Ensō

don’t design your programs, program your designs

Tijs van der Storm, May 17th, 2022

CWI
Centrum Wiskunde & Informatica

university of groningen

What is Ensō?

- Is it a language workbench?
- An architectural pattern?
- A model-driven framework?
- Object relational mapping?
- A programming paradigm?
- **No, it's Ensō**
- (all of the above, and then some)

Peak objects

> What does the future hold? In the late ’90s I started working on enterprise software and found that object-oriented programming in its pure form didn’t provide answers to the kinds of problems I was encountering.

William R. Cook, *Peak Objects,* ECOOP 2006 – Object-Oriented Programming pp 180-185, LNCS 4067

Peak objects

> Thus it should be possible to see traces of future paradigms in ideas that exist today. There are many promising ideas, including generative programming, reflection, partial evaluation, process algebra, constraint/logic programming, model-driven development, query optimization, XML, and web services. It is unlikely that focused research in any of these areas will lead to a breakthrough that triggers a paradigm shift. What is needed instead is a wholistic approach to the problem of building better software more easily, while harnessing specific technologies together to create a coherent paradigm.

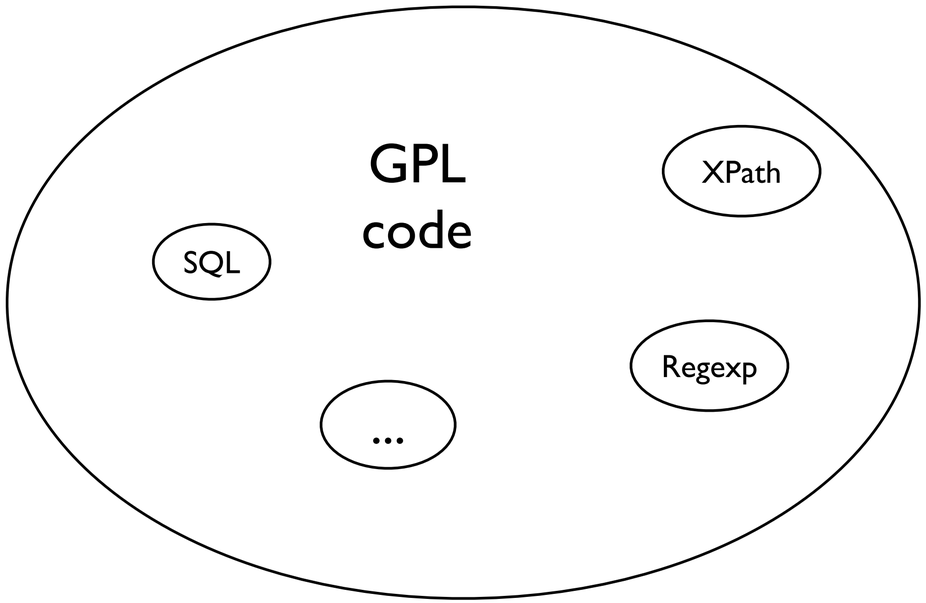
William R. Cook, *Peak Objects,* ECOOP 2006 – Object-Oriented Programming pp 180-185, LNCS 4067

Peak objects

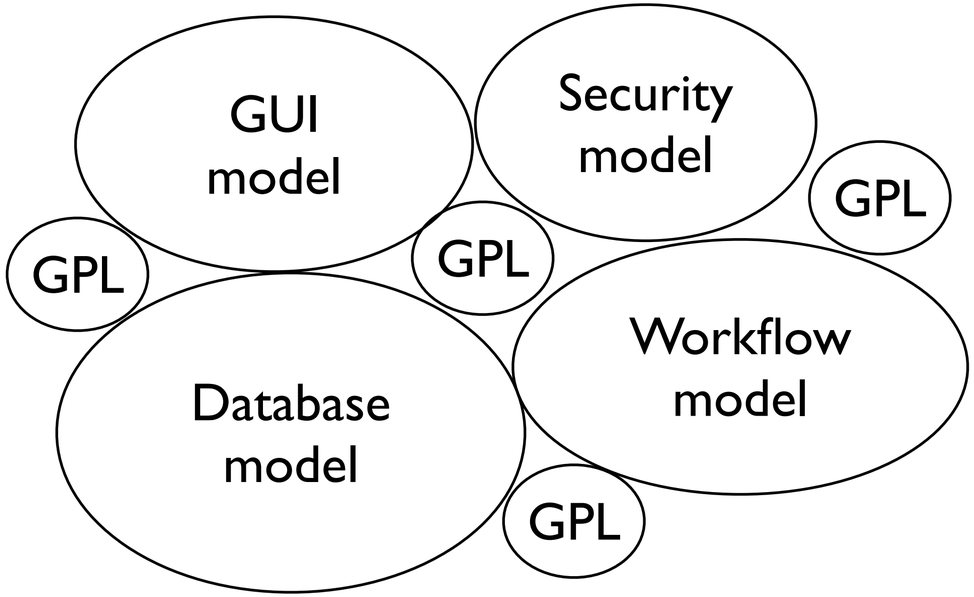
I want a more declarative description of systems. I find myself using domain-specific languages: for semantic data models, security rules, user interfaces, grammars, patterns, queries, consistency constraints, upgrade transformations, workflow processes. Little bits of procedural code may be embedded in the declarative framework, acting as procedural plugins.

William R. Cook, *Peak Objects,* ECOOP 2006 – Object-Oriented Programming pp 180-185, LNCS 4067

Traditional



Ensō



etc...

What vs how: the old debate

- What: Data, Security, GUI, Workflow etc.
- How: “Strategies” / “Designs”
- Normally: tangled and scattered
  - wanna change part of the stack? Change everything! (Including what)
- Don’t design your programs, program your designs!
- How is Ensō different:
    - strictly based on interpreters (no code generation)
    - object-oriented to the core: extensibility, wrapping, etc.

Ensō: a constellation of little languages and tools

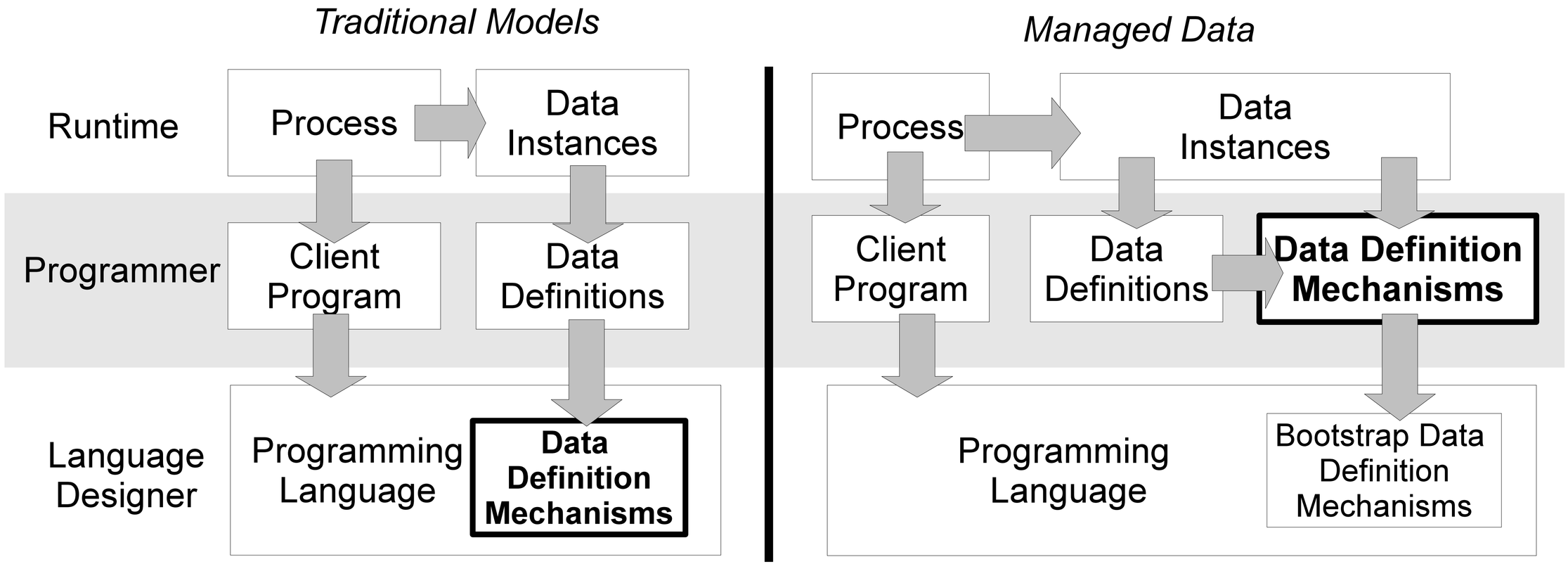
Languages/DSLs

- Schema: data modeling, ASG
- Grammar: parsing/formatting
- Diagram: graphical editors
- Security: access policies
- Web: web applications
- **All** have schema, grammar, and one or more interpreters

Generic Tools

- Print
- Traverse and fold
- Diff: versioning/data migration
- Merge: the basis for modularity
- Persist
- …
- *All guided by model's schema*

Managed data



Alex Loh, Tijs van der Storm, William R. Cook, **Managed data: modular strategies for data abstraction**, in: Onward! 2012: Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software, October 2012 Pages 179–194

Ensō core: schemas and grammars

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

```
class State
  machine: Machine / states
  name   # str
  out    ! Transition*
  in     : Transition*
```

```
class Transition
  event # str
  from  : State / out
  to    : State / in
```

Ensō core: schemas and grammars

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

```
class State
  machine: Machine / states
  name   # str
  out    ! Transition*
  in     : Transition*
```

```
class Transition
  event # str
  from  : State / out
  to    : State / in
```

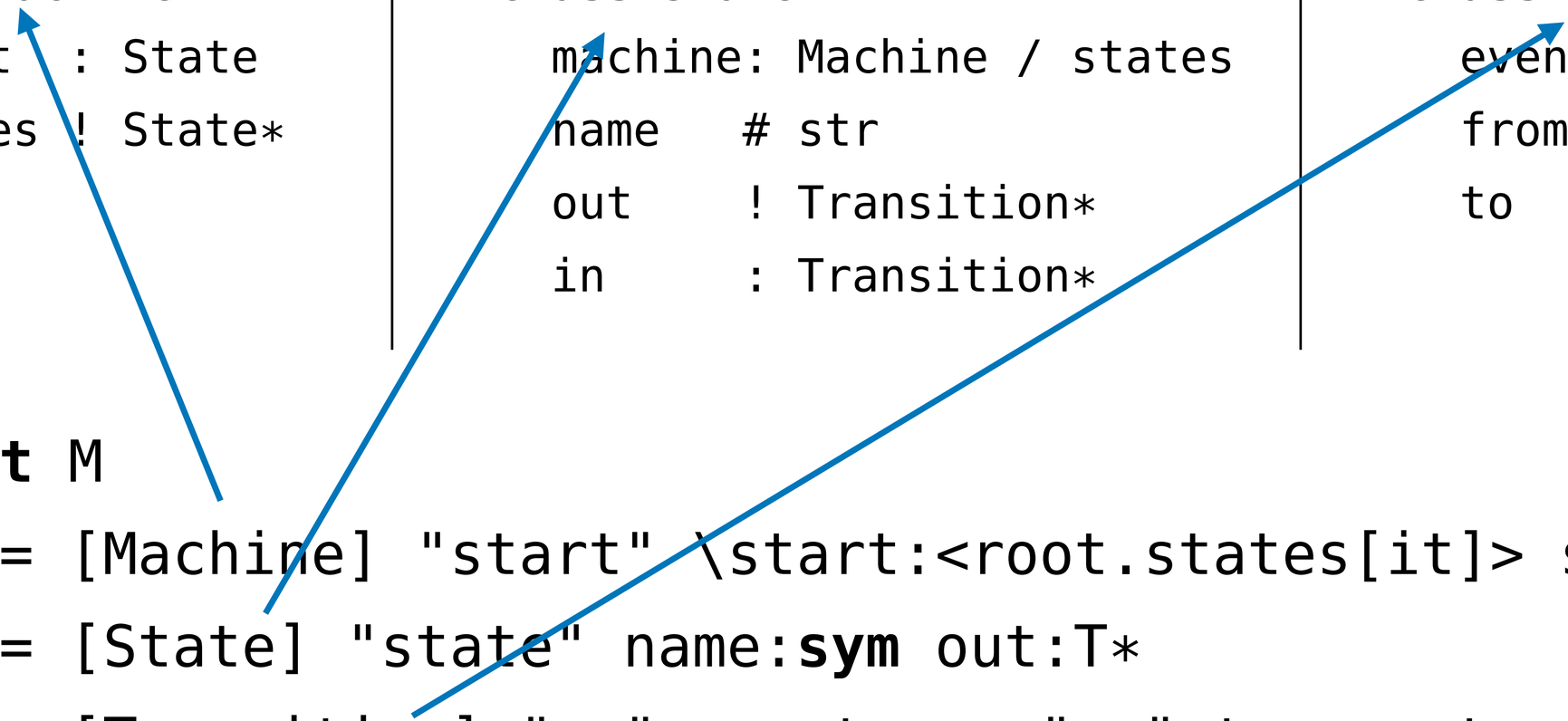
```
start M
M ::= [Machine] "start" \start:<root.states[it]> states:S*
S ::= [State] "state" name:sym out:T*
T ::= [Transition] "on" event:sym "go" to:<root.states[it]>
```

Ensō core: schemas and grammars

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

```
class State
  machine: Machine / states
  name   # str
  out    ! Transition*
  in     : Transition*
```

```
class Transition
  event # str
  from  : State / out
  to    : State / in
```



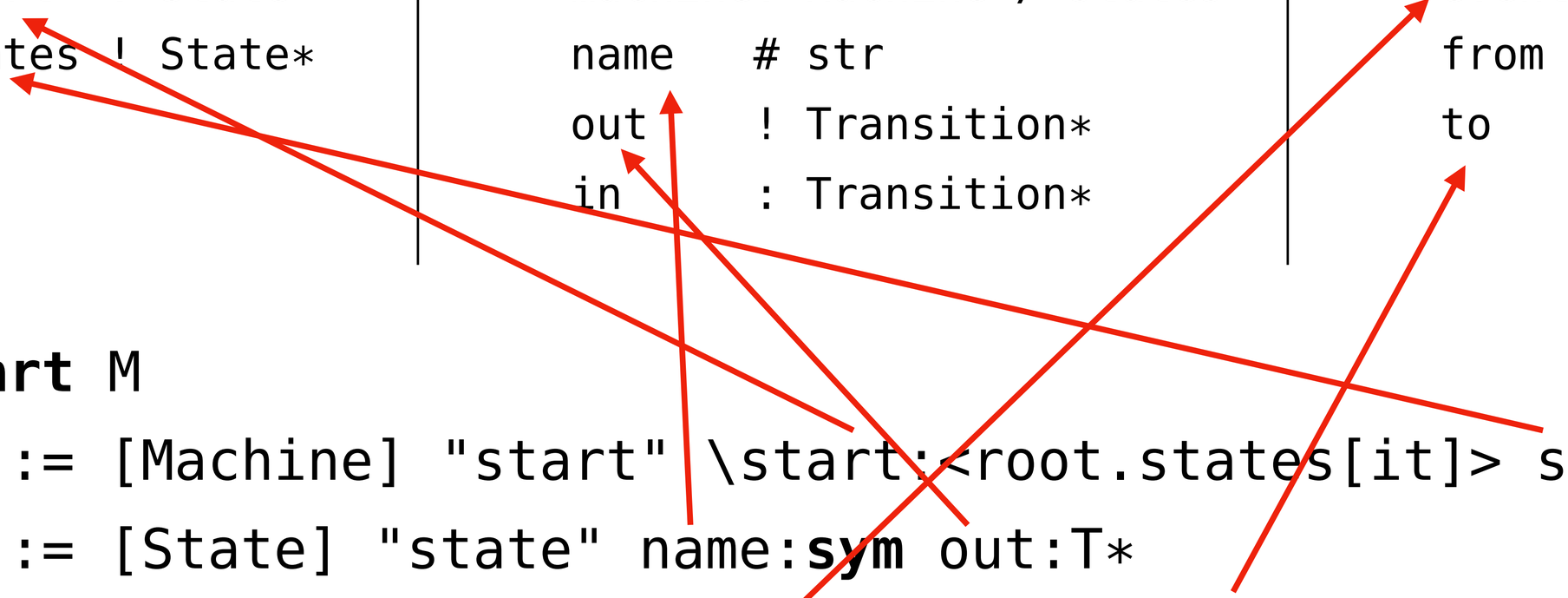
```
start M
M ::= [Machine] "start" \start:<root.states[it]> states:S*
S ::= [State] "state" name:sym out:T*
T ::= [Transition] "on" event:sym "go" to:<root.states[it]>
```

Ensō core: schemas and grammars

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

```
class State
  machine: Machine / states
  name   # str
  out    ! Transition*
  in     : Transition*
```

```
class Transition
  event # str
  from  : State / out
  to    : State / in
```



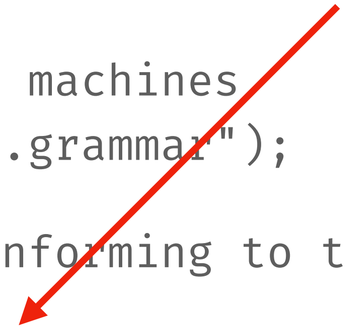
```
start M
M ::= [Machine] "start" \start:<root.states[it]> states:S*
S ::= [State] "state" name:sym out:T*
T ::= [Transition] "on" event:sym "go" to:<root.states[it]>
```

Client code
(pseudo code)

```
var enso = new Enso(); // bootstrapping

// load the schema describing state machines
var stmSchema = enso.loadSchema("statemachine.schema");

// load the grammar to parse (and format) state machines
var stmGrammar = enso.loadGrammar(“statemachine.grammar");

// create a factory to manage state machines conforming to the schema
var factory = DefaultManage.factory(stmSchema);

// load state machine using the grammar, schema as “type”, and factory as manager
var doors = enso.loadModel("door.statemachine", stmGrammar, stmSchema, factory);

// do stuff with it :)
println(doors.start.name)
```

Client code
(pseudo code)

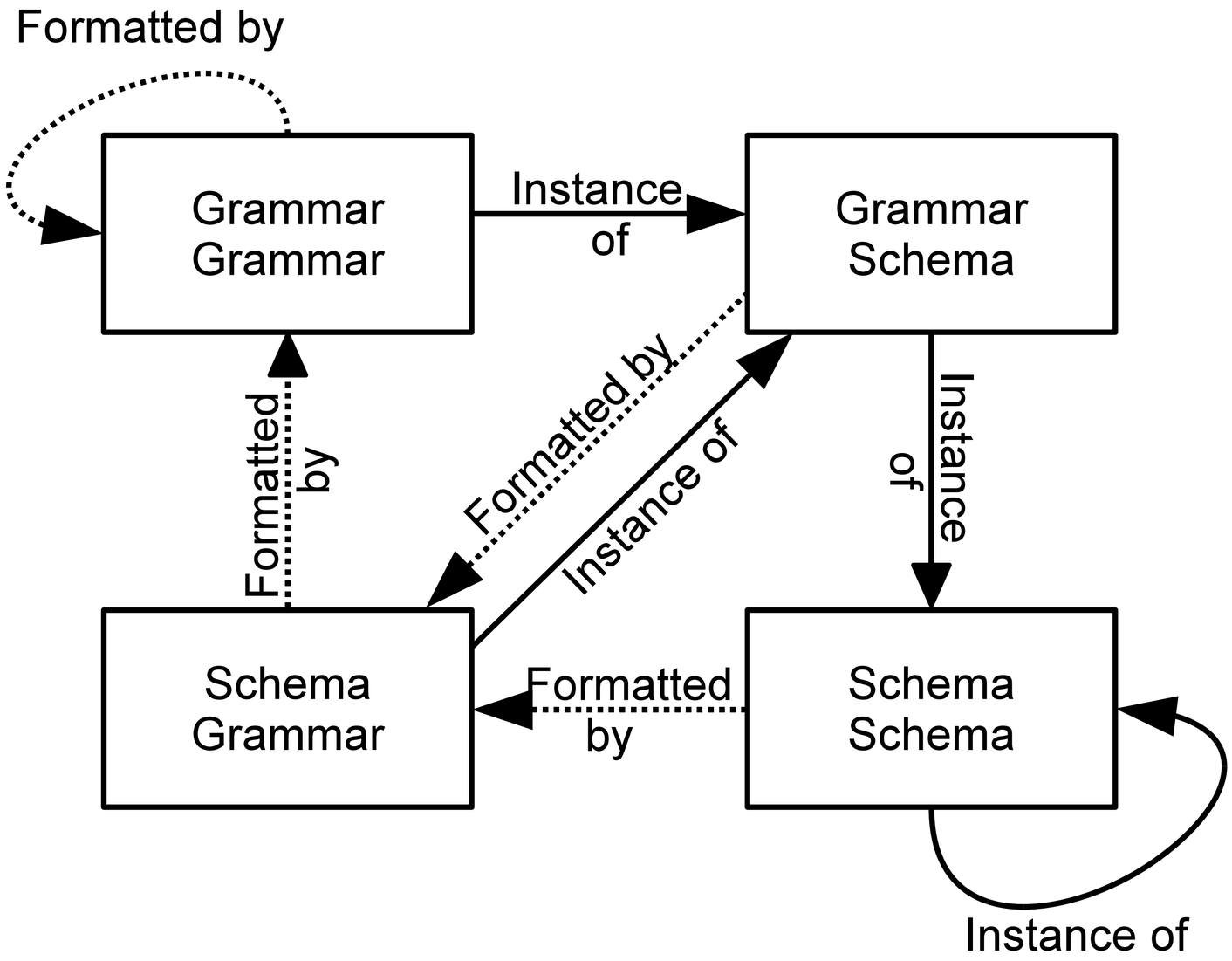
```
var enso = new Enso(); // bootstrapping

// load the schema describing state machines
var stmSchema = enso.loadSchema("statemachine.schema");

// load the grammar to parse (and format) state machines
var stmGrammar = enso.loadGrammar(“statemachine.grammar");

// create a factory to manage state machines conforming to the schema
var factory = DefaultManage.factory(stmSchema);

// load state machine using the grammar, schema as “type”, and factory as manager
var doors = enso.loadModel("door.statemachine", stmGrammar, stmSchema, factory);

// do stuff with it :)
println(doors.start.name)
```



This is the key point!
Alternative factories could:

- implement access control
- do advanced type checking
- automatically persist objects
- log access to objects
- disallow modifications
- … etc.

Meta-circular architecture



Demo

Ensō

Some of William's slogans…

- Don't design your programs, program your designs
- DSLs: synthesis lite + verification lite
- Enable good, instead of preventing bad
- AOP is a very bad solution for a great problem
- The Smalltalk of modeling
- Objects inside
- Overriding the "."