A View of the Past and Future of Objects

William Cook
UT Austin

Dahl-Nygaard Prize lecture
ECOOP 2014

- History
- Enterprise Applications
- Models & Synthesis
- Ensō

OO
came from
simulation

creating & running models

grocery store
airplane
business
user interface
OO class
grammar

behavior

+

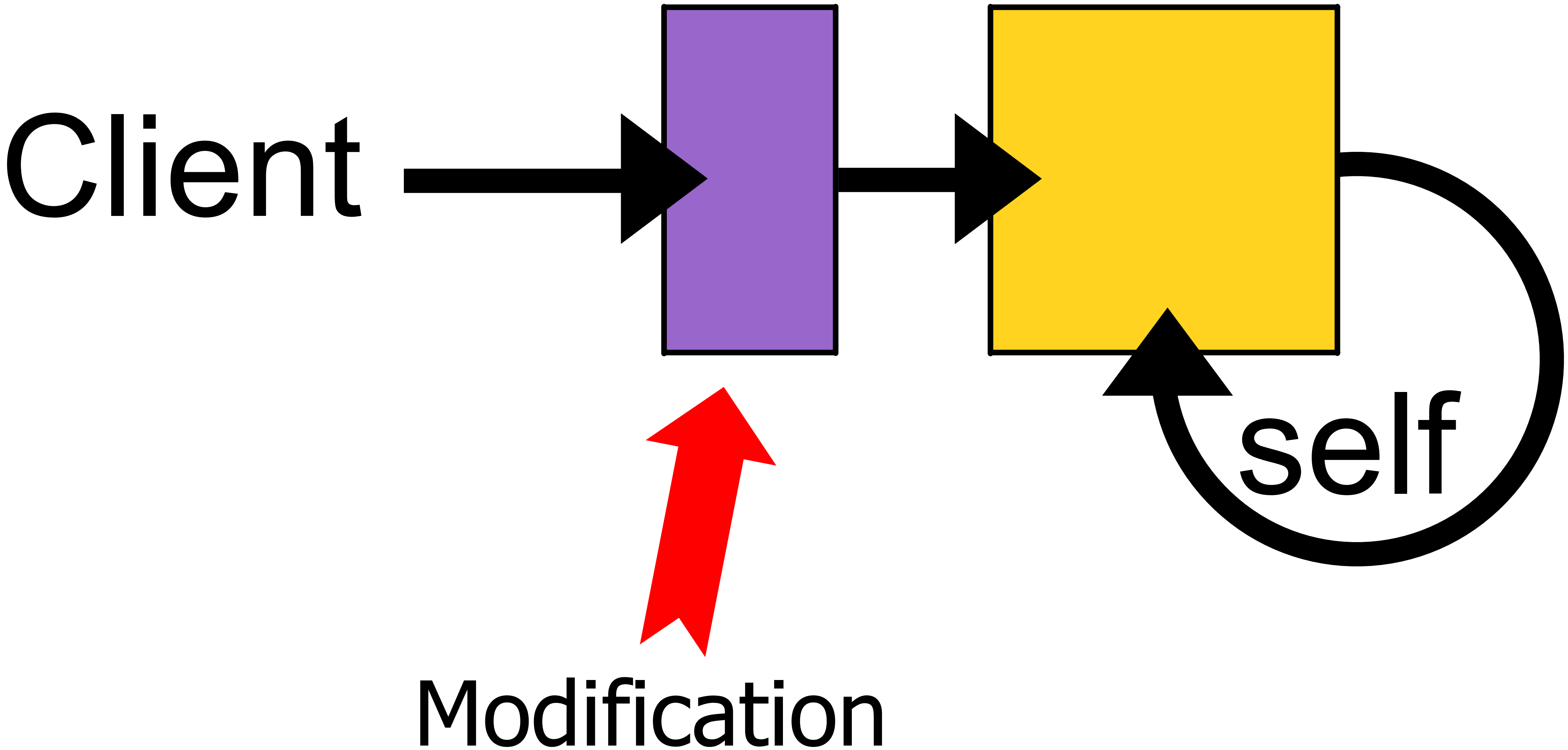
state

A modern definition:

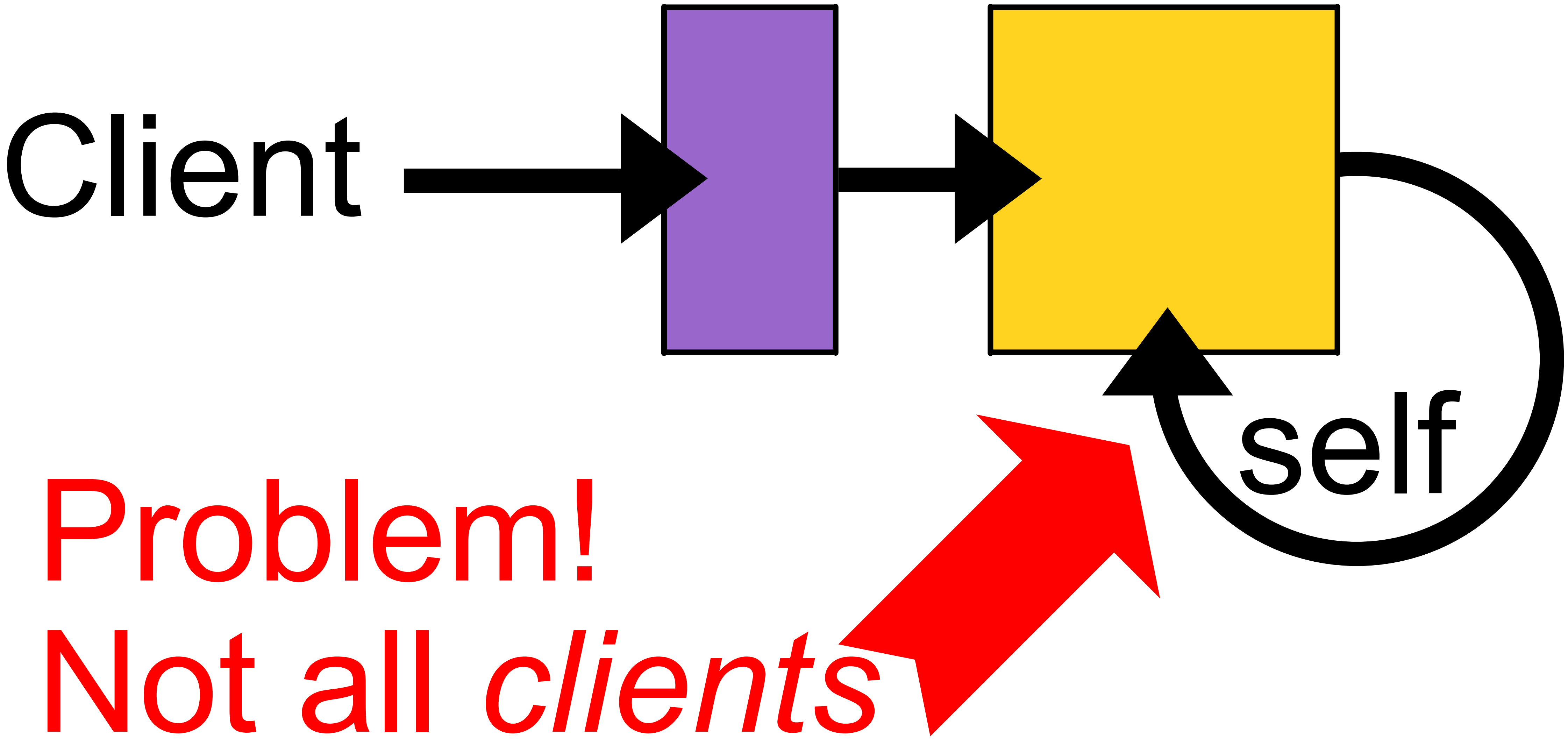
An ***object*** is a first-class, dynamically dispatched behavior.

Inheritance, state, identity, classes are useful but not essential.

Modification and Self-reference

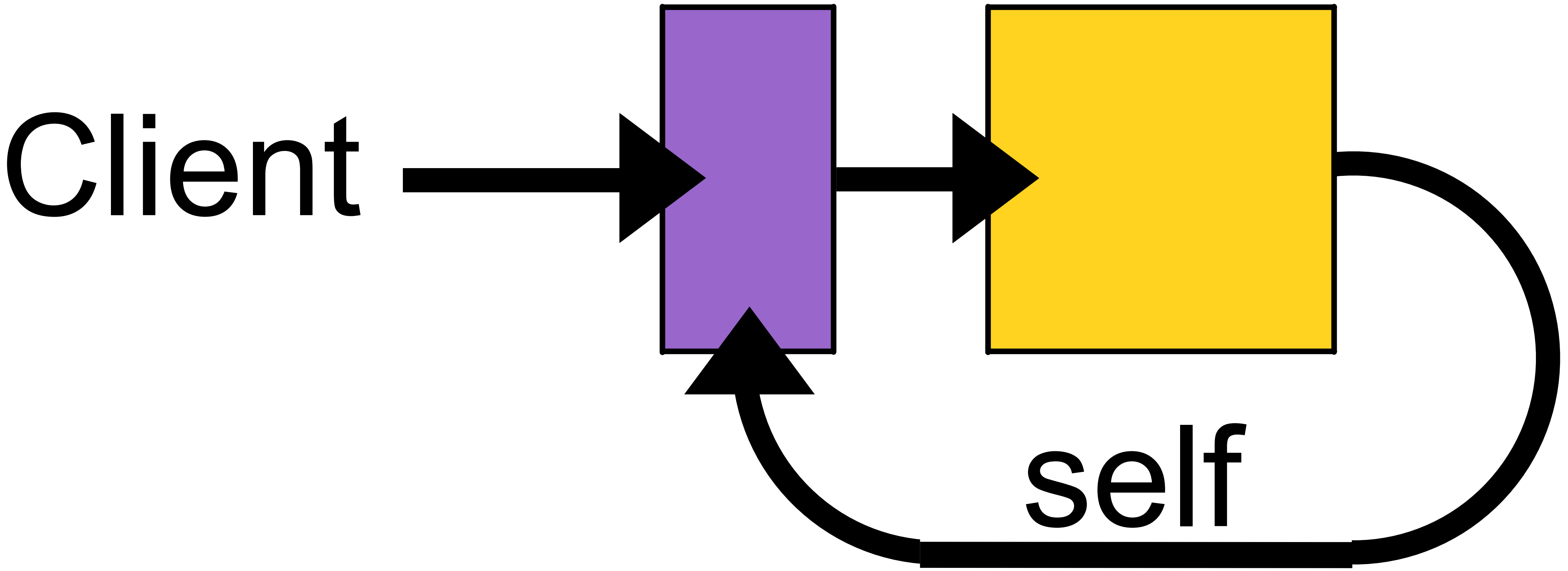


Modification and Self-reference



Problem!
Not all *clients*
are modified

Inheritance



Inheritance:
"Consistently modify a recursive definition"

Killer Application of OO:

GUI frameworks

write classes

instantiate them

to **create** a GUI model

run the model

OO
has
won!

many PL researchers hate objects

NIH

I guess many of them have never programmed a large extensible system

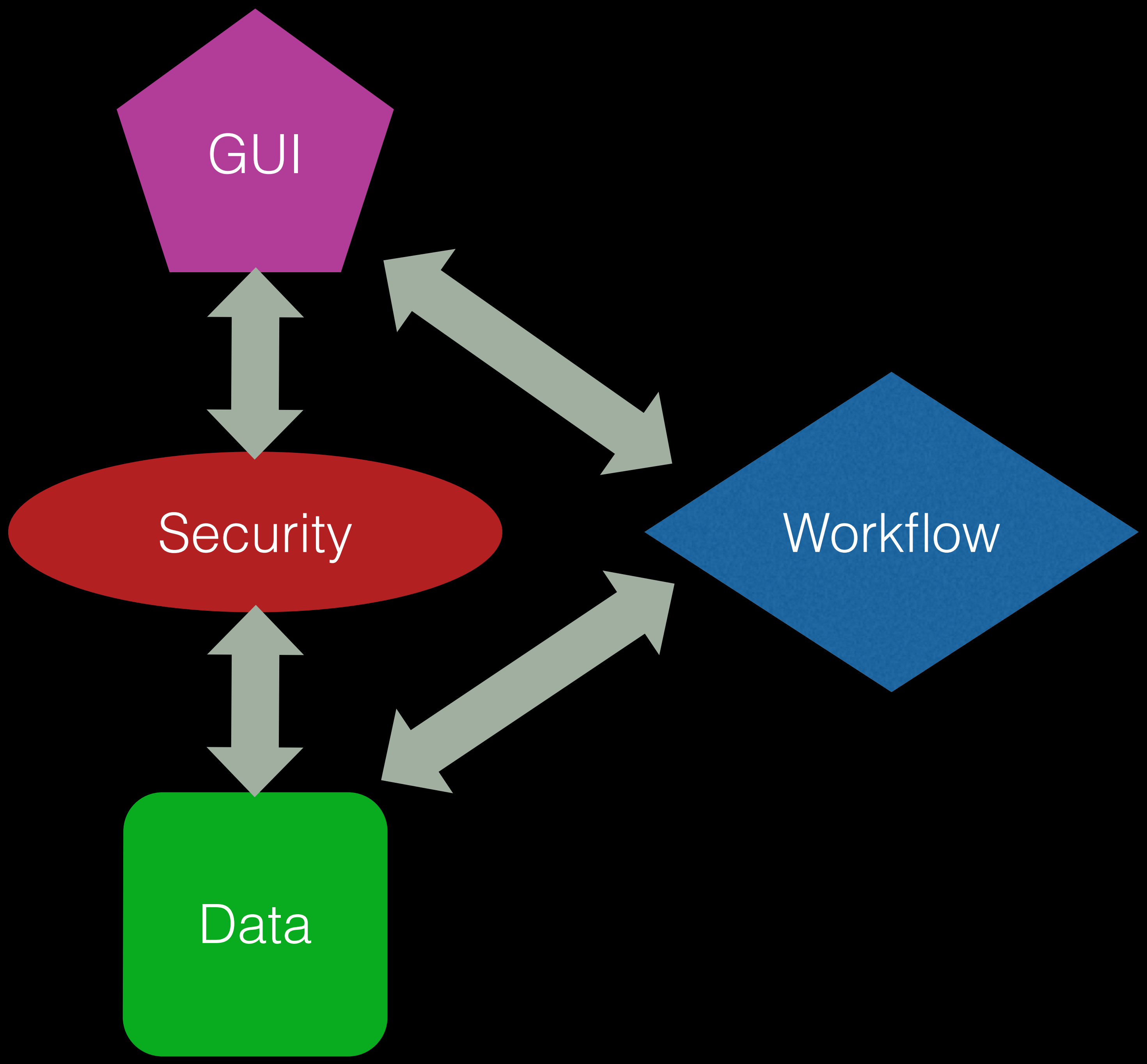
Or if they have, it is a type system or a compiler,
…or financial application

All languages
are
domain specific

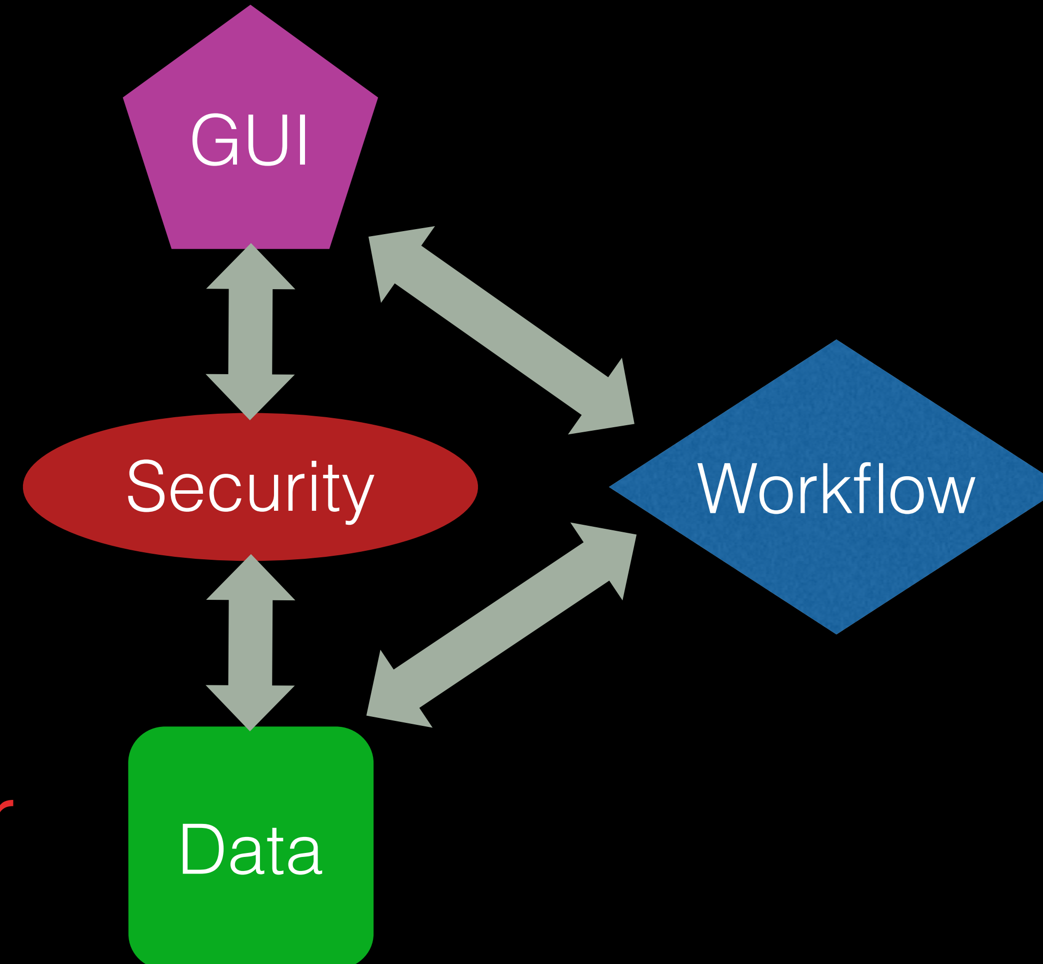
Is OO
best for
everything?

Example

Enterprise Applications


GUI
Security
Workflow
Data

```
box vertical {
 class.name
 for field : class.defined_fields
  if (field.type is Primitive)
   horizontal {
    field.name ":" field.type.name
}
```




```
allow read if grade.student = user
```

```
class Schema
    types: Type*
class Type
    name: string
class Primitive < Type
class Class < Type
    fields: Field*
    super: Type?
```

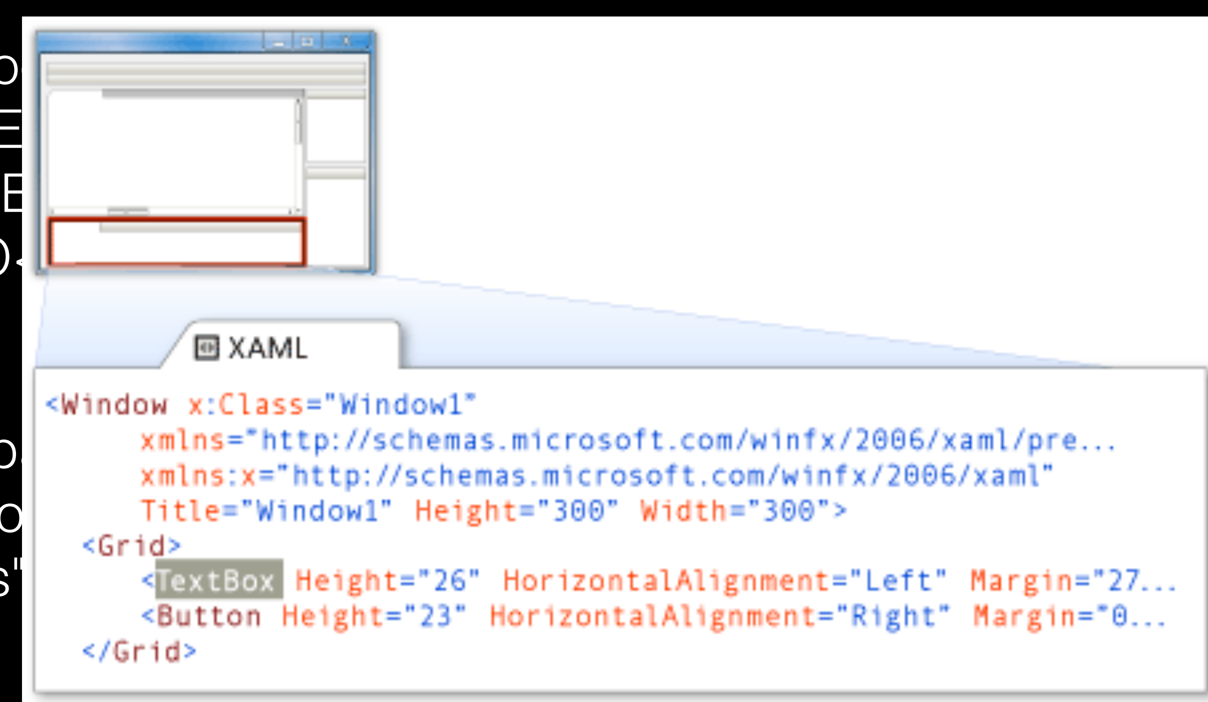
```
on Create Account
  wait(1 month) >> remind
  | cancel
```

Modeling issues

- composites
- multiple languages
- multiple semantics
- optimization
- UML

large models
have many parts

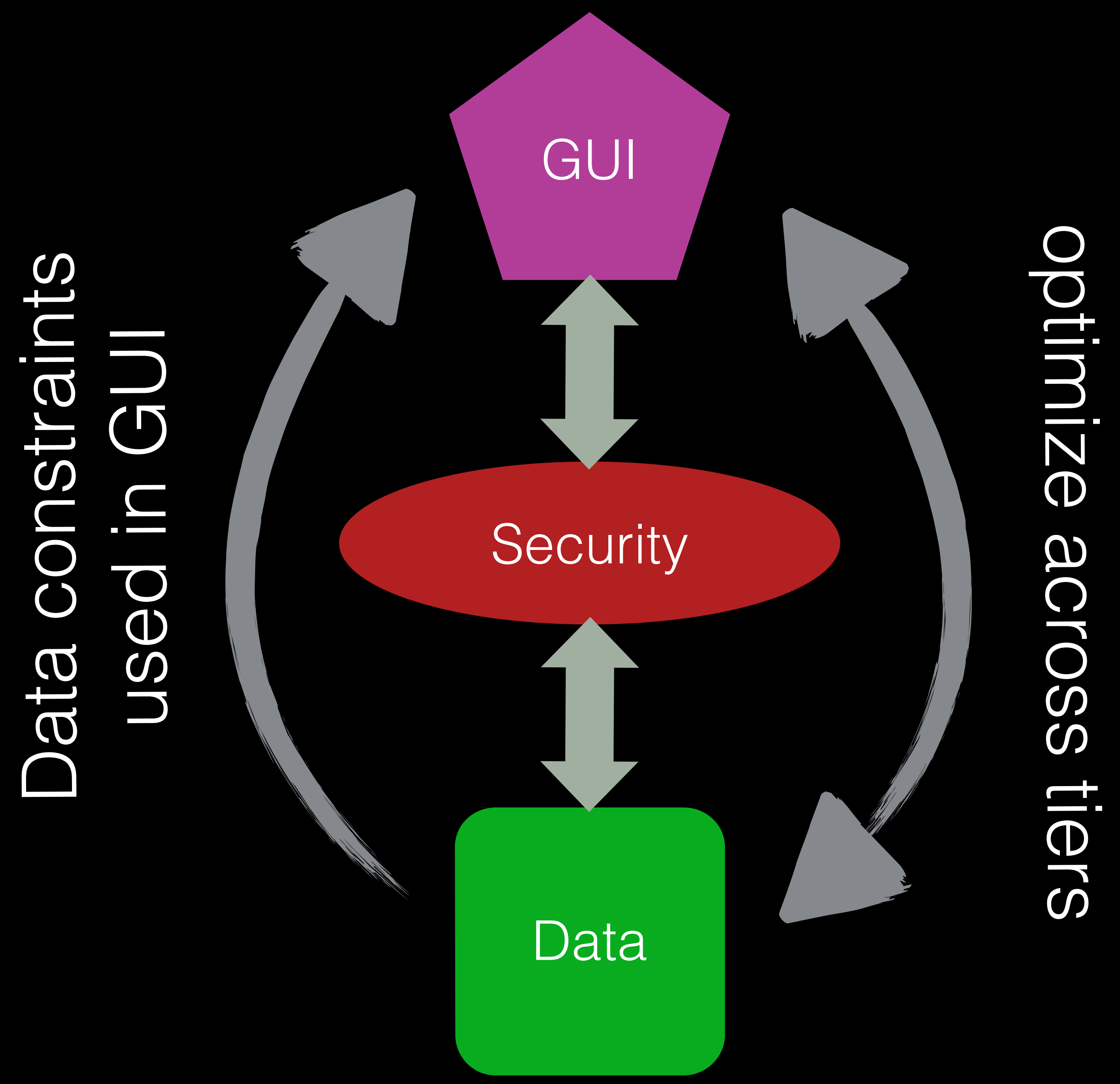
```
<object class="NSWindowTemplate" id="1005">
<int key="NSWindowStyleMask">4111</int>
<int key="NSWindowBacking">2</int>
<string key="NSWindowRect">{{517, 330}, {776, 608}}</string>
<int key="NSWTFlags">544735232</int>
<string key="NSWindowTitle">Window</string>
<string key="NSWindowClass">NSWindow</string>
<object class="NSToolbar" key="NSViewClass" id="726585754">
	<object class="NSMutableString" key="NSToolbarIdentifier">
		<characters key="NS.bytes">994A0CB1-7575-4F39-A65B-7165AB1E8015</characters>
	</object>
	<nil key="NSToolbarDelegate"/>
	<bool key="NSToolbarPrefersToBeShown">YES</bo
	<bool key="NSToolbarShowsBaselineSeparator">YE
	<bool key="NSToolbarAllowsUserCustomization">YE
	<bool key="NSToolbarAutosavesConfiguration">NO
	<int key="NSToolbarDisplayMode">2</int>
	<int key="NSToolbarSizeMode">1</int>
	<object class="NSMutableDictionary" key="NSToolb
		<bool key="EncodedWithXMLCoder">YES</bo
		<object class="NSArray" key="dict.sortedKeys"
```



Model = data instance

not
well supported
in OO
(or PL)

Say It Once


GUI
Security
Data
Data constraints used in GUI
optimize across tiers

UML
models
Object-Oriented Design

UML
models
Object-Oriented Design

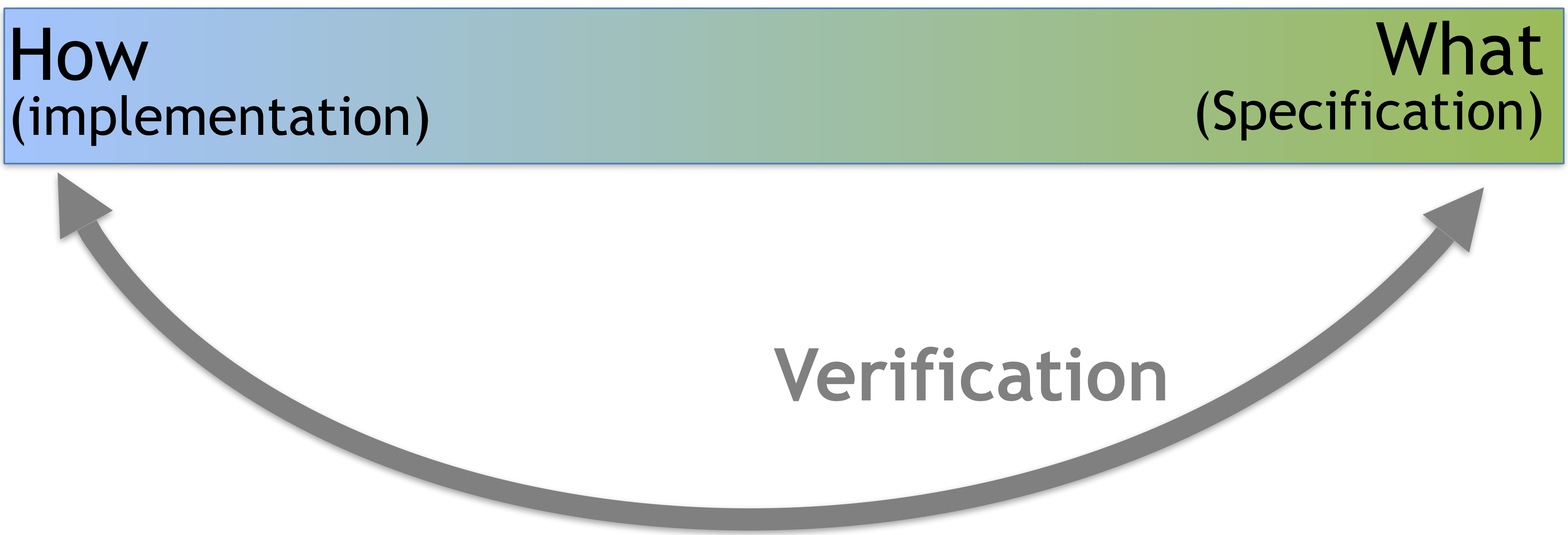
Wrong thing!

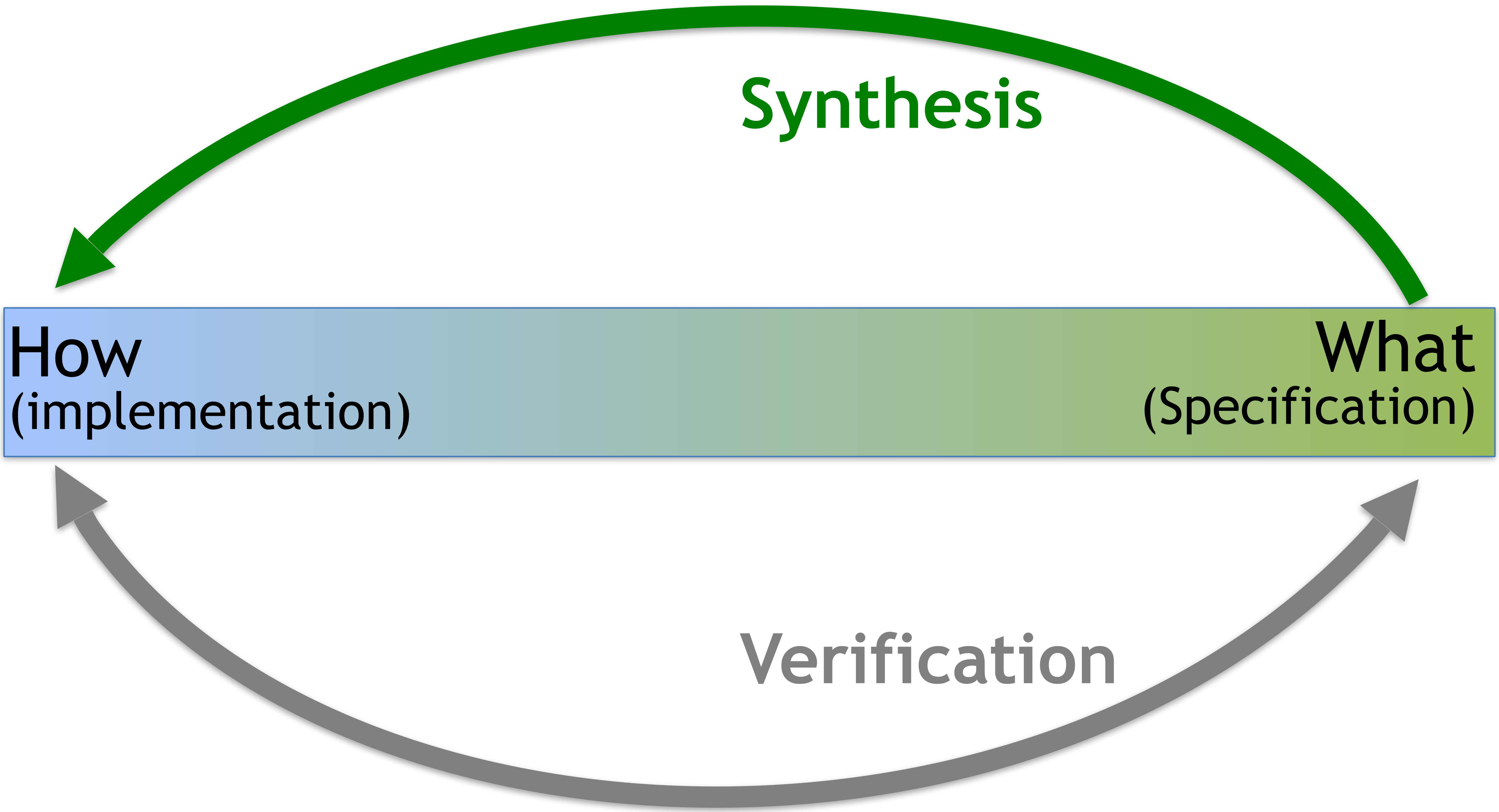
Should model “solution to problem”,
not “OO design of program that solves problem”

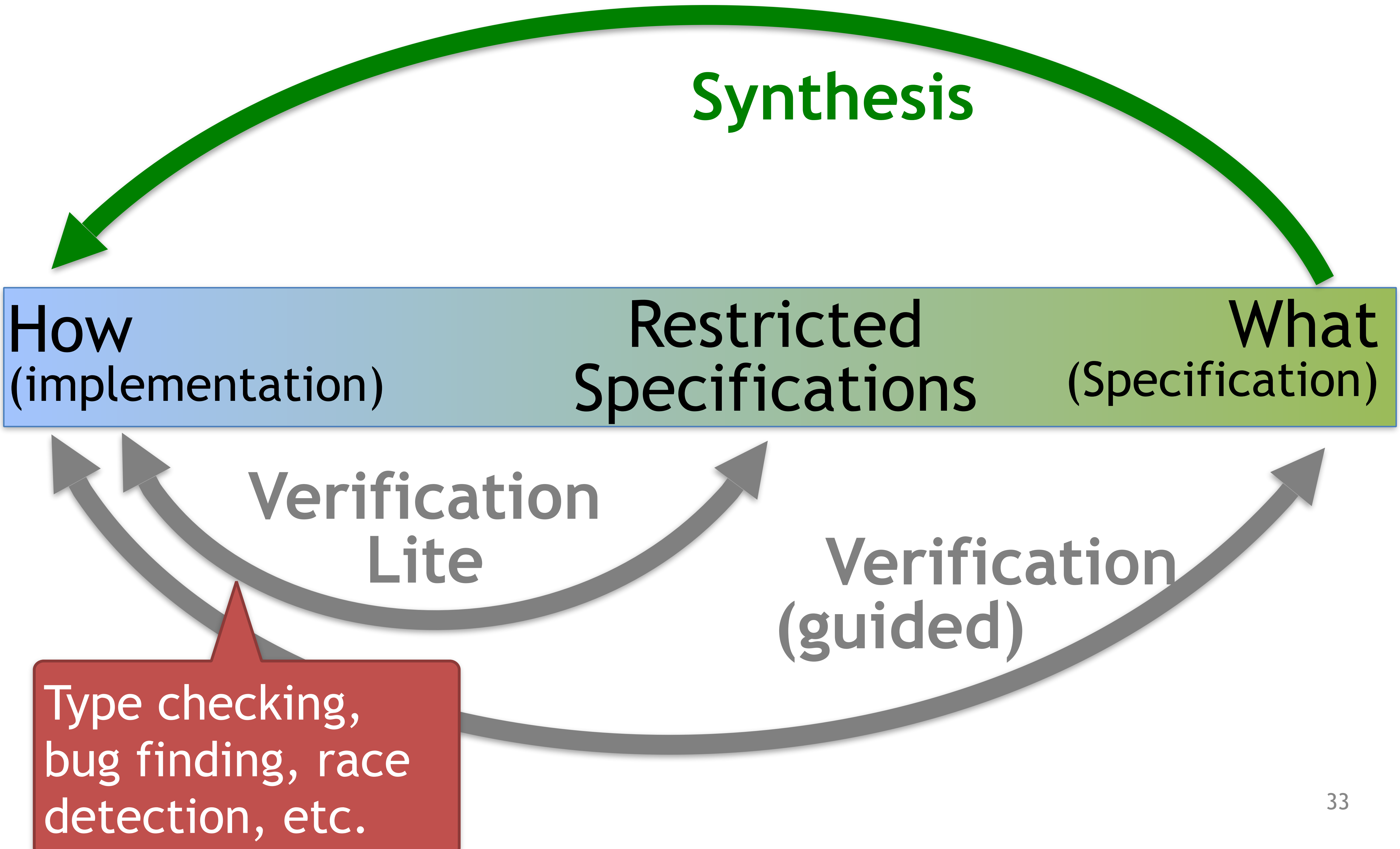
Models & Synthesis

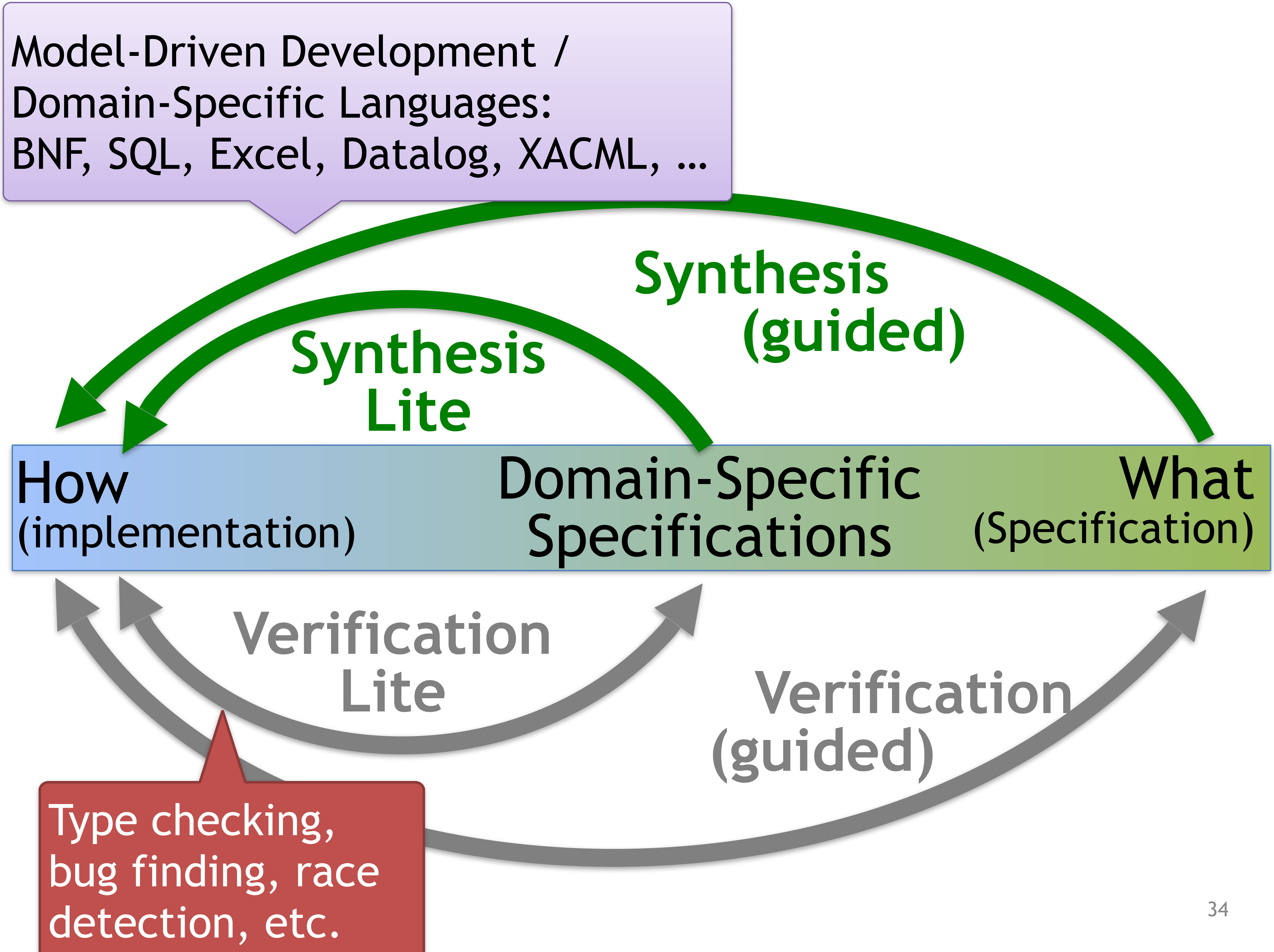
Spectrum of programming

How (implementation)	What (Specification)


How
(implementation)
What
(Specification)
Verification

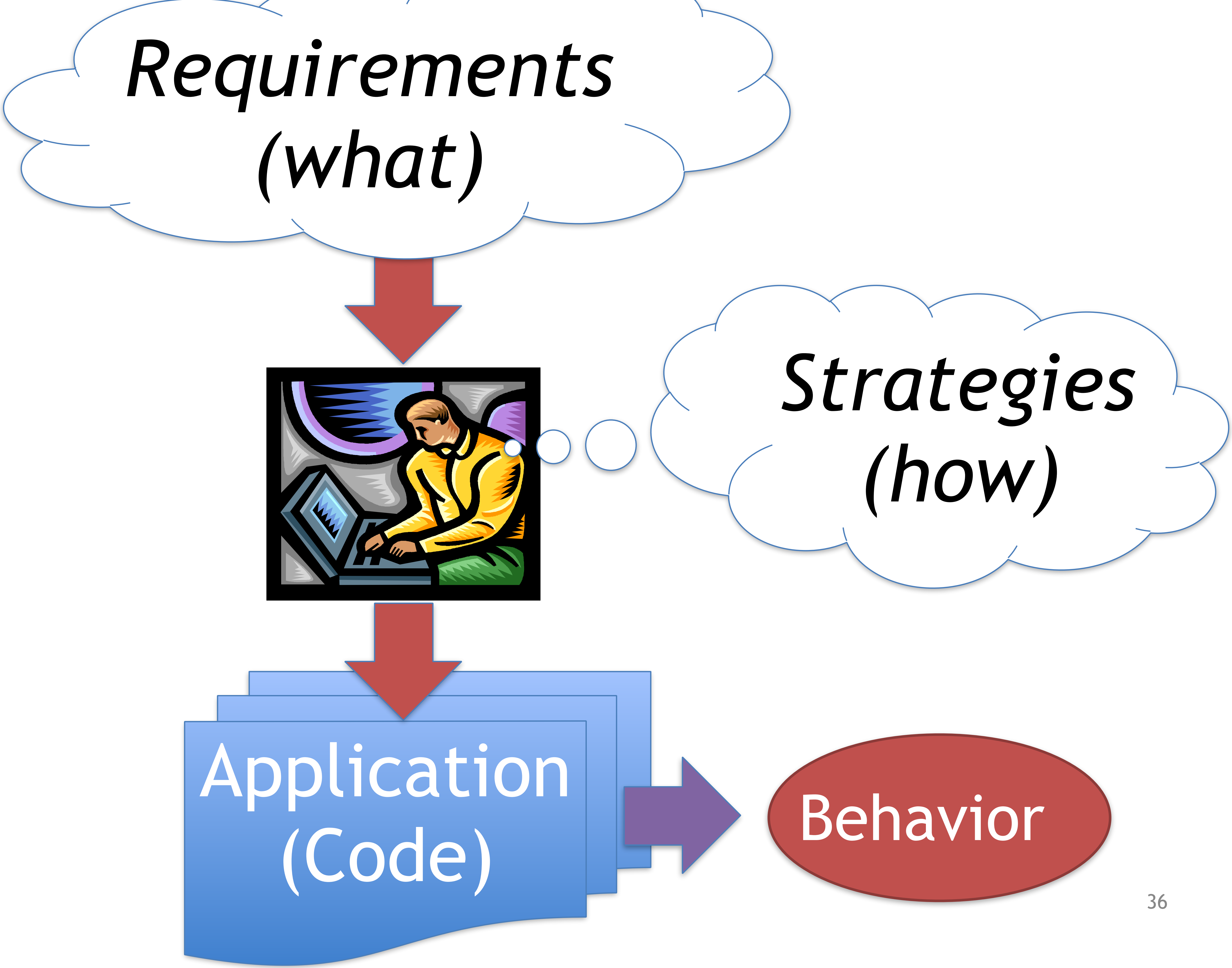

Synthesis
How
(implementation)
What
(Specification)
Verification

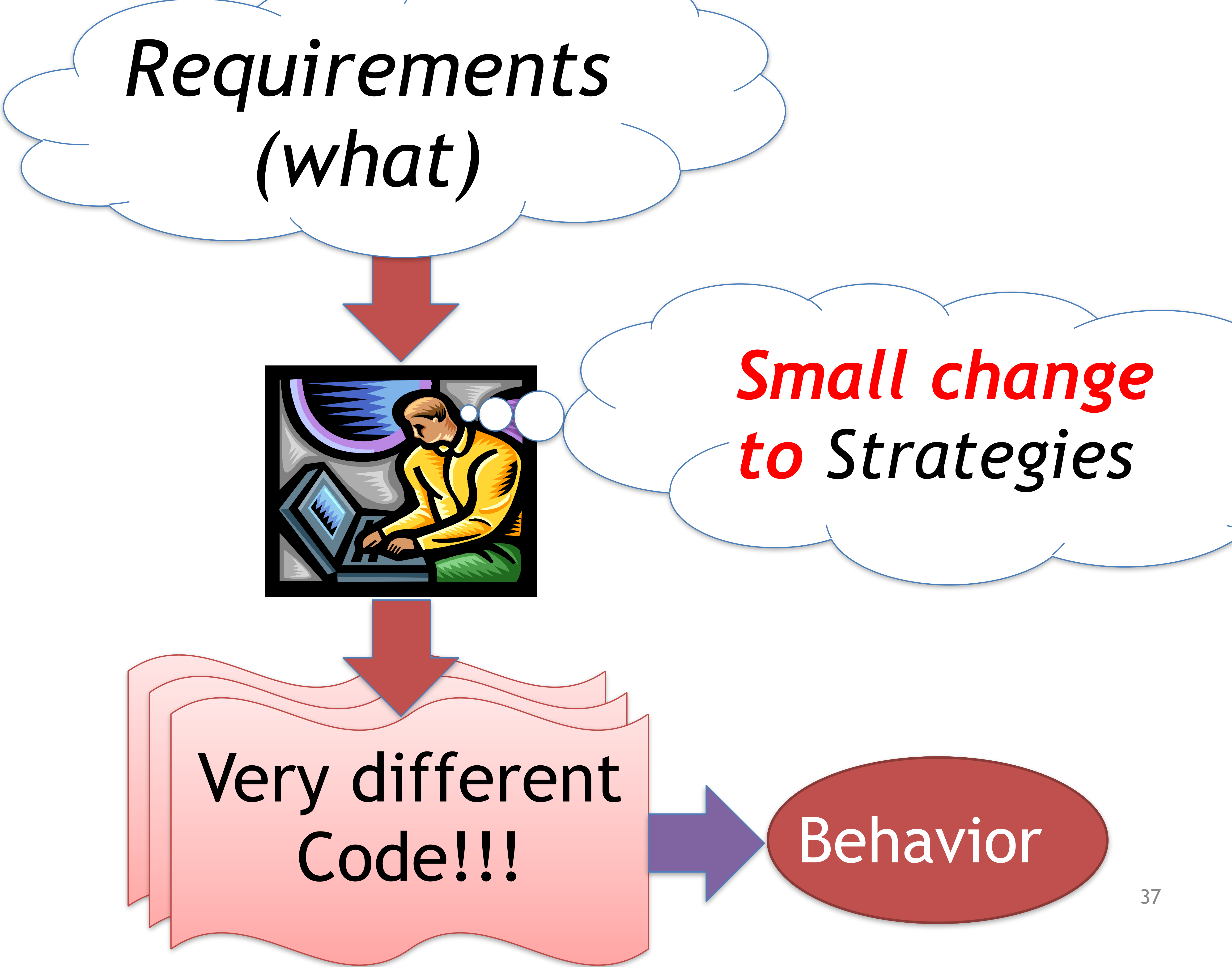

Synthesis
How
(implementation)
Restricted
Specifications
What
(Specification)
Verification
Lite
Verification
(guided)
Type checking,
bug finding, race
detection, etc.



Model-Driven Development / Domain-Specific Languages: BNF, SQL, Excel, Datalog, XACML, …
Synthesis (guided)
Synthesis Lite
How (implementation)
Domain-Specific Specifications
What (Specification)
Verification Lite
Verification (guided)
Type checking, bug finding, race detection, etc.

Ensō


Requirements
(what)
Strategies
(how)
Application
(Code)
Behavior


Requirements
(what)
Small change
to Strategies
Very different
Code!!!
Behavior

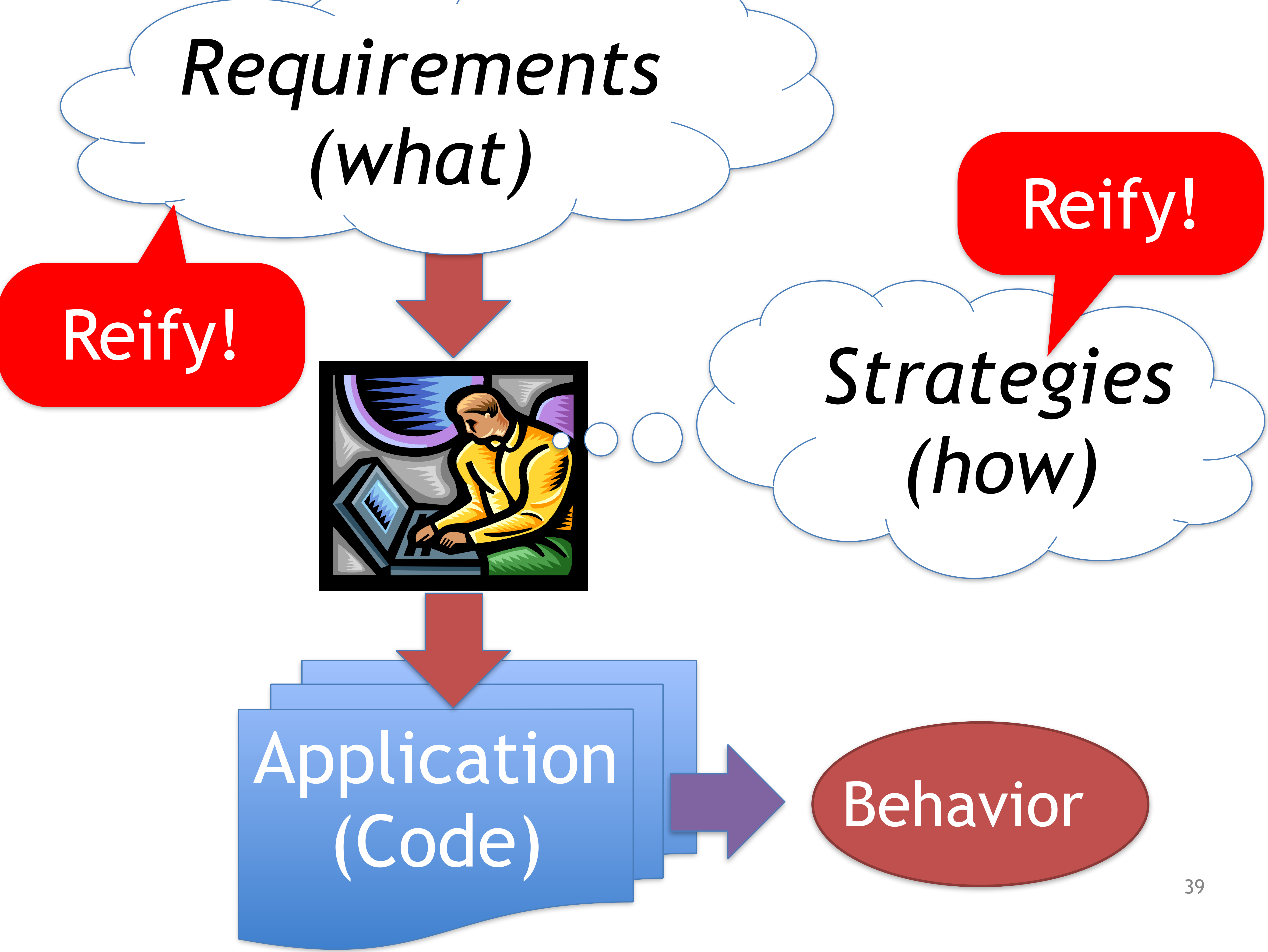
Requirements (what)

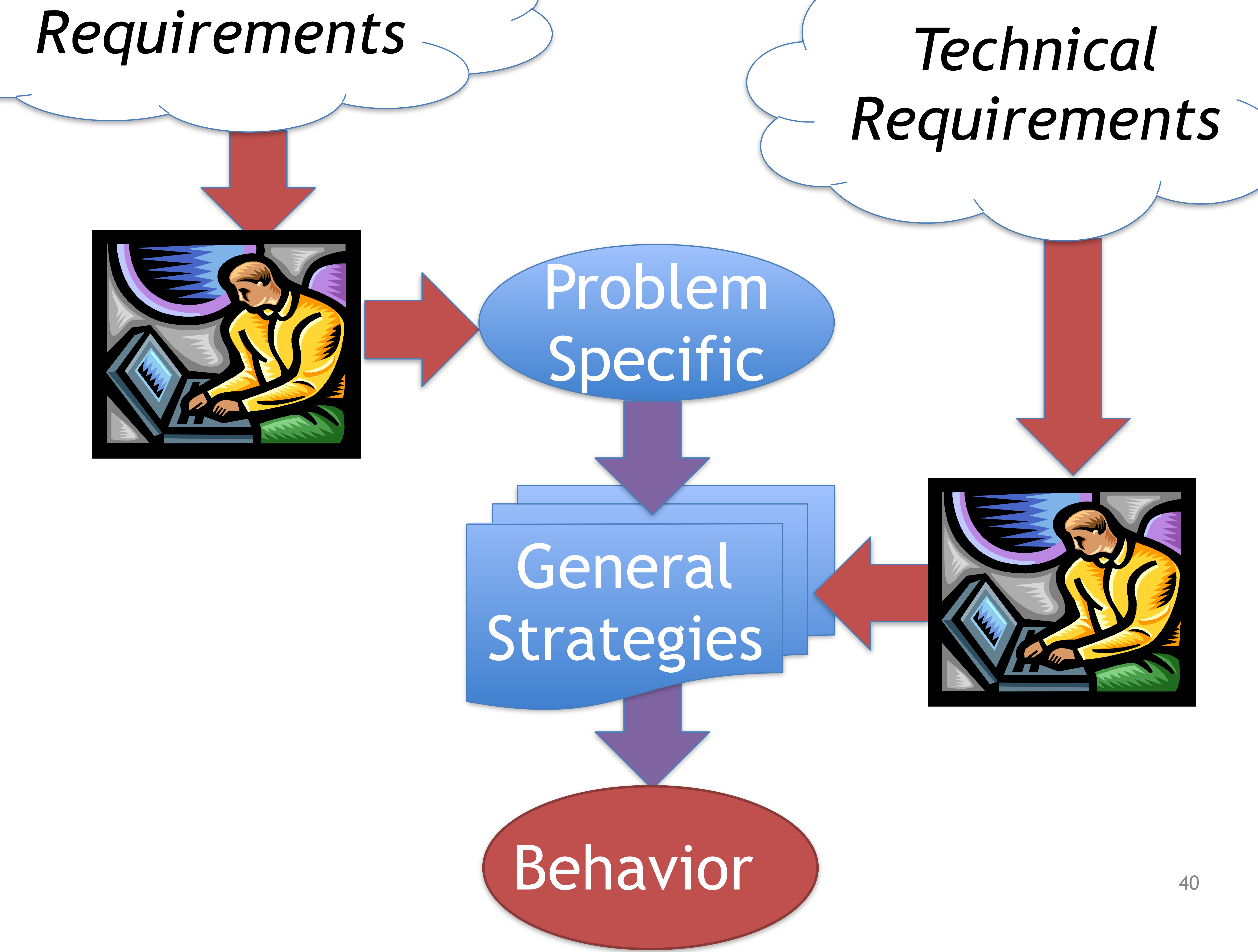
Small change to Strategies

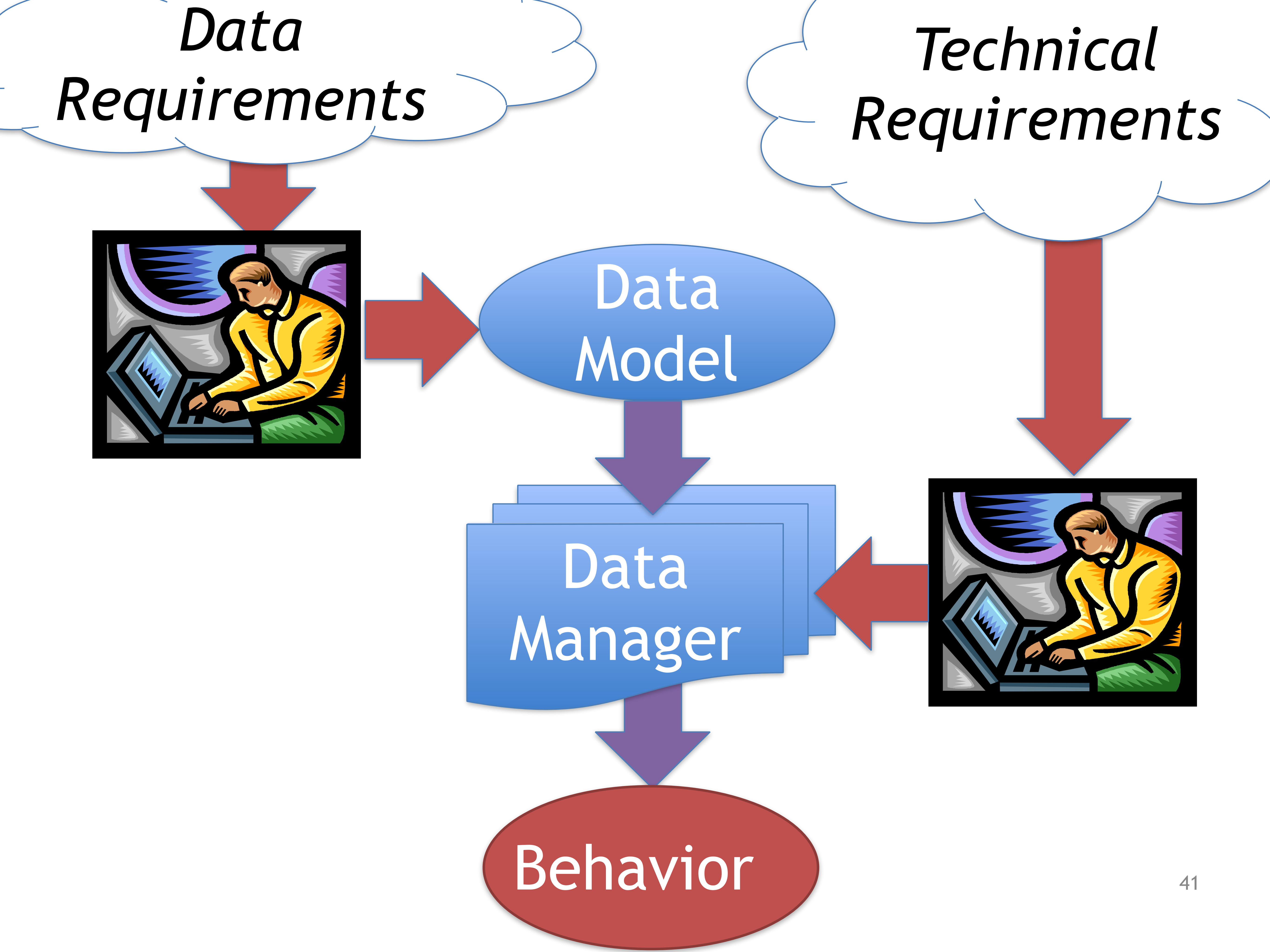
Chaos!

Very different Code

Behavior


Requirements
(what)
Reify!
Reify!
Strategies
(how)
Application
(Code)
Behavior


Requirements
Technical Requirements
Problem Specific
General Strategies
Behavior


Data Requirements
Technical Requirements
Data Model
Data Manager
Behavior

Using Managed Data (Ruby)

- Description of data to be managed

```
Point = { x: Integer, y: Integer }
```

- Dynamic creation based on metadata

```
p = BasicRecord.new Point
p.x = 3
p.y = -10
print p.x + p.y
p.z = 3  # error!
```

- *Factory* BasicRecord: Description<T> → T

Other Data Managers

- Mutability: control whether changes allowed
- Observable: posts notifications
- Constrained: checks multi-field invariants
- Derived: computed fields (reactive)
- Secure: checks authorization rules
- Graph: inverse fields (bidirectional)
- Persistence: store to database/external format
- General strategy for all accesses/updates
- Combine them for *modular strategies*

Grammars

- Mapping between *text* and *object graph*
- A *point* is written as (x, y)

Individual	*Grammar*
(3, 4)	*P ::= [Point] "(" x:int "," y:int ")"*

class

fields

- Notes:
  - Direct reading, no abstract syntax tree (AST)
  - Bidirectional: can parse and pretty-print
  - GLL parsing, *interpreted!*

State Machine Example

Door StateMachine

```
start Opened

state Opened
  on close go Closed

state Closed
  on open go Opened
  on lock go Locked

state Locked
  on unlock go Closed
```

StateMachine Grammar

```
M::= [Machine] "start" \start:</states[it]> states:S*
S ::= [State] "state" name:sym    out:T*
T ::= [Trans] "on" event:sym "go" to:</states[it]>
```

StateMachine Schema

```
class Machine
  start  : State
  states! State*

class State
  machine: Machine
  name     #   str
  out    !     Trans*
  in       :     Trans*

class Trans
  event      :  str
  from :  State / out
  to           :  State / in
```

A StateMachine Interpreter

```
def run_state_machine(m)
  current = m.start
  while gets
    puts "#{current.name}"
    input = $_.strip
    current.out.each do |trans|
      if trans.event == input
        current = trans.to
        break
      end
    end
  end
end
```

Sample Expression

3*(5+6)

Expression Grammar

```
E   ::= [Add] left:E "+" right:M  | M
M   ::= [Mul]  left:M "*" right:P  | P
P   ::= [Num] val:int             | "(" E ")"
```

An *Expression* Interpreter

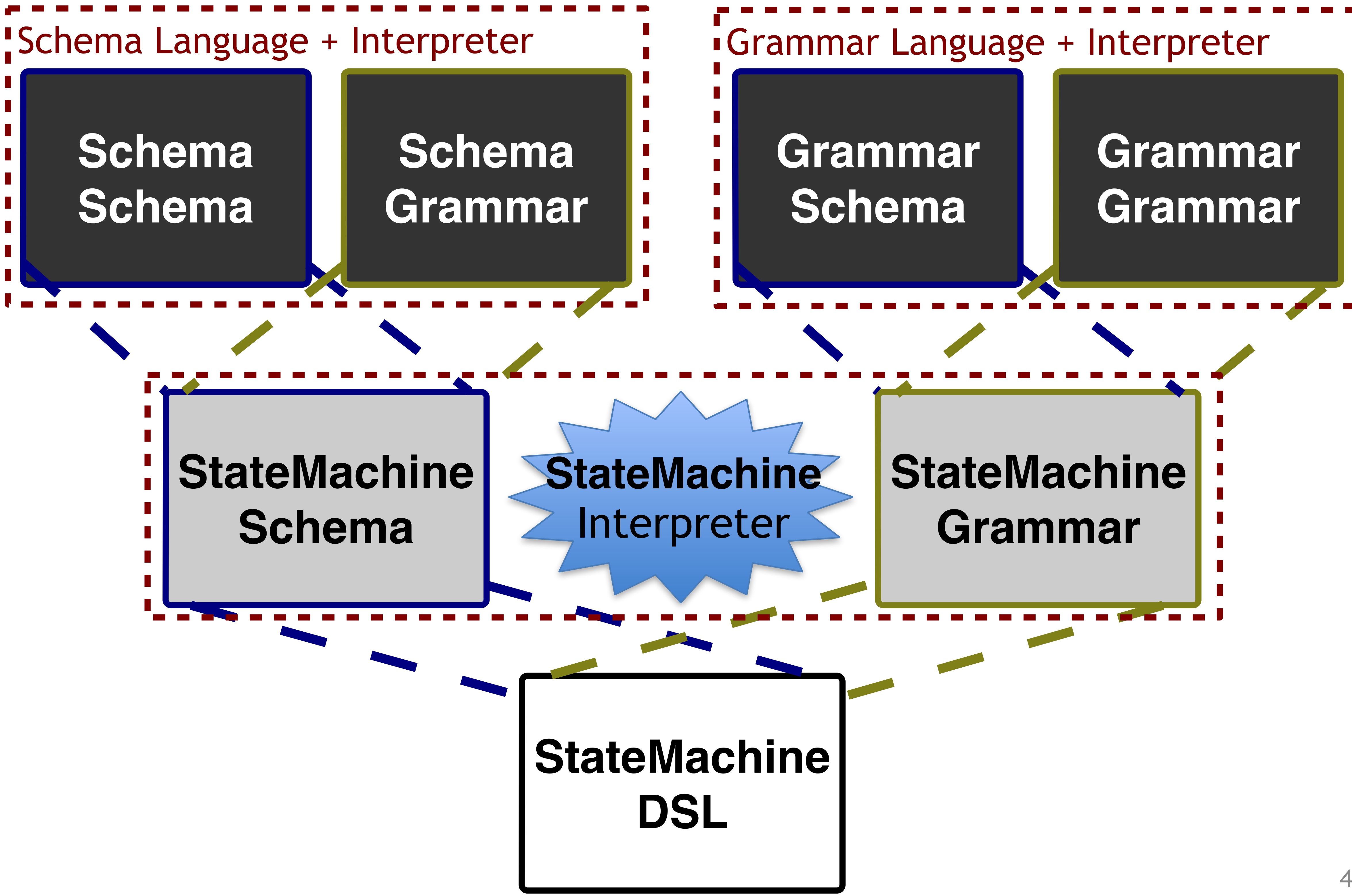
```
module Eval
  operation :eval

  def eval_Num(val)
    val
  end

  def eval_Add(left, right)
    left.eval + right.eval
  end

  def eval_Mul(left, right)
    left.eval * right.eval
  end
end
```

Expression Schema

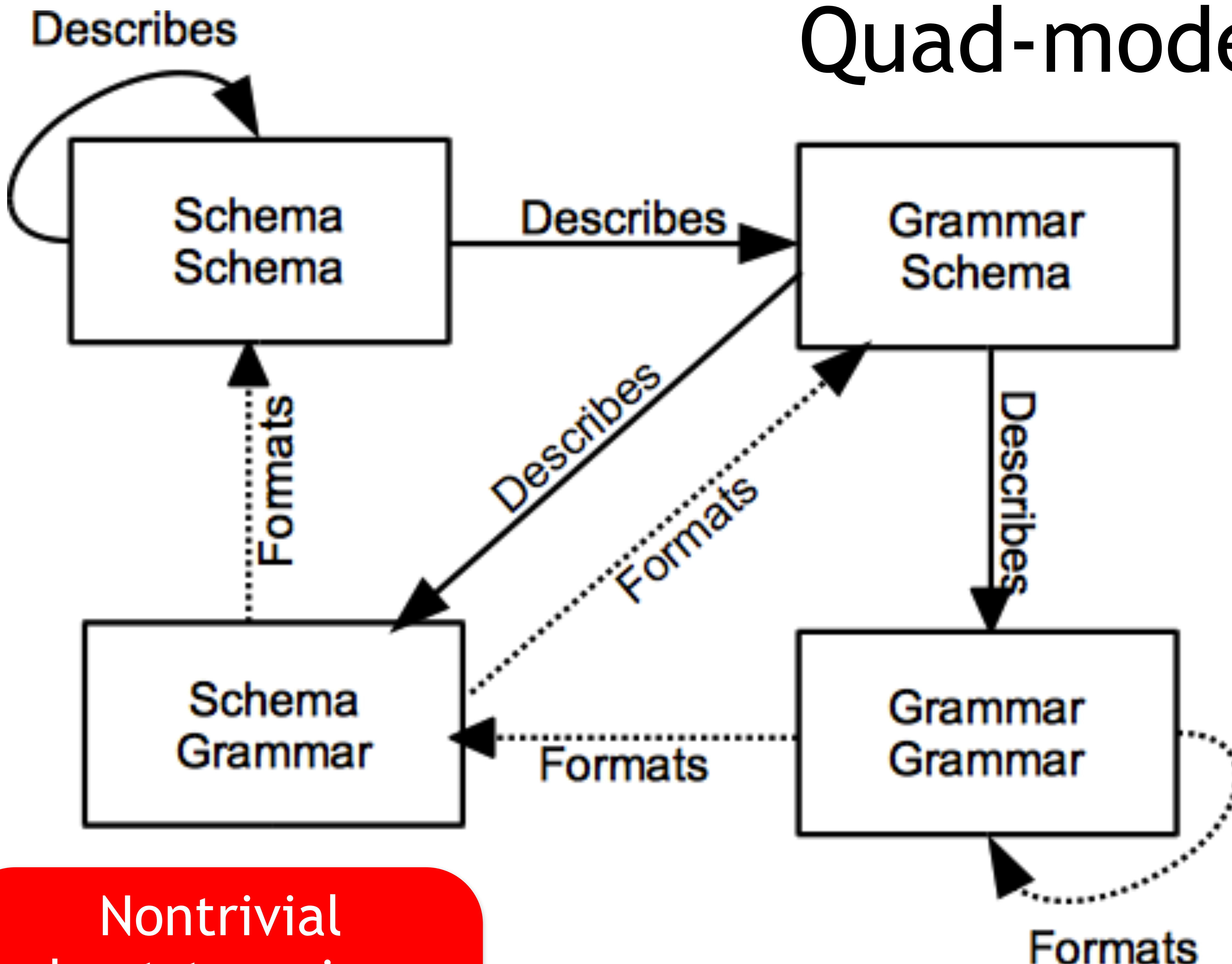
```
class Exp

class Num
  val : int

class Add
  left   : Exp
  right : Exp

class Mul
  left   : Exp
  right : Exp
```

Expression Example

Everything is a language



Quad-model



Nontrivial bootstrapping

Schema Schema

class Schema
types: Type*
class Type
name: string
class Primitive < Type
class Class < Type
fields: Field*
super: Type?

class Field
name: string
type: Type
many: bool
optional: bool

primitive string
primitive bool

(Self-Description)

Schema Grammar

start S

S ::= [Schema] types:T*

T ::= P | C

P ::= [Primitive] "primitive" name:sym

C ::= [Class] "class" name:sym ("<" S+)? fields:F*

S ::= <root.classes[it]>

F ::= [Field] name:sym ":" type:<types[it]> M? A?

M ::= "*" { many and optional }

  | "?" { optional }

  | "+" { many }

A ::= "/" inverse:<this.type.fields[it]>

  | "=" computed:Expr

Grammar Grammar

```
start G
G ::= [Grammar] "start"  start:</rules[it]> rules:R*
R ::= [Rule]      name:sym "::=" arg:A
A ::= [Alt]        alts:C+ @"|"
C ::= [Create]  "[" name:sym "]" arg:S  |  S
S ::= [Sequence] elements:F*
F ::= [Field]     name:sym ":" arg:P       |  P
P ::= [Lit]        value:str
      |  [Value] kind:("int" | "str" | "real" | "sym")
      |  [Ref]     "<" path:Path ">"
      |  [Call]    rule:</rules[it]>
      |  [Code]  "{" code:Expr "}"
      |  [Regular] arg:P "*" Sep?    { optional && many }
      |  [Regular] arg:P "?"          { optional }
      |  "(" A ")"
Sep ::= "@" sep:P
```

non-terminal name
→ reference to rule

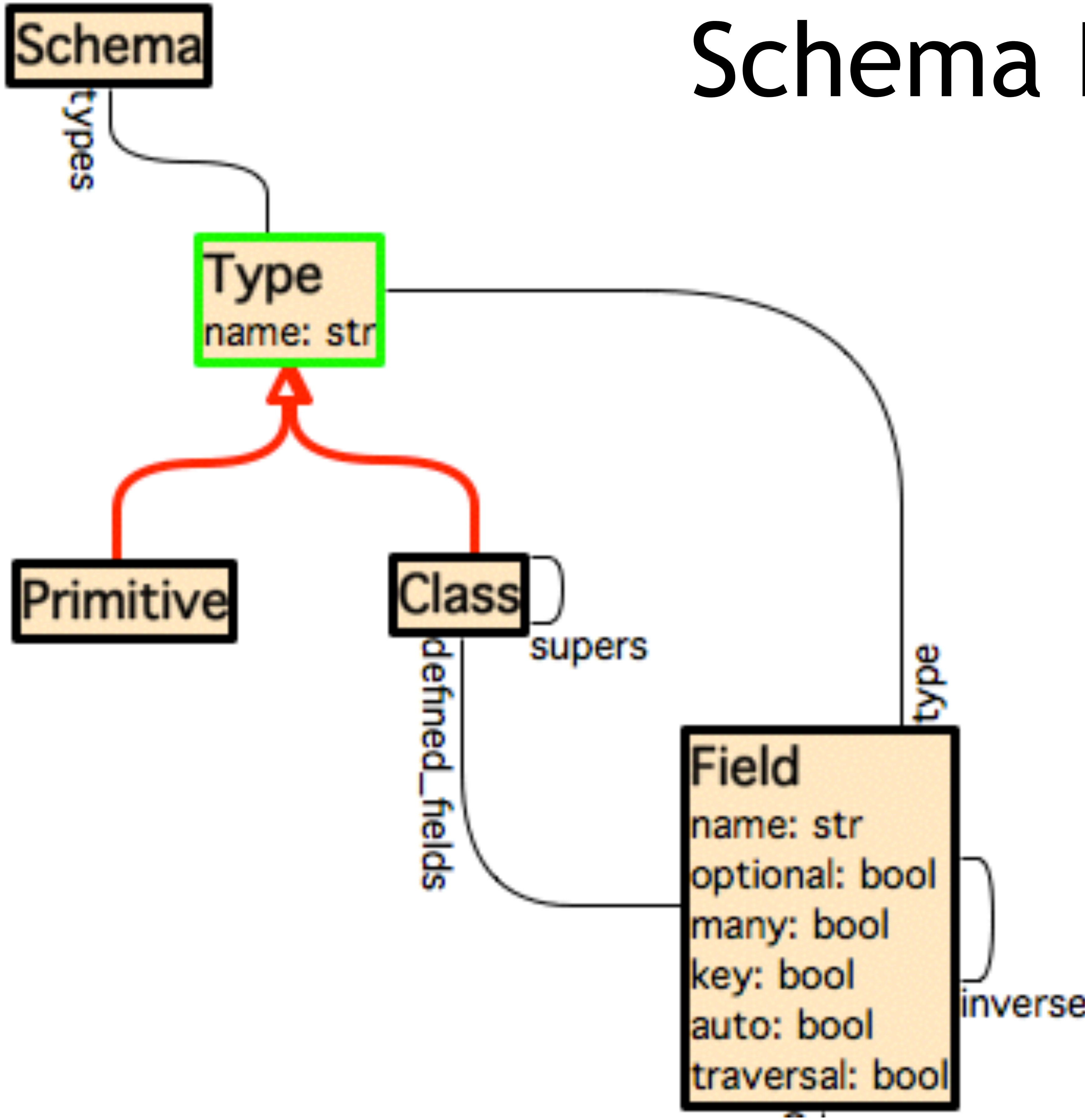
Grammar Schema

class Grammar start: Rule rules: Rule*
class Rule name: str arg: Pattern
class Pattern
class Terminal < Pattern
class Lit < Terminal value: str
class Value < Terminal kind: str
class Ref < Terminal path: Expr
class Alt < Pattern alts: Pattern+
class Sequence < Pattern elements: Pattern*
class Call < Pattern rule: Rule
class Create < Pattern name: str arg: Pattern
class Field < Pattern name: str arg: Pattern
class Regular < Pattern arg: Pattern sep: Pattern ?
optional: bool many: bool

Diagrams

- Model
  - Shapes and connectors
- Interpreter
  - Diagram render/edit application
  - Basic constraint solver

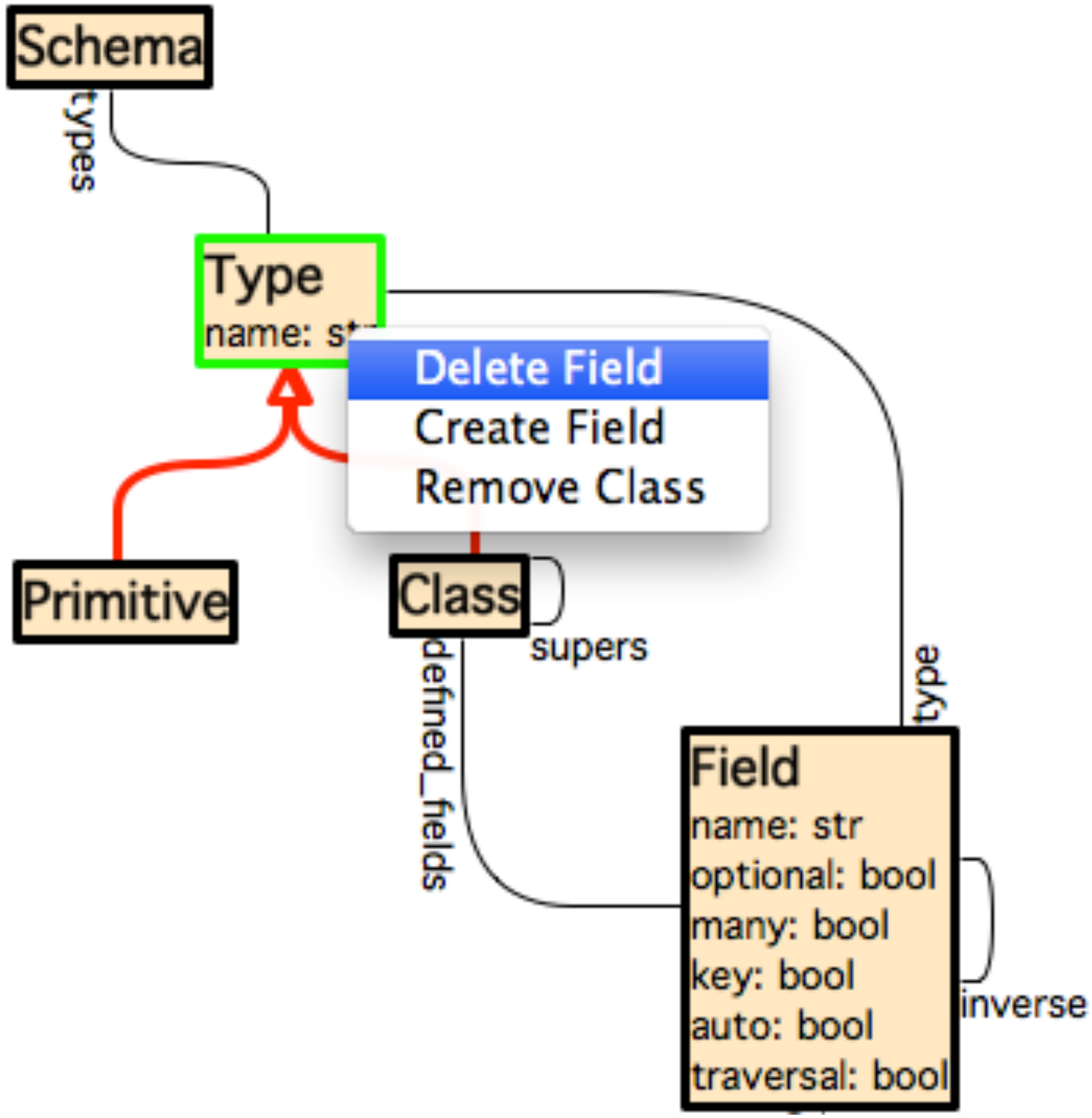
Schema Diagram



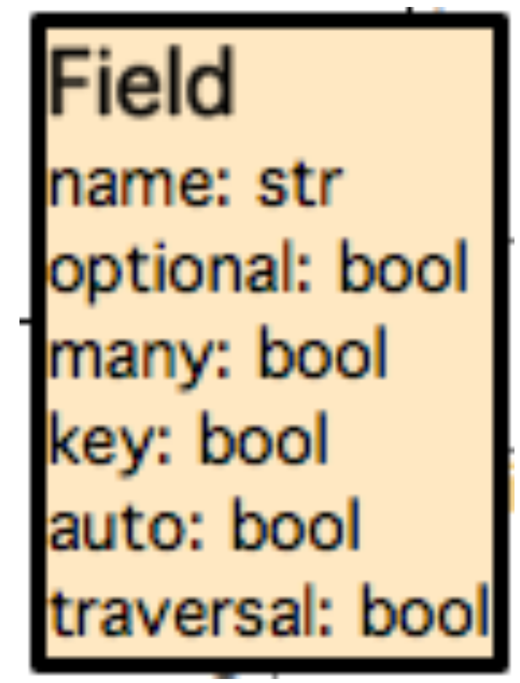
Stencils

- Model: mapping object graph → diagram
- Interpreter
  - Inherits functionality of Diagram editor
  - Maps object graph to diagram
    - Update projection if objects change
  - Maps diagram *changes* back to object graph
  - Binding for data and collections
    - Strategy uses schema information
    - Relationships get drop-downs, etc
    - Collections get add/remove menus

Schema Diagram Editor



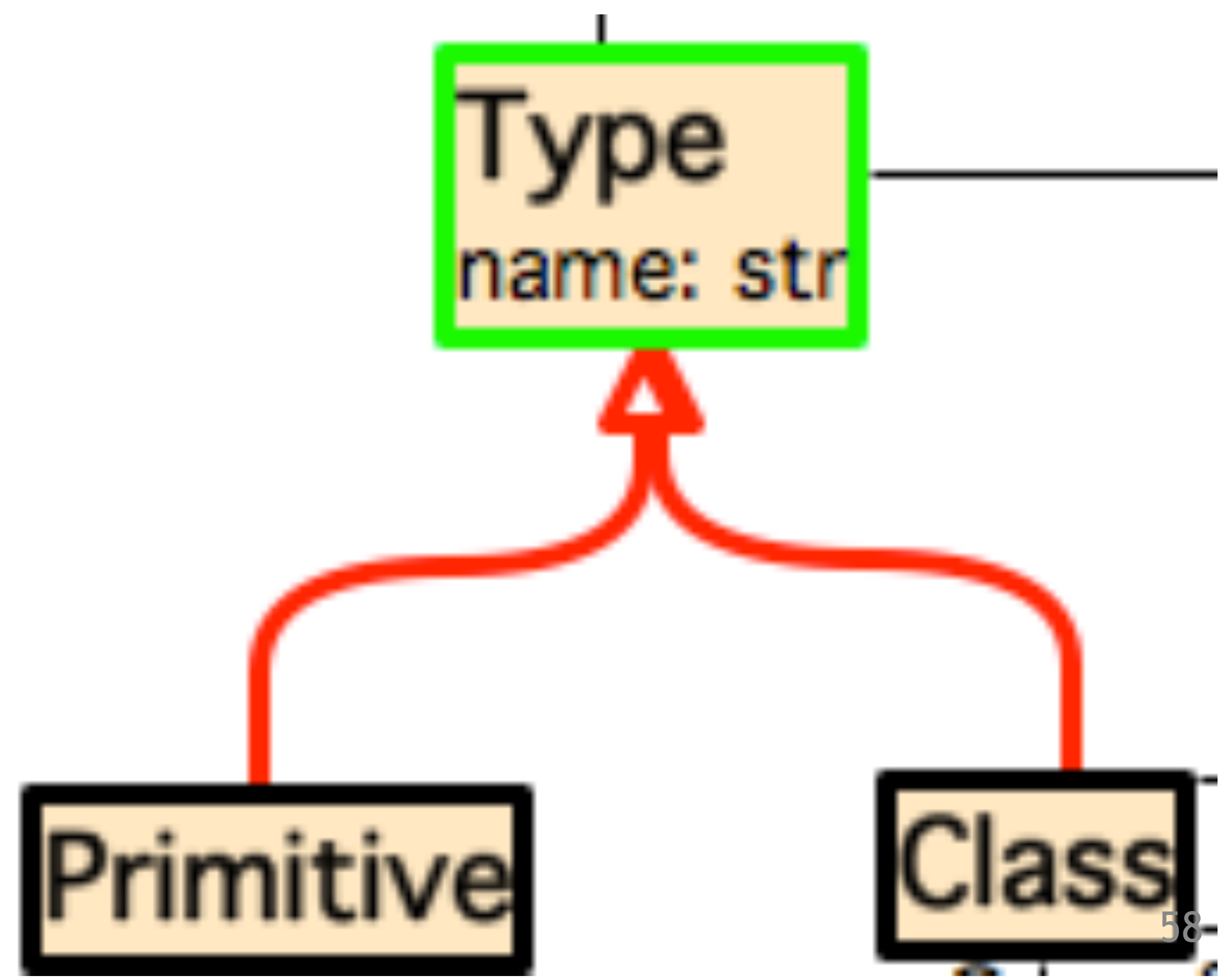
Schema Stencil




```
diagram(schema)
graph [font.size=12,fill.color=(255,255,255)] {
for "Class" class : schema.classes
   label class
     box [line.width=3, fill.color=(255,228,181)] {
       vertical {
          text [font.size=16,font.weight=700] class.name
          for "Field" field : class.defined_fields
            if (field.type is Primitive)
               horizontal {
                 text field.name // editable field name
                 text ": "
                 text field.type.name // drop-down for type
               }}}}
```

Schema Stencil: Connectors

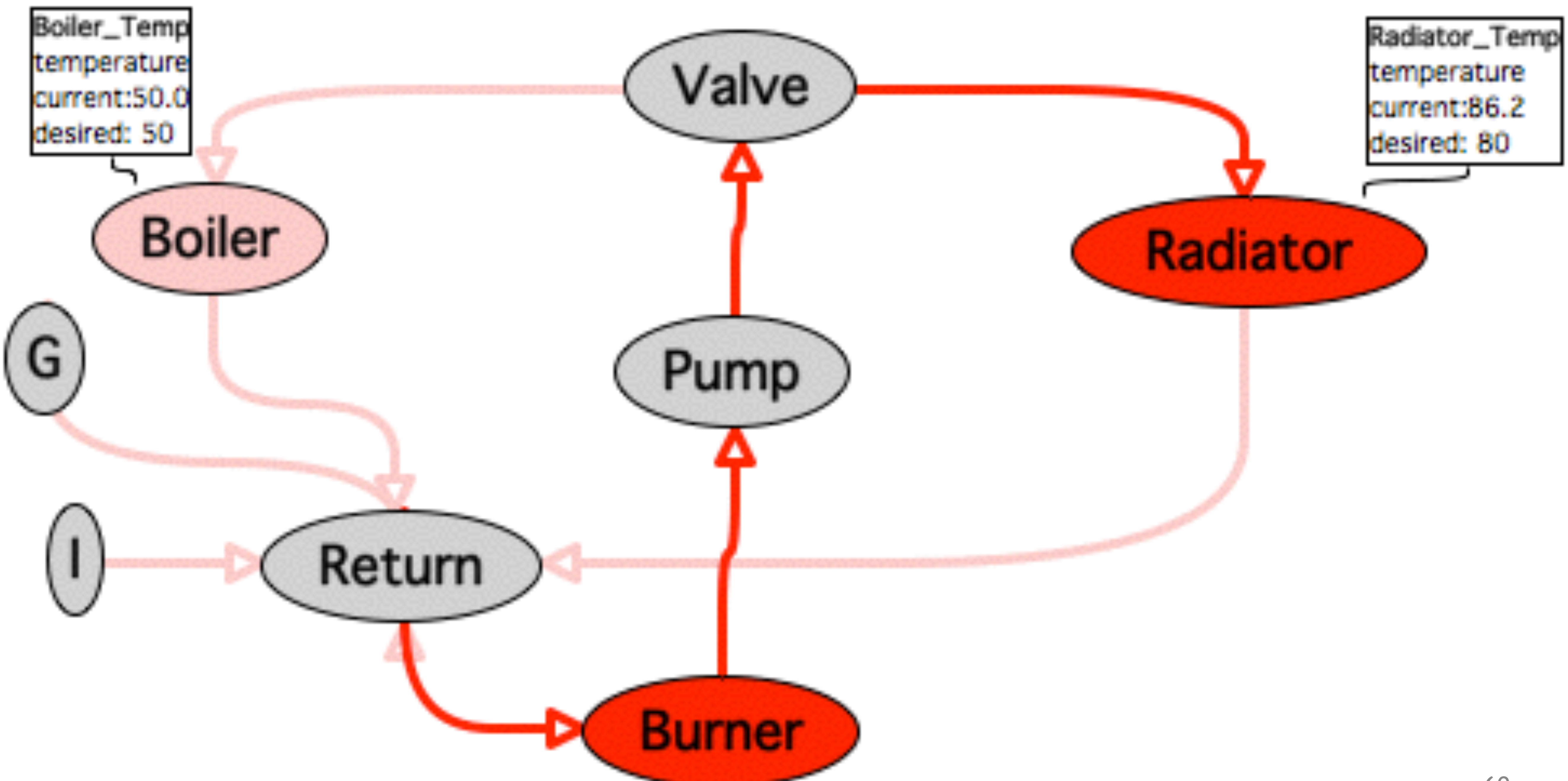
```
…
// create the subclass links
  for class : schema.classes
    for "Parent" super : class.supers
      connector [line.width=3, line.color=(255,0,0)]
          (class --> super)
…
[also for relationships]
```



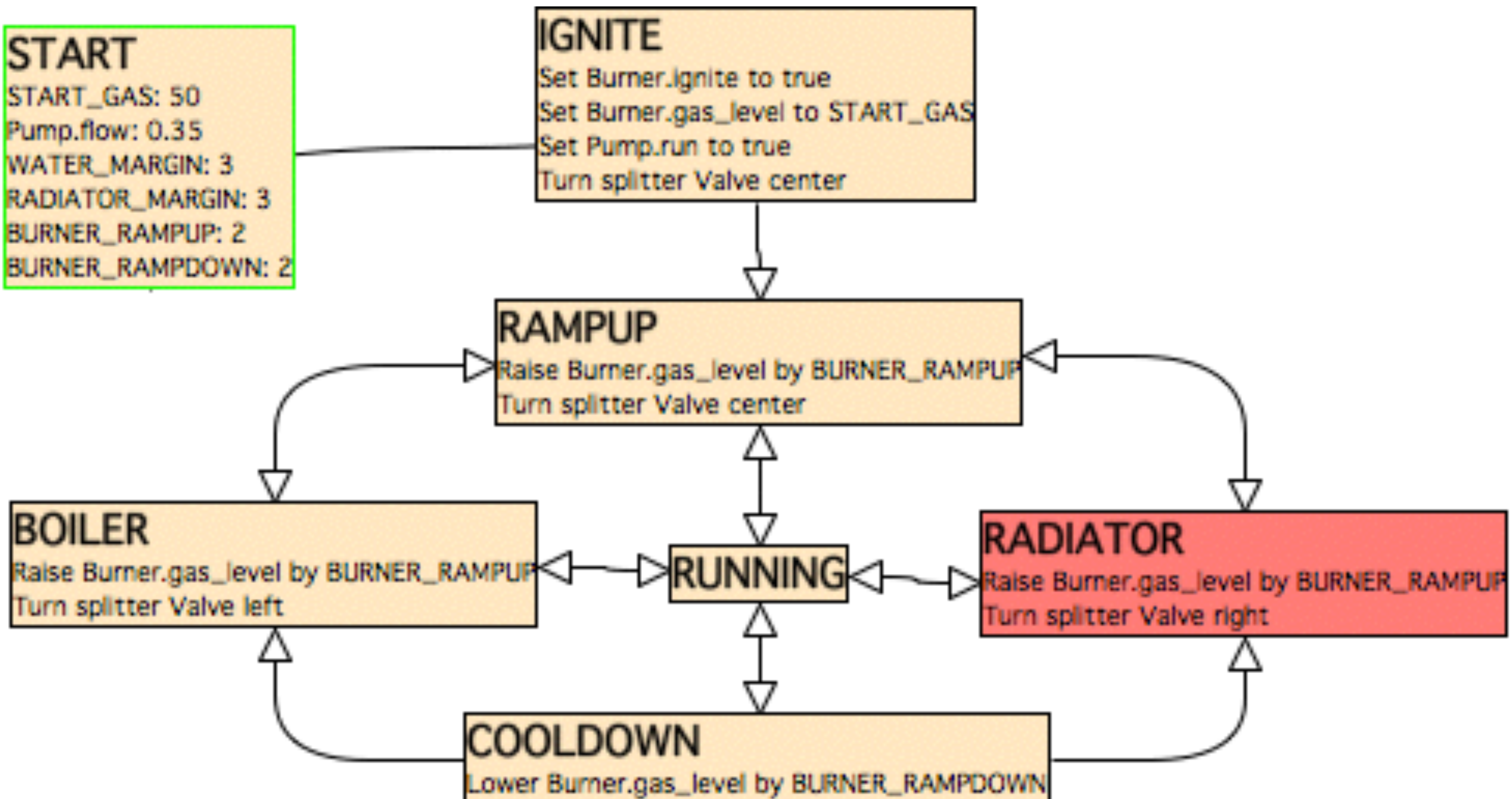
Language Workbench Challenge

- Models
  - Physical heating system
    – furnace, radiator, thermostat, etc
  - Controller for heating system
- Interpreter
  - Simulator for heating system
    – pressure, temperature
  - State machine interpreter
    – Events and actions

Physical Heating System Model



Piping Controller



Performance

- Ensō is currently slow but usable
  - Accessing a field involves two levels of meta-interpretation
  - My job is to give compiler people something to do
- Partial Evaluation of model interpreters

$\texttt{\textbf{web}}\texttt{(UI, Schema, db, request) : HTML}$

$\texttt{\textbf{web}}_{\texttt{[UI, Schema]}}\texttt{(db, request) : HTML}$

static *dynamic*

Aspect	Code SLOC	Model SLOC
Bootstrap	387	—
Utilities	256	—
Schemas	691	51
Grammar/Parse	885	106
Render	318	17
Web	932	305
Security	276	46
Diagram/Stencil	1389	176
Expressions	448	144
Core	5582	844
Piping	527	268

Ensō Summary

- Executable Specification Languages
  - Data, grammar, GUI, Web, Security, Queries, etc.
- External DSLs (not embedded)
- Interpreters (not compilers/model transform)
  - Multiple interpreters for each languages
- Composition of Languages/Interpreters
  - Reuse, extension, derivation (inheritance)
- Self-implemented (Ruby for base/interpreters)
  - Partial evaluation for speed

Don't Design
Your Programs

Program Your Designs

Ensō
enso-lang.org