Building a Large-Scale Generic Object Model:

Applying the CYC Upper Ontology to Object Database Development in Java

Stephen Strom
strom@acm.org
Senior Architect
TechTrader

www.techtrader.com

Summary

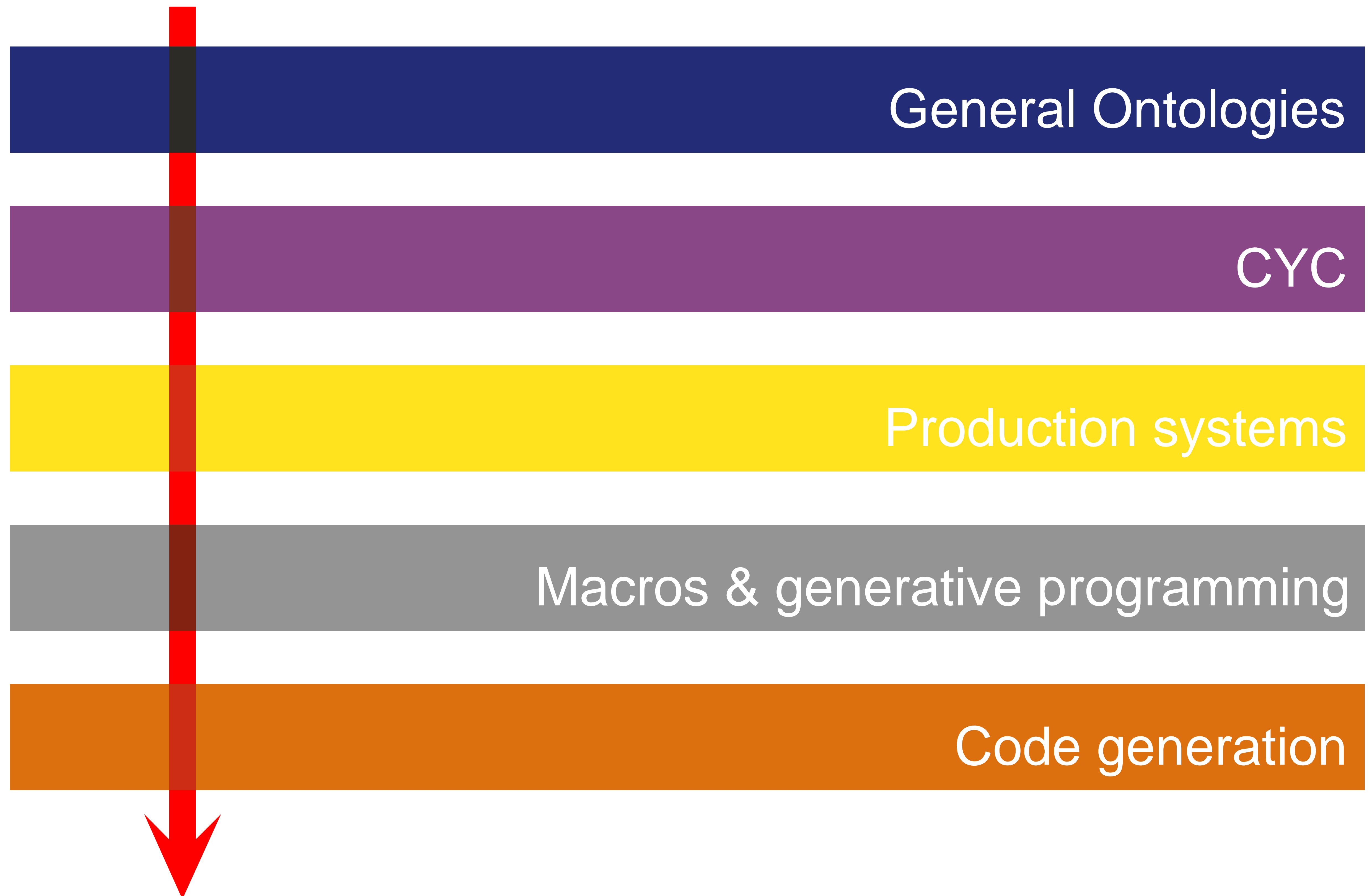
The CYC upper ontology is a large, publicly available "theory of everything."

A production (rule-based) system can be used to effectively translate the CYC upper ontology into a Java object model.

The task is easier if an intermediate target language is used.

Java lacks several features which would make this task easier and which would improve its suitability as a knowledge-representation language.

The approach in this talk: Provide a narrow drill-down through several (very) broad topics.



What is an ontology?

In philosophy, ontology is

"The study of being in so far as this is shared in common by all entities, both material and immaterial. It deals with the most general properties of beings in all their different varities."

Kim & Sosa, *A Companion to Metaphysics.*

The term has come to mean a model of external reality.

Nearly synonymous with

* Analysis model
* Domain model
* Data model

Contrast with "epistemology"...

What is a general ontology?

* It must be "applicable in more-or-less any special-purpose domain (with the addition of domain specific axioms)’’

* "Different areas of knowledge must be **unified**.’’

from Russel and Norvig, *Artificial Intelligence: A Modern Approach*

Within the programming and OO worlds, attempts to develop general ontologies include

- Analysis Patterns (Fowler)
- Data Model Patterns (Hay)
- Universal Data Models (Silverston, Inmon, Graziano)

Why use general ontologies?

Object-oriented analysis has lagged behind design and implementation:
* Tool support is still very weak.
* Little reuse ÑEach project concentrates on building an analysis model that addresses only its current requirements, usually codified into a more-or-less formal "domain model’’: a set of classes that is incorporated into the final product, includ-ing the system database.

Consequences:
* Analysis models for the system do not evolve gracefully. They are fragile and must usually be re-developed for each new project.
* Database schemas cannot be gracefully extended, even in object databases which are generally more friendly to schema evolution.
* Additionally, it is difficult for disparate systems to communi-cate.

What is CYC?

* Short for enCYClopedia.

* Conceived in the early 1980s, the brainchild of AI researcher Douglas Lenat.

* Based on the idea that AI is only possible when computers know a lot.

* Attempts to encode all the knowledge contained in a desk-top encyclopedia into a form readily available/understandable to a machine (a first-order logic language called CycL).

* Originally developed as part of MCC in Austin, now a separate company.

Part of what has come out of CYC is a general ontology, a fraction of which has been released for public use as the "CYC Upper Ontology" (approximately 3000 facts and rules).

Why use CYC?

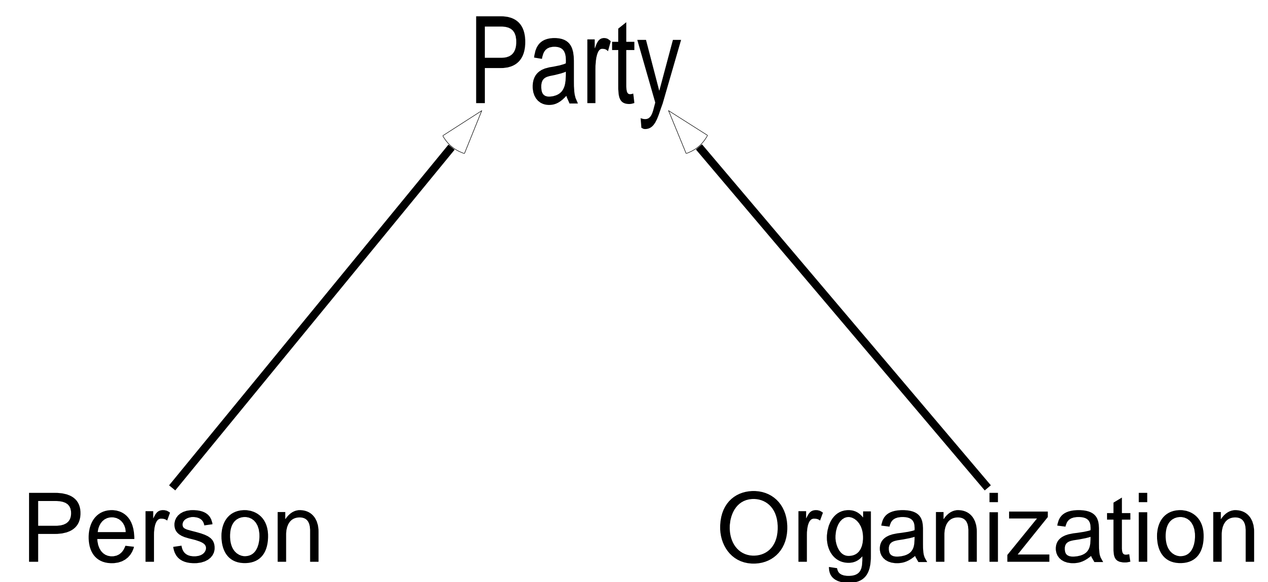
* CYC is the product of many person-years of development.

* The consistency of its facts has been automatically checked.

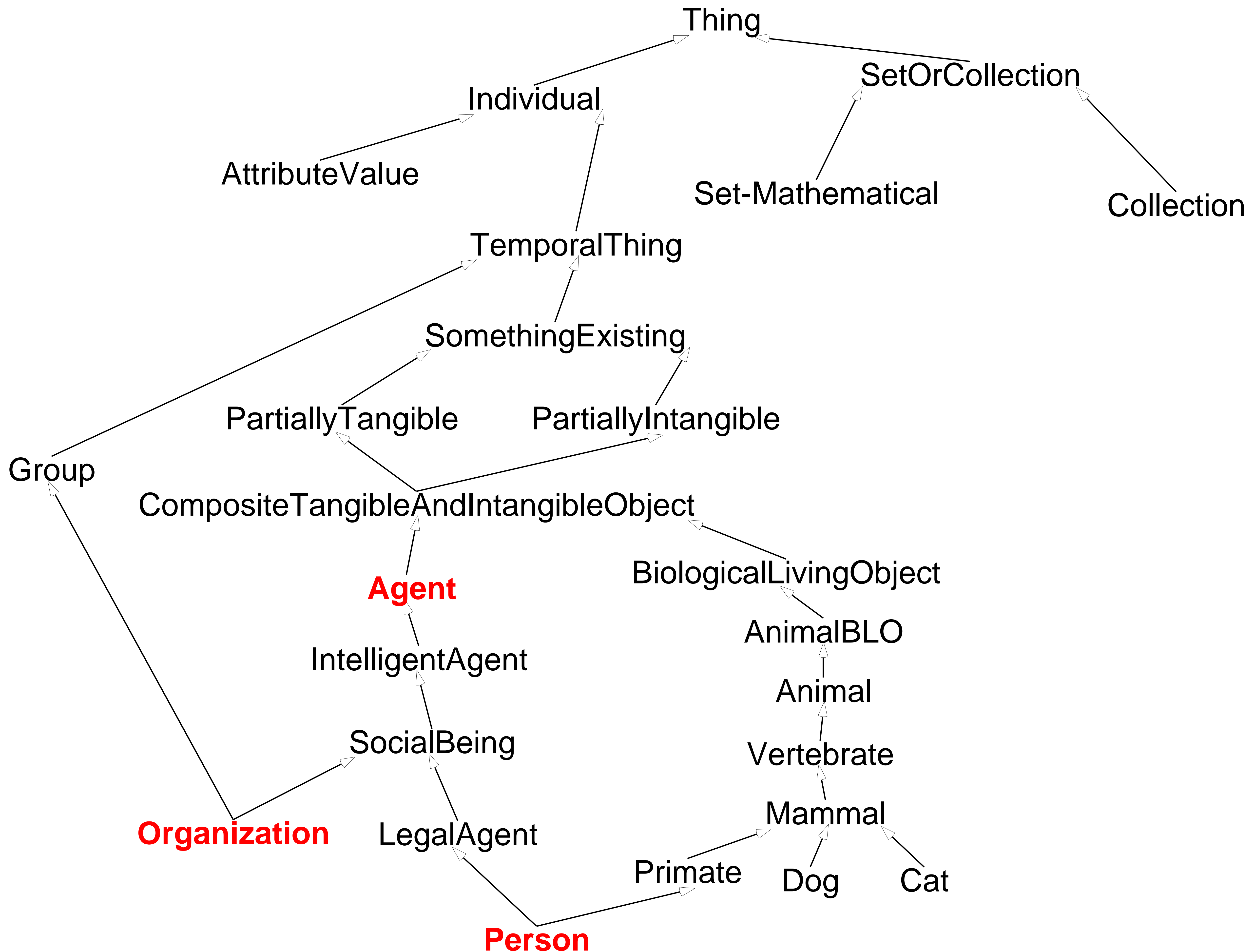
* In sheer size, it dwarfs systems such as Fowler's and covers a much broader range of concepts.

* It is available in a form which easily makes for an executable model, and which can be translated into many alternate forms.

The taxonomy for the Party analysis pattern from Fowler is broadly applicable...



But CYC places the same concepts within a broad network of taxonomic relationships.



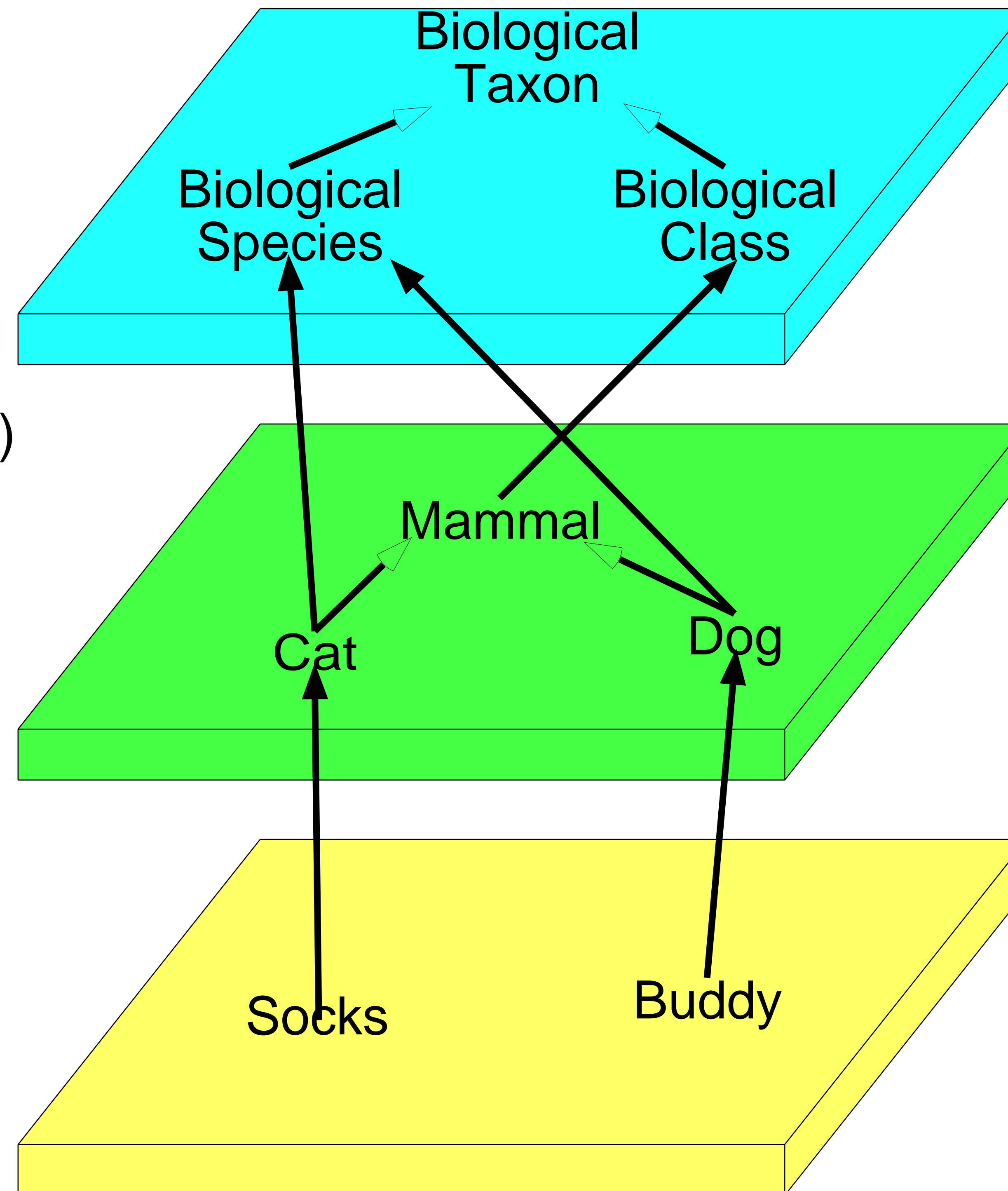
CYC supports multiple layers of classes (Fowler’s "knowledge level," sometimes misleadingly referred to as "meta-model").

(genls BiologicalSpecies
BiologicalTaxon)
(genls BiologicalClass
BiologicalTaxon)

(is-a Cat BiologicalSpecies)
(is-a Dog BiologicalSpecies)
(is-a Mammal BiologicalClass)

(genls Cat Mammal)
(genls Dog Mammal)

(is-a Cat Socks)
(is-a Dog Buddy)



2nd-order Classes (3rd-order objects)

Classes (2nd-order objects)

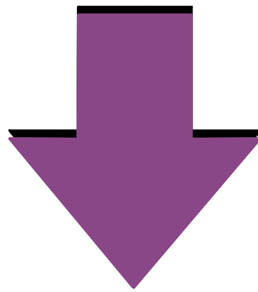
Objects (1st order)

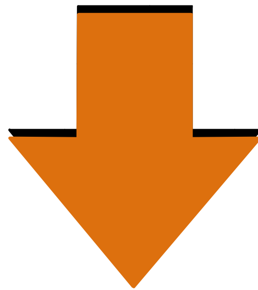
is-a

genls

Can the CYC upper ontology be translated into Java???

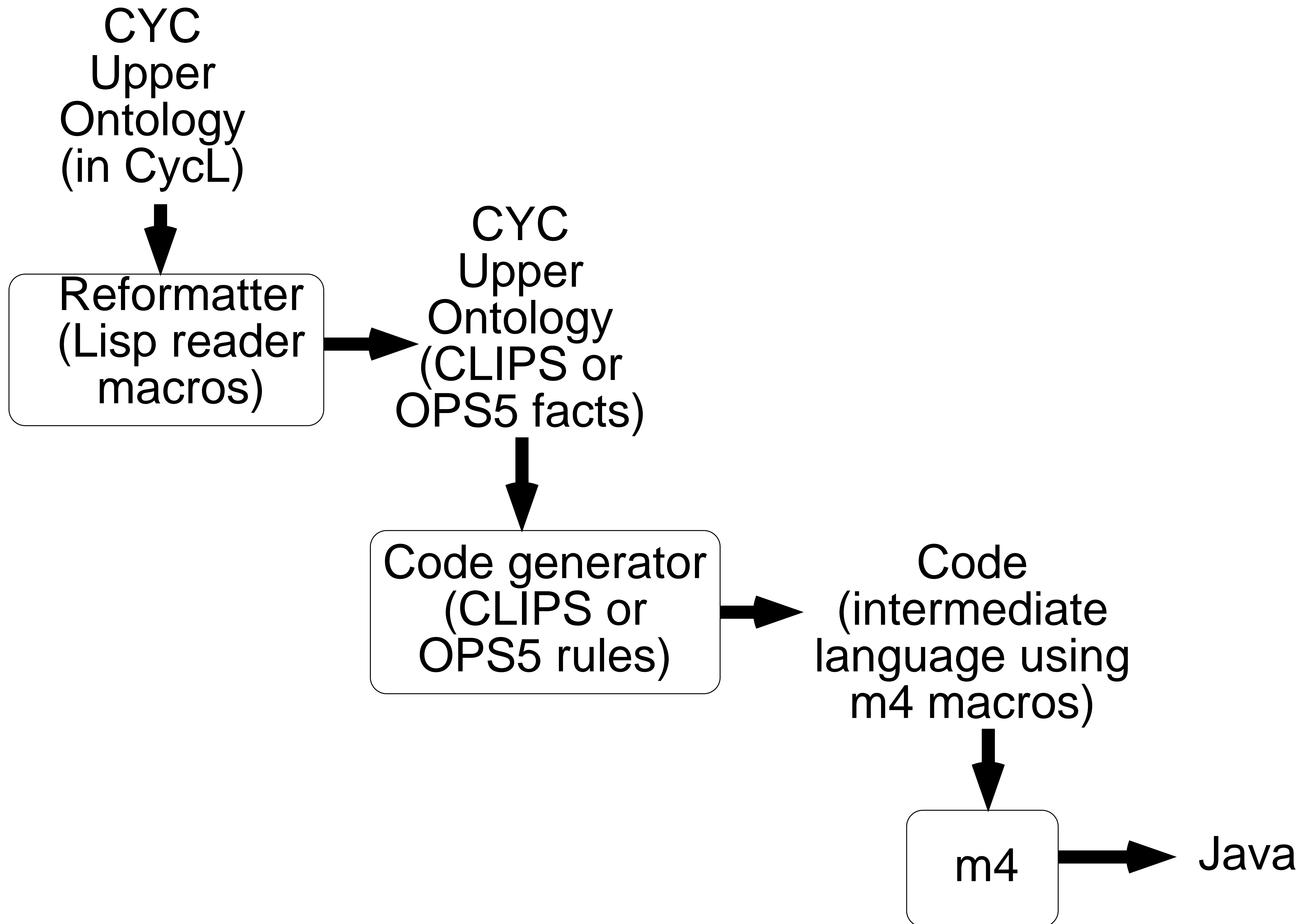
CYC Ontology



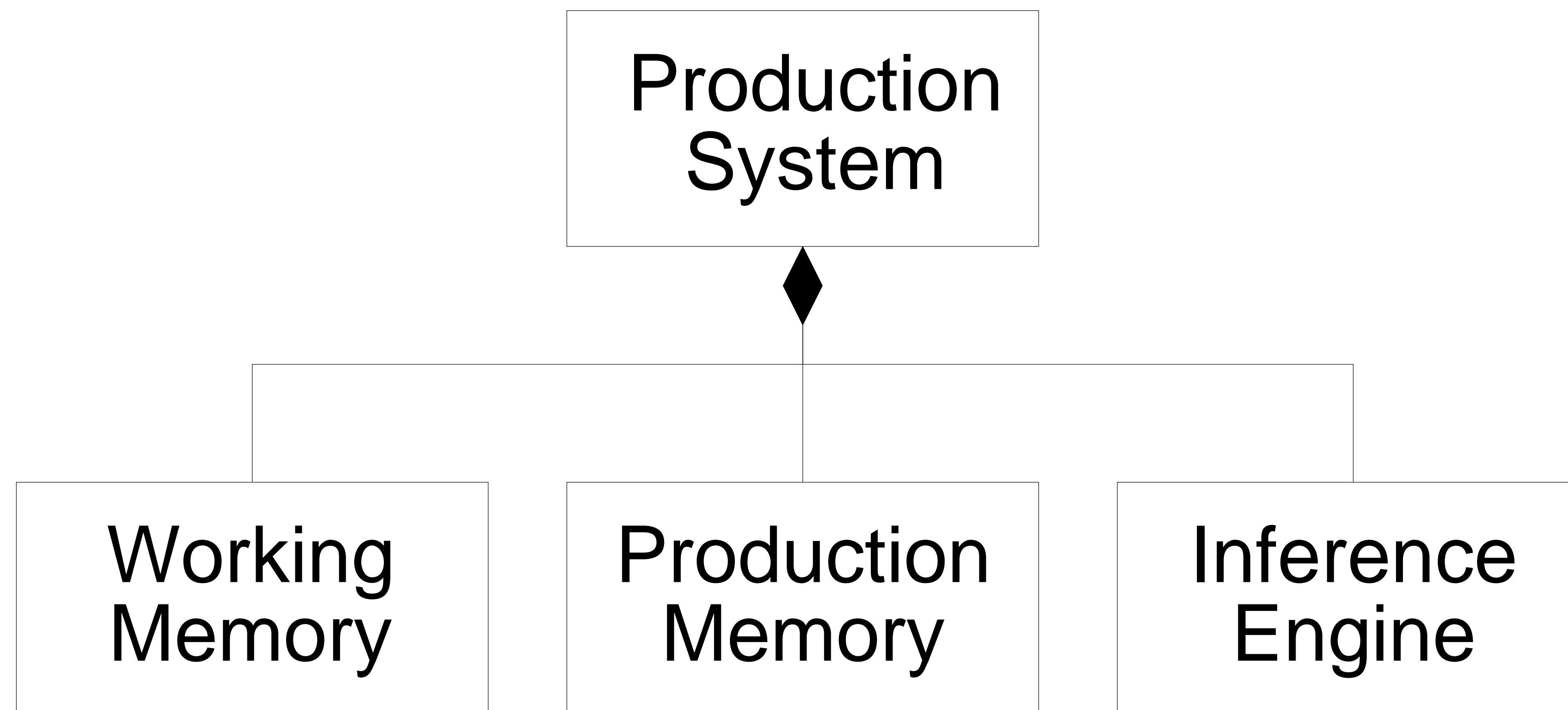


Java Object Model

Basic approach: A "pipe-and-filter" architecture.



The heart of the system is a code generator written using a forward-chaining product (rule-based) system.



Example of production system at work (using OPS5)

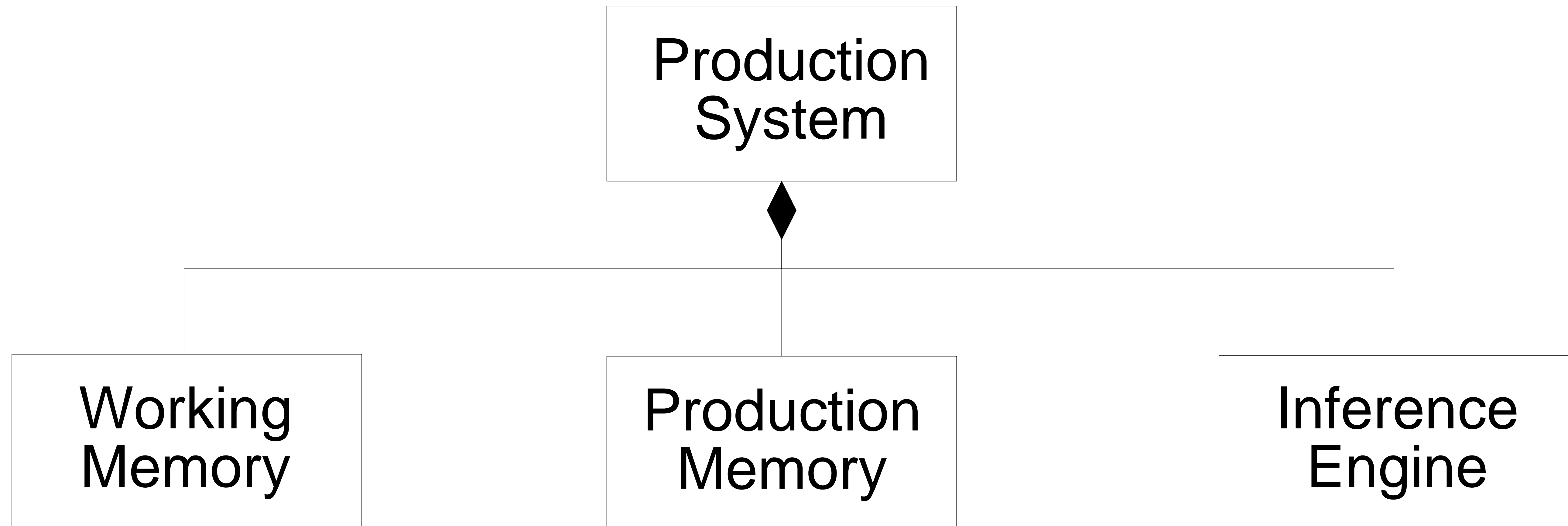
```
? (p infer-superclass
      (genls <a> <b>)
      (genls <b> <c>)
      - (genls <a> <c>)
      -->
      (make genls <a> <c>))
*
NIL
? (make genls cat mammal)
NIL
? (make genls mammal animal)
NIL
? (wm)

1:  (GENLS    CAT    MAMMAL)
2:  (GENLS    MAMMAL    ANIMAL)
NIL
```

Add rule to production memory

Add facts to working memory

Print working memory

Result before inferencing.




```
(GENLS CAT MAMMAL)
(GENLS MAMMAL ANIMAL)
```

```
(infer-superclass
     (genls <a> <b>)
     (genls <b> <c>)
     - (genls <a> <c>)
     -->
     (make genls <a> <c>))
```

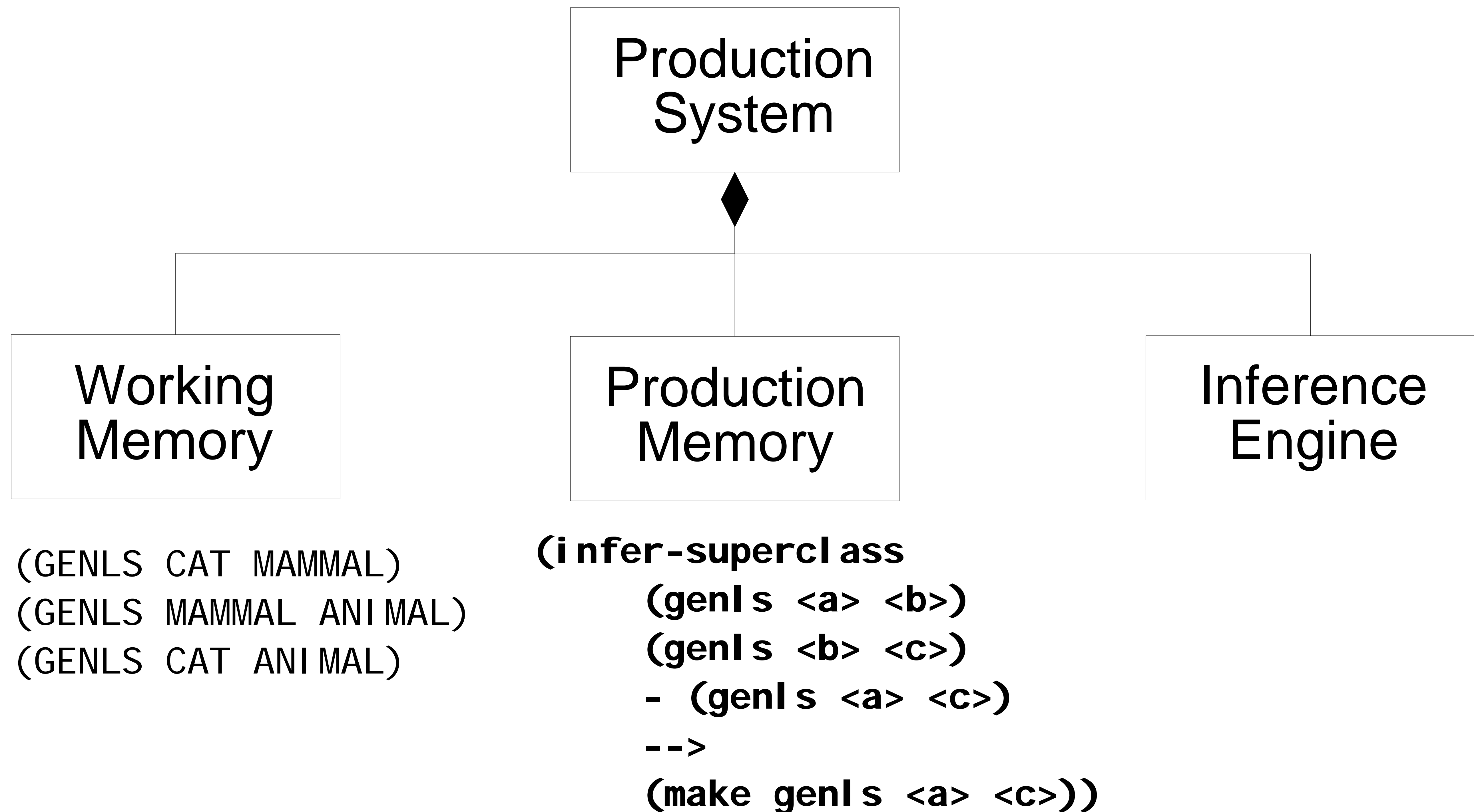
Apply inferencing.

```
? (run)
1. INFER-SUPERCLASS 1 2
end -- no production true
  1 productions (6 // 10 nodes)
  1 firings (3 rhs actions)
  2 mean working memory size (3 maximum)
  1 mean conflict set size (1 maximum)
  3 mean token memory size (3 maximum)
NIL
? (wm)

1:  (GENLS    CAT    MAMMAL)
2:  (GENLS    MAMMAL    ANIMAL)
3:  (GENLS    CAT    ANIMAL)
NIL
?
```

Invoke inference engine

Print working memory

Result after inferencing.



Examples of code generation rules:

Generalization

(genls A B)

```
public interface A extends B {
    ...
}
```

Binary predicate = attribute

(arg1Isa x A)
(arg2Isa x B)
- (arg3Isa x C)

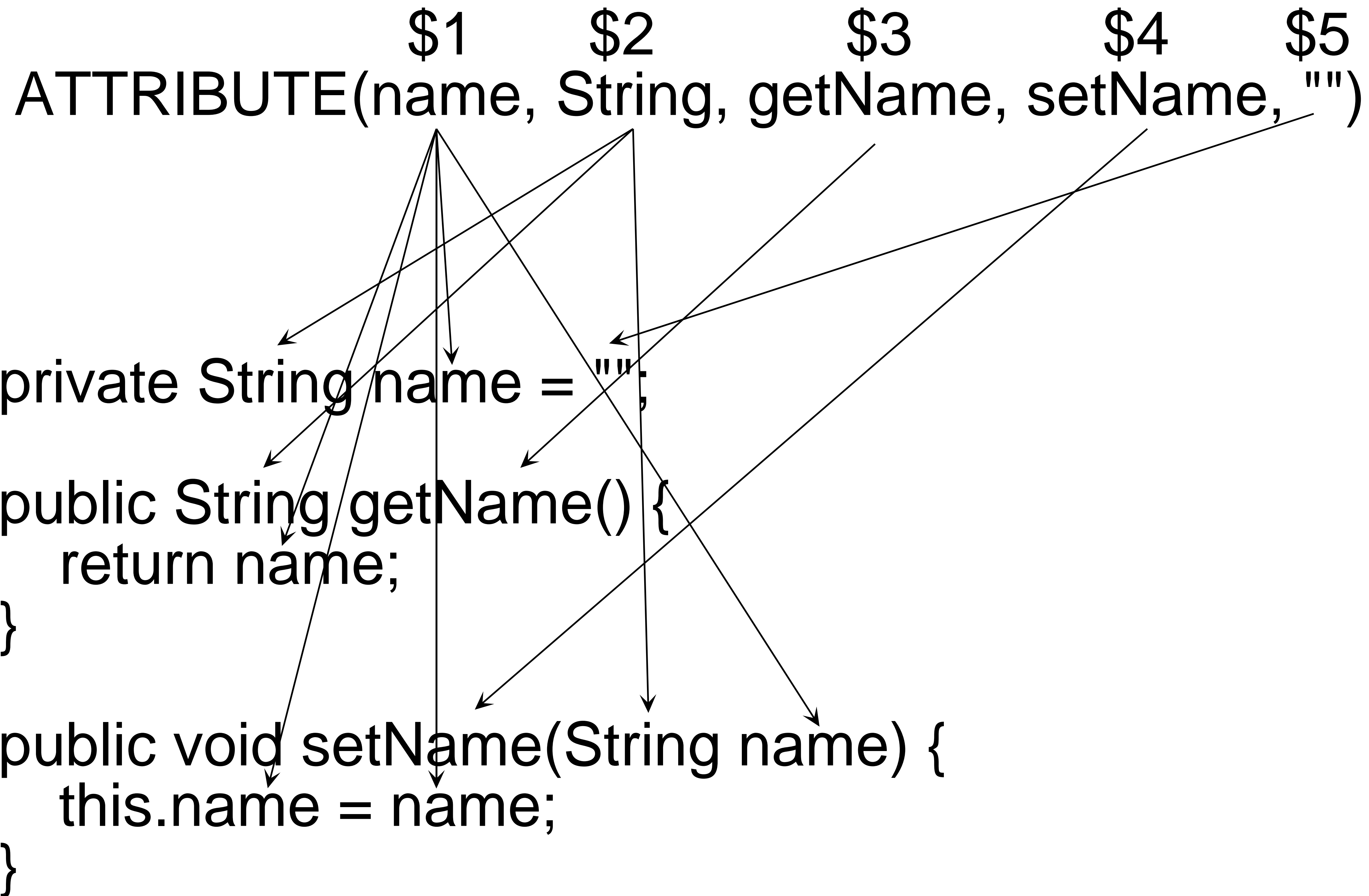
```
public class A ... {
    ...
    ATTRIBUTE(x, B, getX, setX, new X())
    ...
}
```

It was easier to target an intermediate language (written in m4 macros) rather than to generate Java directly.

The ATTRIBUTE macro implements part of the JavaBean naming "pattern".

```
define(ATTRIBUTE,
   private $2 $1 = $5;

   public $2 $3() {
      return $1;
   }

   public void $4($2 $1) {
      this.$1 = $1;
   }
)
```

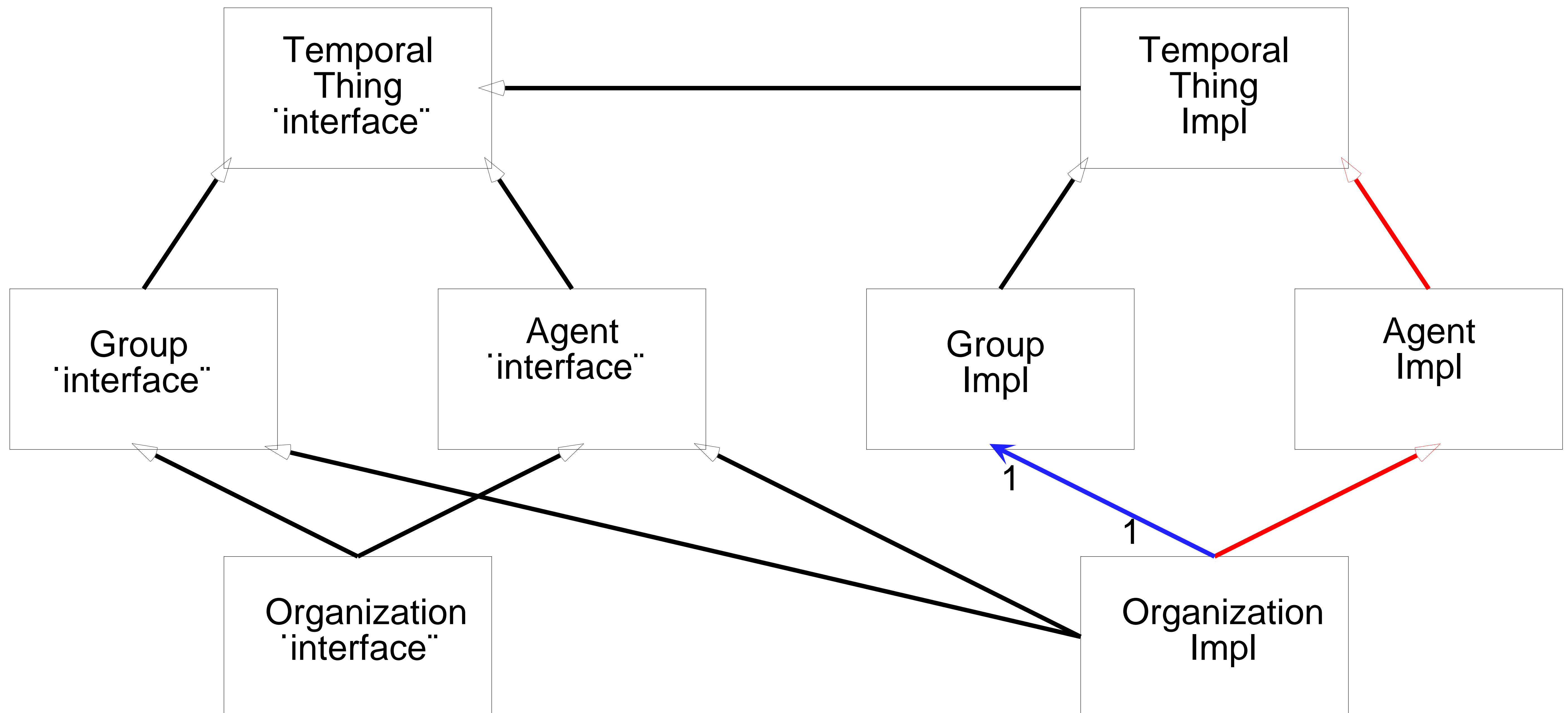
A typical ATTRIBUTE macro expansion



Problems in Java code generation

* Multiple inheritance.
* Multiplicity of relationships is not explicit in CYC.
* How do you represent higher-order classes?
* Relationships have relationships to other relationships.

Multiple inheritance ÑPick a main implementation inheritance line and delegate to an instance of the other.



There are several possible approaches to representing higher-order classes in Java.

1. Ignore them. (Many applications need to work at only one level of abstraction.)

2. Implement two levels (object and class) using static data for data about the class itself (e.g., the number of cats in the world).

3. Implement two (or more) levels as ordinary Java classes. Add an "isa" or "type" reference(s) from all objects (including classes) to the classes they are instances of.
   Problem: Duplicates "normal" inheritance structure at bottom level.

Our wish-list for Java features

The following would enhance the ability to use Java
as a knowledge-representation language:

* Multiple inheritance of classes

* Multiple return values from functions

* Relationships as first-class objects

* Generics

* Macros integrated with the language

* Higher-order classes

Our biggest mistake was...

Not extending the use of code generation to user interfaces.

* The large number of classes tended to overload GUI developers writing GUIs by hand.

* The GUIs were not guaranteed to match the ontology Ñ encouraged very late and ad hoc "ontological re-engineering".

Conclusions

* Developing the analysis model in an executable rule-based language has many advantages over the use of a purely graphical notation.

* Use of a standard ontology for such models is practical and reasonable.

* Automatic translation of the resulting model into Java (or other language) is feasible.

* All stakeholders must understand the reasons for standardizing on the ontology and be willing to operate under its framework. Any late "ontological re-engineering" ("we know better than CYC") can and generally will result in chaos and unmaintainability of the resulting system.

* Code generation should be applied throughout the process (including UIs), otherwise major bottlenecks due to hand-crafting of many individual components results.

Acknowledgements:

Much of this work was done at FGM, Inc. in support of the Tracker project of the Non-Proliferation and Disarmament Fund (NDF), of which the author was Chief Architect.

The results were successfully incorporated into the data model of the latest release of Tracker, Tracker 2001.

Tracker uses the Versant object database.

Lisp code was executed using Macintosh Common Lisp (MCL). (When will an OS X-compatible version be available?)

The rule-based system was initially implemented using the CLIPS expert system shell, originally developed at Johnson Space Center. It is being re-implemented for comparison in the OPS5 system, originally developed by Charles Forgey at CMU. And again, executed using MCL...

For further reference:

More details are at the author's home page:
home.att.net/~stephenstrom

For CYC: www.cyc.com

For CLIPS: www.ghgcorp.com/clips/CLIPS.html

For JESS (a Java version/extension of CLIPS):
http://herzberg.ca.sandia.gov/jess/

For Tracker: www.trackernet.org