Building a Large-Scale Generic Object Model: Applying the CYC Upper Ontology to Object Database Development in Java

Stephen W. Strom
FGM, Inc.
45245 Business Court, Suite 400
Dulles VA 20166
e-mail: strom@acm.org
Phone (mobile): 703-626-6663
Phone (office): 703-478-9881
Fax: 703-478-9883

April 3, 2000

Abstract

Can the benefits of re-use and patterns that are so well known in software design and implementation be carried over into analysis and database design? The fledgling field of Analysis Patterns has provided only a partial affirmative answer. We have instead attempted to import the CYC upper ontology (an AI-derived "model of everything") into an object model expressed in Java. The result is an object model that is extremely resiliant to change and should be extensible to practically any domain. We describe the difficulties involved and some of the most important lessons learned.

1 Background

As a development community, we are now building third- and even fourth-generation object-oriented systems.

What are we doing better?

- Systems are no longer being built entirely from scratch. Reuse and refactoring are occuring.

- Large systems can be built.
- Technologies for system intercommunication are becoming mature (CORBA, RMI, EJB, XML, etc).
- CASE tool technology is maturing and converging (for example, in the area of UML).

All of these improvements are on the design and implementation side. Object-oriented analysis has generally lagged behind:

- Tool support is still very weak.
- Each system has concentrated on building an analyis model that addresses its current requirements. These analysis models are usually codified into a more-or-less formal "domain model" set of classes that is encorporated into the final product, including the system database.

Results?

- 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.
- It is not possible to for disparate systems to easily intercommunicate.

The success of patterns in the area of software design has led to an attempt to apply the same concepts in analysis and has spawned the field of "analysis patterns". We have previously attempted to apply the concepts of analysis patterns to our current project (Tracker) with mixed results.

The Tracker project is being developed by FGM for the Nonproliferation and Disarmament Fund (NDF) of the US Governement. Tracker is an automated export control system that permits countries to track in real-time exports of proliferation concern, and to consult electronically (through an international network) with other government ministries and foreign governments. Tracker 2000 is being implemented using Swing; Java Enterprise

Edition technologies, including Enterprise Java Beans (EJB), Java Server Pages (JSP), and Java Messaging Service (JMS); all atop an object database (Versant).

The first generation of Tracker (Tracker 97) was built using a custom domain model specialized for the area of export license processing.

The next generation of Tracker (Tracker 99) used Fowler's analysis patterns [4] to successfully design a more generic domain model. Over time, however, it has been necessary to continue to tweak these models in ways that are not always backward compatible. Both Hay [5] and Silverston et al [6] have produced generic object models similar to Fowler's, but it seemed clear that these would suffer from the same long-term problems.

We began looking for other approaches.

In the AI world, Douglas Lenat proposed in the early 1980s that it was possible to build a "general ontology" which has the following characteristics (as stated by Russell and Norvig in their textbook on AI [1, p. 227])

- 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*."

The result of this proposal was the CYC project. A good introduction has been written by Lenat [2].The CYC (short for enCYClopedia) project was initially part of MCC but is now its own company. (See www.cycorp.com.)

The CYC project has released the "CYC upper ontology" for general use.[1] While just a tiny subset of the entire CYC database, the CYC upper ontology nevertheless dwarfs systems such as Fowler's and seemed to promise the long-term completeness and robustness we were looking for.

[1]Cycorp is providing this material from the Cyc(tm) Upper Ontology at no charge, for everyone to use, including commercial service use and incorporation into products. However, it is not "Public Domain." Please acknowledge Cycorp, 3721 Executive Center Dr., Austin, TX 78731 in any use or citation of this material, and request that each further user include a full copy of this notice as well, in any use or citation they make of the material. All these terms equally apply to renamings and other logically equivalent reformulations of the material in any natural or formal language. Cycorp intends to amend and expand the material from time to time; the latest version is available at http://www.cyc.com

Lenat has summarized CYC in these words: "One can think of CYC as an expert system with a domain that spans all everyday objects and actions." Is it similarly possible to build an object model "with a domain that spans all everyday objects and actions"? A model that is "applicable in more-or-less any special-purpose domain (with the addition of domain specific *c*lasses)"?

In particular:

- Could the CYC upper ontology be translated into a standard object-oriented language such as Java?
- Would the resulting "general object model" be immediately usable, in particular in Tracker 2000?

From another perspective, the question was: Did it make sense to express the domain model in a system of first-order logic and set theory, rather than in a notation such as UML? Could such a model be automatically forward engineered into Java?

FGM initiated an internal research project to answer these questions.

The next section of this paper contains relevant portions of the original proposal for the research project. The final section covers the results.

2 Original Proposal

2.1 Concept Summary

One of the main problems with the Web right now is that everyone is creating their own ad hoc mechanisms for dealing with their individual domains. Even if these separate domains are standardized, there is no easy way to talk across domains for data mining. For example, the census might have lots of information describing demographics, the medical community lots of information describing disease statistics, but because of the separate terminology used in each domain (it may be as trivial as not understanding that a "Patient" in a medical database is the same as a "Person" in another database), it is very difficult to combine the two sets of information. This is

described as having separate "ontologies" or descriptions of the world. This problem may become only worse (or at least more frustrating) as SGML and XML become more widespread. What is needed is an over-arching scheme that can be used cross-domain for queries, etc.

The creation of such a scheme would be a monumental task. Fortunately, it has already been done! Douglas Lenat and a team formerly with the MCC in Austin have generated the CYC database. CYC is short for enCYClopedia and is meant to capture the everyday knowledge that a person understands. The overall CYC database is huge. They have publicly released what is called the "CYC upper ontology" which captures the relationships between many objects as a set of rules. This goes beyond simple classification to look at higher-order relationships (what I have started calling "higher-order objects"). This is metadata that can be used for data mining and so on. The CYC upper ontology is the draft for a proposed ANSI-standard ontology.

As an example of what I mean, ordinary object-oriented development allows for classification like the following: A dog is a type of animal. A cat is a type of animal. Buddy is an instance of a dog. Socks is an instance of a cat. To add in a higher level, dog and cat can be considered instances of "animal species" which can have its own properties (e.g., "is endangered"). Ordinary OO schemes are inflexible and draw an absolute distinction between classes and objects. If I build a system in which cat and dog are classes, I can't make them instances of the class "animal species". A system like CYC allows for these higher-order relationships through a set of rules like the following:

- Buddy is-a Dog.
- Socks is-a Cat.
- Animal generalizes Dog.
- Animal generalizes Cat.
- Dog is-a AnimalSpecies.
- Cat is-a AnimalSpecies.

- Species generalizes AnimalSpecies.

2.2 Technical Approach

For our rule representation and production (expert) system engine, we will use CLIPS (developed at Johnson Space Center, and freely available) and its port to Java, the Java Expert System Shell (JESS) (available from Sandia Labs). JESS is extremely well integrated with the Java language, including Java beans.

The CYC upper ontology (several thousand rules) is publicly available from the CYC website (www.cyc.com). It is written in CYC's own knowledge representation language, CycL. A first step is to convert it to CLIPS format, which should be relatively straightforward, since both are based on Lisp syntax.

The lower-level database system will be built using an object database and the standard Object Query Language (OQL) and its Java bindings.

2.3 Tasks and Schedule

Task No.	Weeks from start	Task
1	1	Set up development environment, collect tools
2	2	Translate CYC to CLIPS/JESS
	3	
3	4	Build bottom-level ODB
4	5	CYC-to-OQL translation engine
	6	
5	7	XML input/output
6	8	Evaluation and completion of documentation

3 Results

Of the six major tasks identified, this paper will focus entirely on task number 3, which was the core of the effort.

In discussing the various issues we encountered in implementing this project, we will use the following highly-abbreviated and highly-simplified version of the CYC taxonomy.

```
Thing
  Individual
    AttributeValue
      Distance
      Time
      Mass
      Volume
    Product
    TemporalThing
      Event
      Agent
        Organization
        Animal
          Person
      Group
        Organization
  Collection
    StuffType
      ChemicalCompoundType
    ObjectType
      AttributeType
      BiologicalTaxon
        BiologicalKingdom
    ProductType
```

Note that CYC uses the term "collection" for the object development world's "class". The CYC upper ontology thus deals with both standard class relationships, and with relationships between classes themselves (higher-order classes).

Note that each subtype under Individual is an *i*nstance of one of the types under Collection. For example: Animal is an instance of BiologicalKingdom.

3.1 Translation of CycL into CLIPS

CycL has been implemented as an embedded language within Lisp. Our initial processing was therefore done using Macintosh Common Lisp.

As anticipated, it required only light editing (we did it using Lisp reader macros) to change the CycL statements into CLIPS rules.

3.2 Translation into Java

Java code generation is performed in CLIPS by rules which pattern-match on facts from the CYC upper ontology. Also, rather than directly generating Java code, we generate an intermediate language defined using the m4 macro language and then process that output using m4 to produce the final Java output. Perhaps not too surprisingly, we have modeled this intermediate language after the Common Lisp Object System (CLOS) [3, pp. 770-864]. For example, a class for a Person looks like this in our intermediate language:

```
public interface Person extends Agent
{
    ATTRIBUTE(firstName, String, getFirstName, setLastName)
    ATTRIBUTE(lastName, String, getLastName, setLastName)
   // and so on...
}
```

From a Java code generation perspective, we were initially interested in three kinds of CYC statements:

1. Generalizations (which define the class inheritance structure)
2. Instances (which define either objects or higher-order class relationships)
3. Two-argument predicates (which define attributes)

We were immediately confronted with the following issues regarding code generation:

- Multiple inheritance

- Higher-order classes
- Varying multiplicity of predicate arguments

3.2.1 Multiple Inheritance

We initially handled multiple inheritance by generating Java interfaces. We intended to provide a set of "default" implementations for each class (similar to Swing adapter classes for event listeners). There would be null implementations of each interface, and a developer could choose to implement whichever parts of the interface were desired.

To put it bluntly: This did not work.

When we processed the entire set of facts, this resulted in 1768 Java interface definitions, each containing zero or more attributes expressed using JavaBean conventions (for an attribute named `x`, both an accessor named `get`X and a mutator `set`X are created). This number of classes and attributes was simply overwhelming. Instead, it was necessary to prune the CYC ontology to cover only the problem at hand. While this may at first sound like a retreat from our original goals, we know that as new classes from the CYC ontology are needed they can be added without breaking the existing classes. (Versant can handle this kind of schema evolution without even taking the database offline.)

3.2.2 Higher-order classes

The existence of higher-order classes (what Fowler calls the "knowledge level") can be dealt with in either of two ways:

1. Plant a link in an object to a separate class defining its type. This is Fowler's recommended approach [4, pp. 26-28].
2. Utilize the Prototype design pattern [7, pp. 117-126].

We chose the first alternative for now but are continuing to explore the second.

3.2.3 Attribute multiplicity

At first, we handled the problem of variable multiplicity by simply having each accessor return an array, thus treating each relationship as intrinsically 1-to-many or many-to-many, regardless of the fact that the true representation of the relationship might be 1-to-1 or many-to-1. This has proven too confusing and we have altered the accessors to return only a single value where appropriate.

3.2.4 Units of Measure

In Tracker 99, we closely followed Fowler's work in dealing with units of measure [4, pp. 36-39].

To summarize:

- Attributes are represented as Quantities.
- Each Quantity includes a reference to its units and includes a full set of "algebraic operations".
- Each Unit carries a set of "from/to" conversion ratios to other texttt Units, along with a convertTo operation.

In practice, this caused several problems:

- Maintaining the conversion ratios from unit to every other unit proved to be tedious and error prone, and generated significant overhead in, for example, serialization.
- It was possible to mix quantities of different types, e.g., distance and mass, and only detect the error at runtime when no conversion ratio could be found.

In Tracker 2000, we have introduced different mechanisms, modeled after CYC's approach:

- There is an AttributeValue type, similar to Fowler's Quantity type, but it is abstract.

- Different types of attributes are subclasses of `AttributeValue`, e.g., there are `Distance` and `Mass` types.
- All values are stored in internal units (international metric or SI wherever possible). Conversion between units happens only on demand, at input and output.
- Values of units are created using factory methods in a `UnitOfMeasure` class, e.g., there are `newDistance` and `newMass` methods which take numbers and units strings as arguments and generate new attribute values.

3.3 Customization of displays

One of the very nice features of Java is its support for internationalization and localization. We have used this to customize terms from the domain model as they may be presented to the users in a particular application. For example, in the Tracker domain, the term "Party" is used rather than "Agent". We use "Agent" as a key to look up the value that can be displayed. The value can be customized by locale (including US English!).

3.4 Desireable extensions to Java

This is our own plea for a set of enhancements to the Java programming language which would ease this kind of work.

1. Multiple inheritance of classes
2. Multiple return values from functions
3. Relationships as first-class objects
4. Generics
5. Macros
6. Higher-order classes (not metaclasses)

4 Summary

It is, indeed, possible to take a domain model expressed in first-order logic and (largely) automatically generate Java code from it. The CYC upper ontology is a powerful (indeed, sometimes too powerful) system to use as the basis for such a domain model. The problems encountered in implementing such a system point to possible extensions of the standard object model itself (into areas such as higher-order classes) as well as to "tweaks" of various magnitudes which could greatly improve the expressiveness and extensibility of the Java programming language.

Additionally information, including an extended version of this paper, a diagram of the parts of the CYC upper ontology that we used, and code examples, is available at the author's web site: http://home.att.net/ stephenstrom.

References

[1] Stuart Russell and Peter Norvig. 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

[2] Douglas B. Lenat. 1995. "CYC: A Large-Scale Investment in Knowledge Infrastructure." *COMMUNICATIONS OF THE ACM*, November 1995/Vol. 38, No. 11

[3] Guy L. Steele, Jr. 1990. *Common Lisp: The Language*. Second Editon. Digital Press.

[4] Martin Fowler. 1997. *Analysis Patterns: Reusable Object Models*. Addison-Wesley.

[5] David C. Hay. 1996. *Data Model Patterns: Conventions of Thought*. Dorset House.

[6] Len Silverston, W. H. Inmon, and Kent Graziano. 1997. *The Data Model Resource Book: A Library of Logical Data Models and Data Warehouse Designs*. Wiley.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.