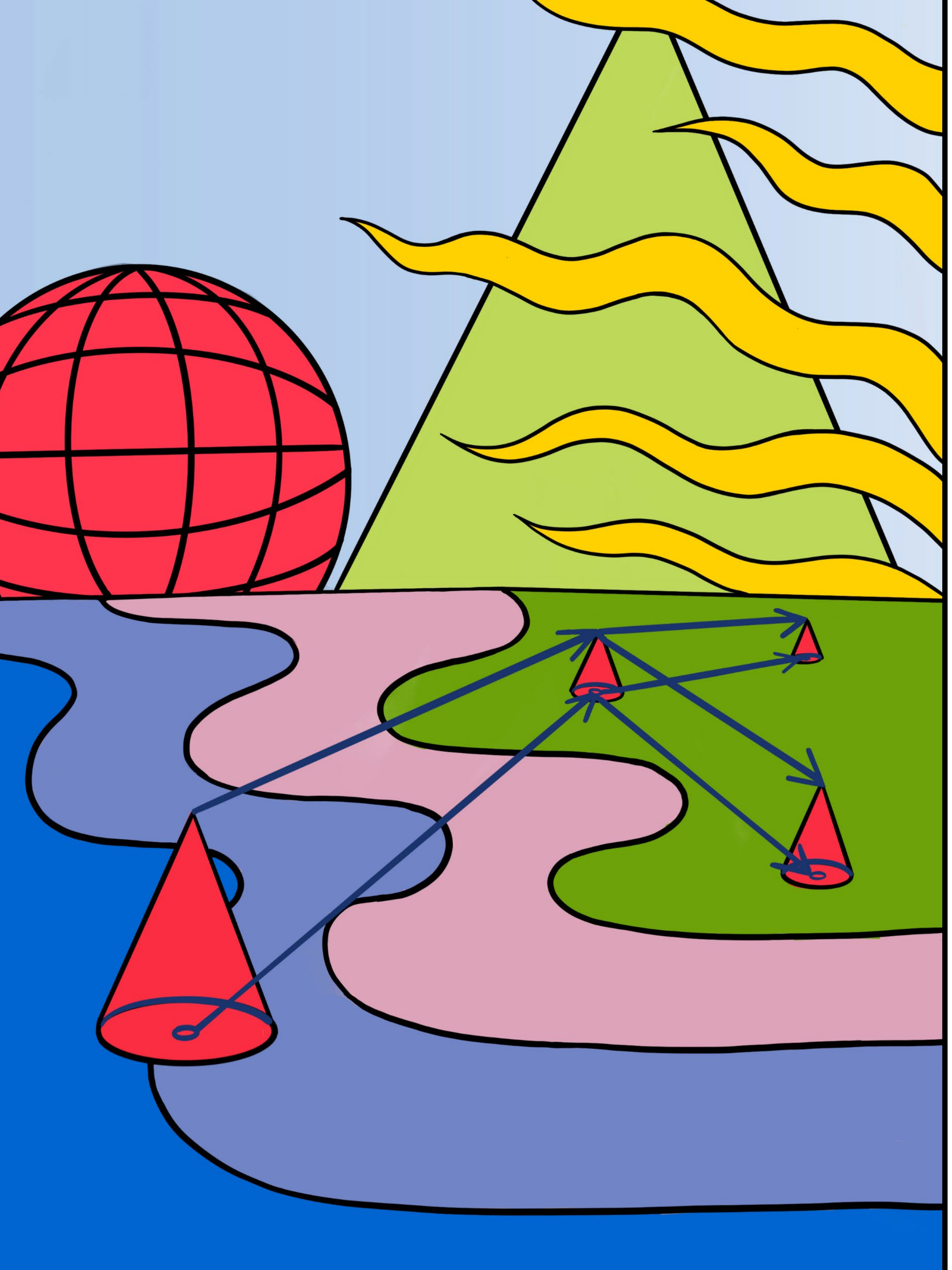
Two programmers **A**&**B** checkout program P and make their edits



A checks-in her changes, **B** updates his copy, VCS rearranges both edits



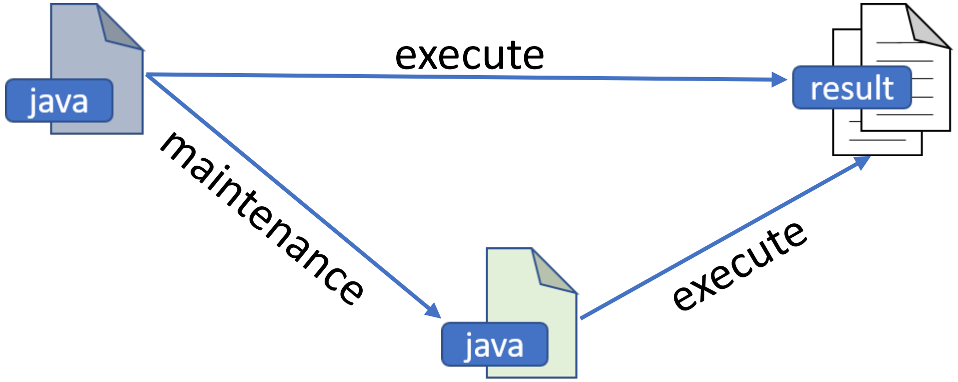
VCS computes pushout (union) of non-conflicting edits. Asks **B** to resolve his conflicting changes manually to complete pushout



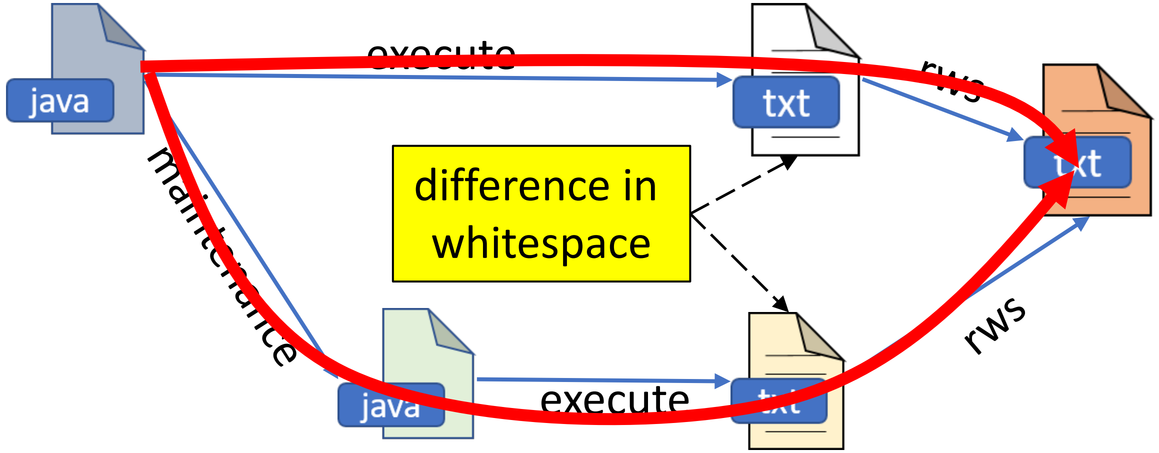
Testing Design Patterns

Testing

- Regression Testing discussed earlier
  - this diagram commutes

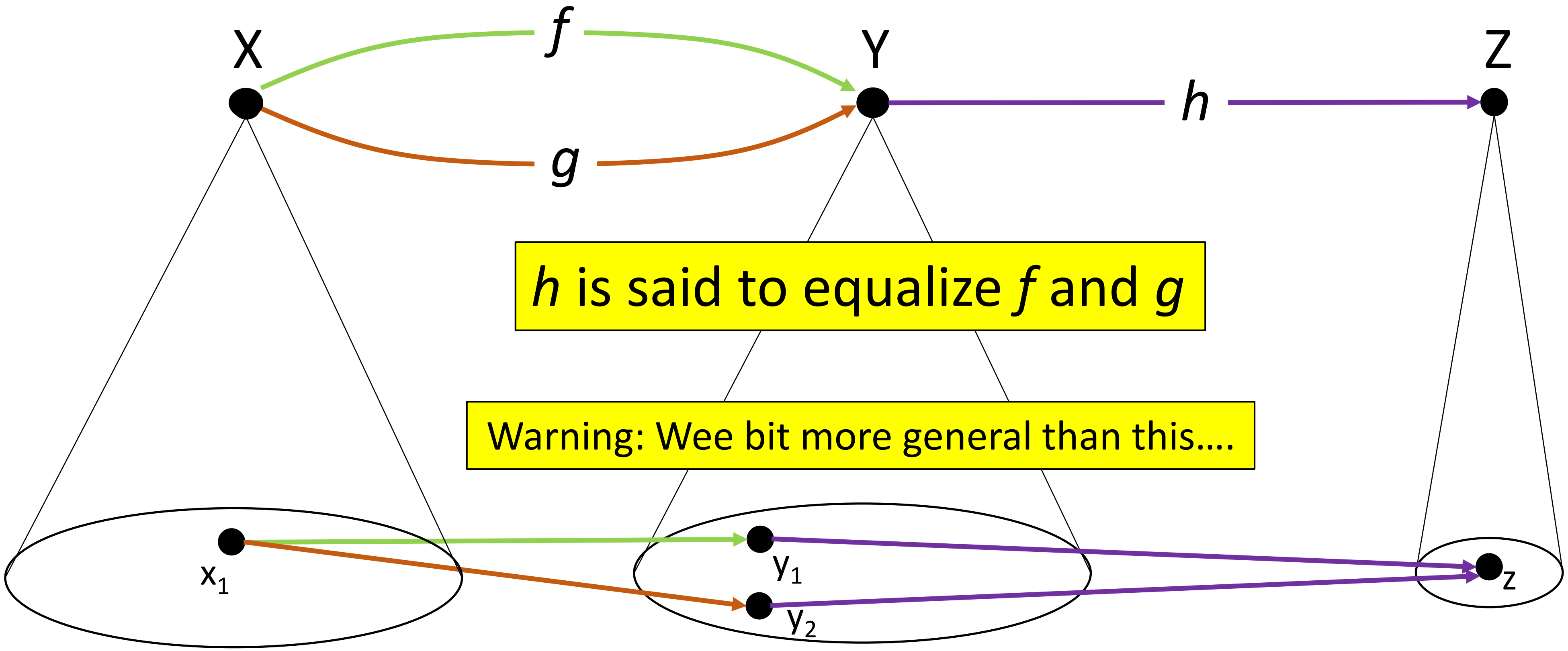


- Variation I encountered:



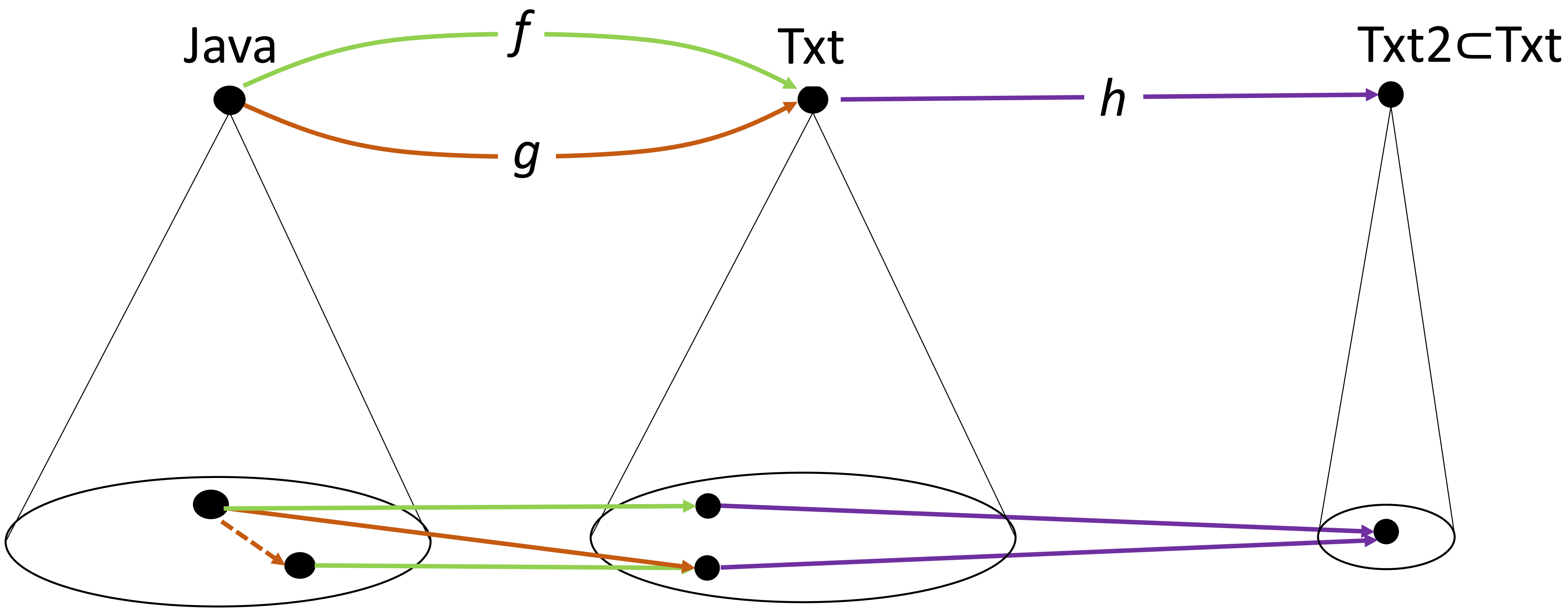
- Instead of changing my test output
- I added an arrow “removeWhiteSpace” to my regression test
  - this diagram commutes

CT Construct: Co-Equalizer



$$h \bullet f = h \bullet g$$

Mine is a Special Case of Co-Equalizer

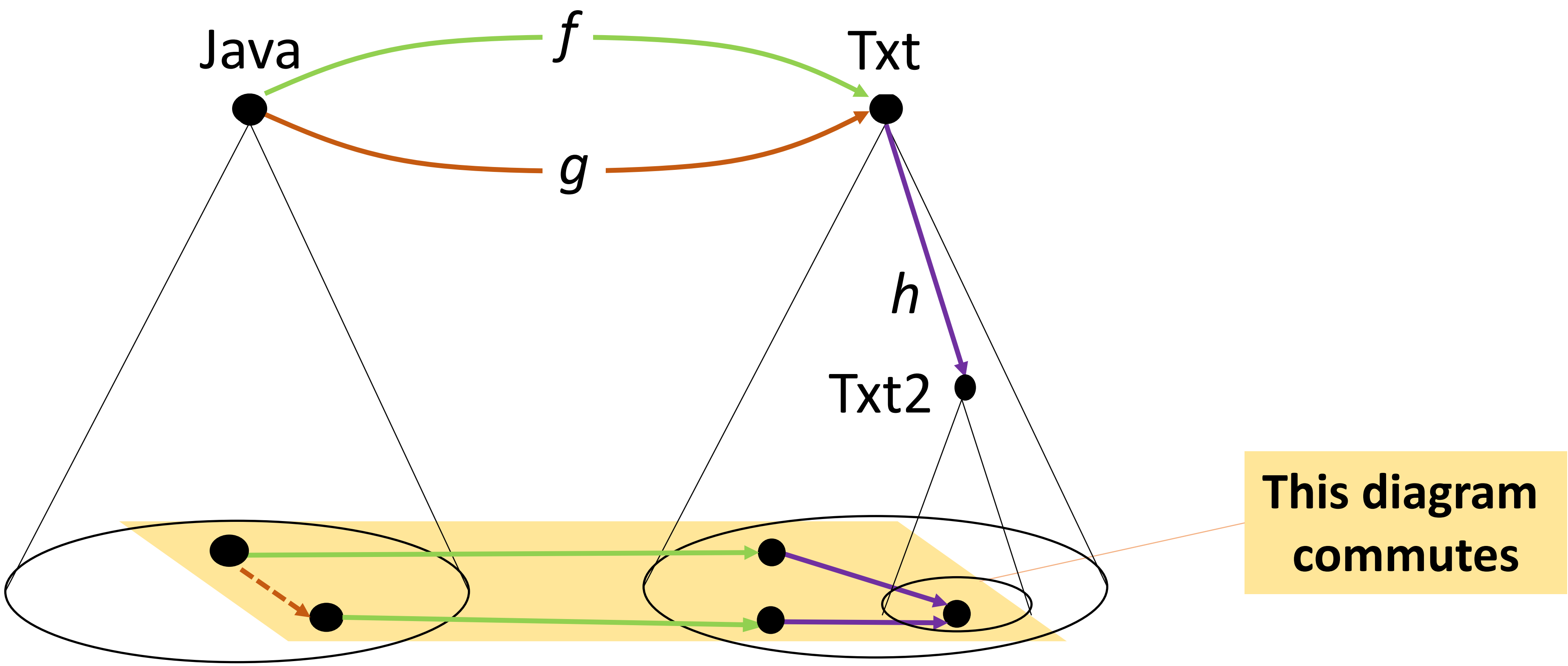


Co-Equalizer

$$h \bullet f = h \bullet g$$

How I “see” this in my mind…

Mine is a Special Case of Co-Equalizer

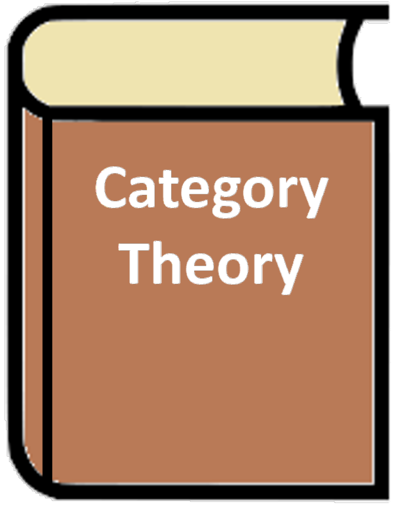


Co-Equalizer

$$h \bullet f = h \bullet g$$

Why ***CT*** is Intimidating

- I asked my Ph.D. students to read a ***CT*** text:

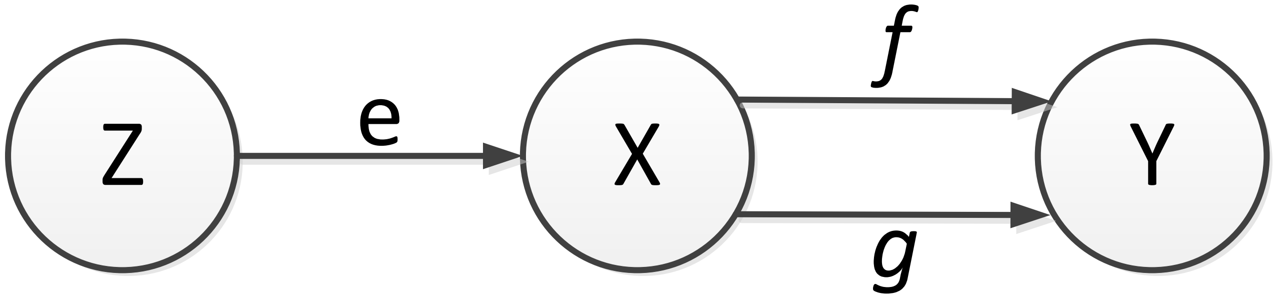


- Reason: Abstract mathematics with examples from mathematics
- With examples from SE, CT ideas do become alive… But

Many ***CT*** Design Patterns

- I can't translate into a practical example

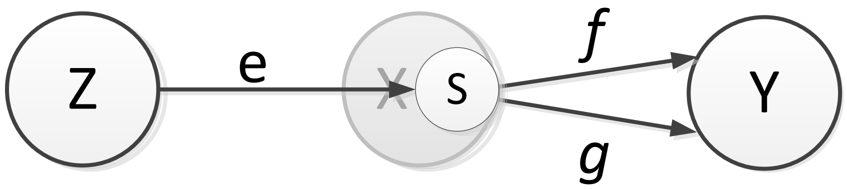
Equalizer



"*e*" equalizes arrows *f* and *g*

$$\forall z \in Z:\ f \cdot e(z) = g \cdot e(z)$$

find a function
e : Z→S



find subdomain $S \subset X$
s.t. $\forall p \in S: f(p) = g(p)$

Many ***CT*** Design Patterns

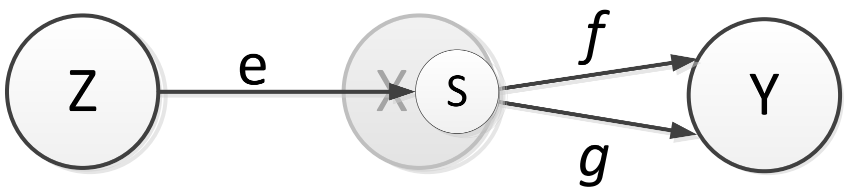
- I can’t translate into a practical example

Equalizer

“*e*” equalizes arrows *f* and *g*

Many *CT* Constructions (Design Patterns) are like this

find a function
$e : Z \to S$



find subdomain $S \subset X$
s.t. $\forall p \in S: f(p) = g(p)$

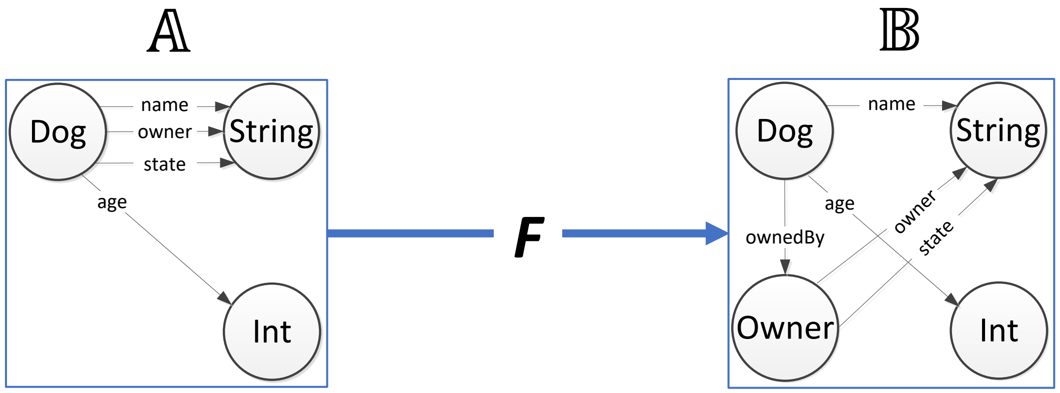
End of Lecture #1

- What you should have learned about ***CT***: it
  - defines computational relationships among structures as functions
  - is declarative, like UML class diagrams
  - is a foundation of MDE and SE
  - expresses common “design patterns” of structures and computations
- Just a wee bit of basics

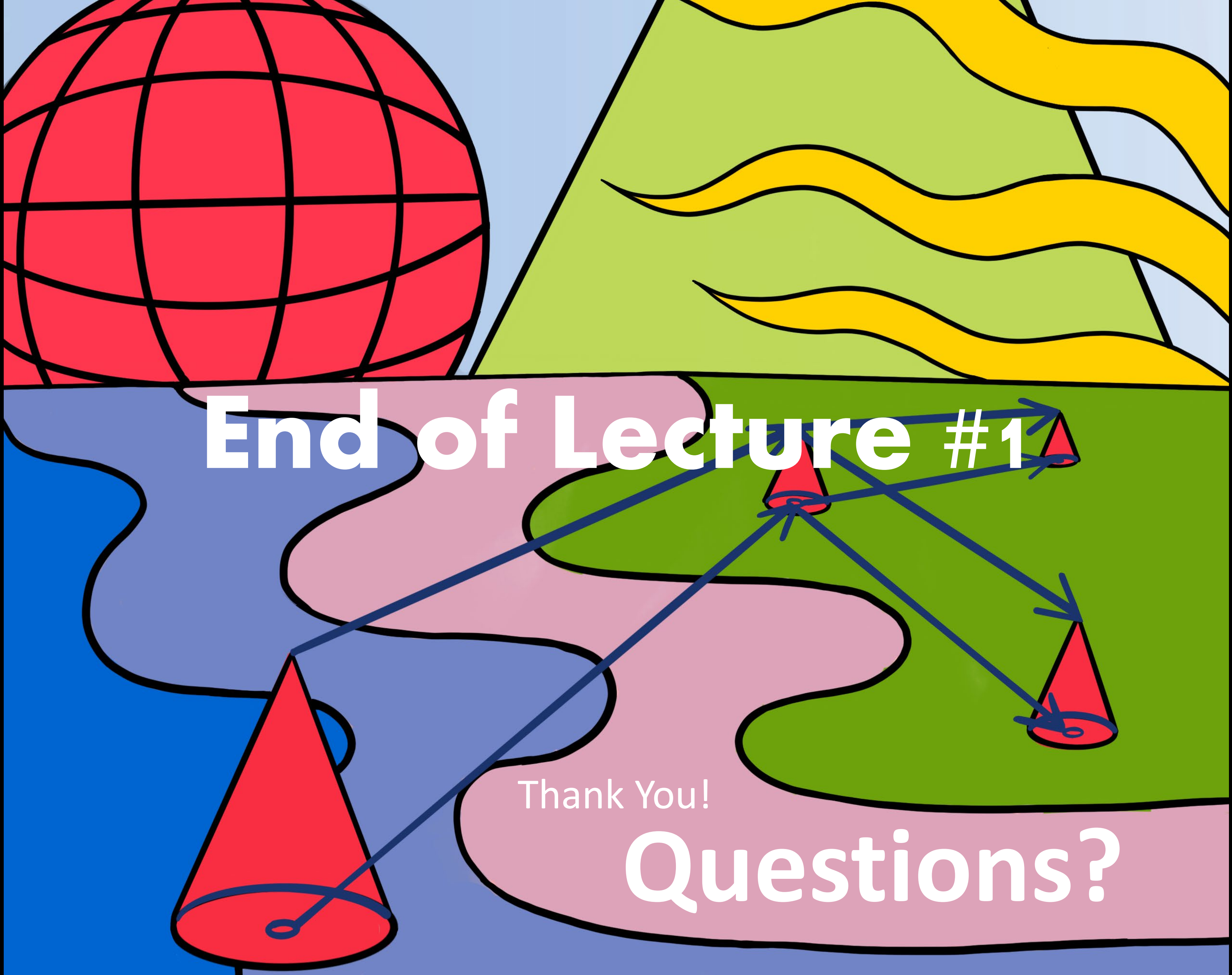
End of Lecture #1

- What you should have learned about ***CT***: it
  - defines computational relationships among structures as functions
  - is declarative, like UML class diagrams
  - is a foundation of MDE
  - expresses common “design patterns” of structures and computations
- Just a wee bit of basics

- What is in Lecture #2? ***Functors***
  - A ***functor*** is an arrow between categories $F : \mathbb{A} \rightarrow \mathbb{B}$
  - Simple but non-obvious
  - Brilliant idea
  - Where almost all “action” in ***CT*** resides



End of Lecture #1



Thank You!

Questions?