Metamodeling in OO
OOPSLA'95 Workshop summary

Hafedh Mili[1], Francois Pachet[2], Ilham Benyahia[3], Fred Eddy[4]
1. University of Québec at Montréal (Canada), 2. University of Paris VI (France)
3. Institut de recherche d'hydro-québec (Canada), 4. OMT Consulting Inc. (USA)

1. Introduction

The purpose of this workshop was to explore metamodeling concepts in the context of object-oriented software development, from a methodological perspective, namely:

1) Exploring the additional modeling constructs that are needed to model information at several levels of abstraction,
2) Exploring changes to modeling procedures/heuristics that are needed to build object meta-models and integrating them with object models,
3) Exploring the effects of such changes on design and implementation (e.g. reflective languages, metaprogramming, etc).

As is usual (and healthy) in such events, the papers we received did not fit in exactly with the advertised agenda, with its heavy emphasis on methodological (analysis and design) aspects, and the proposed agenda was as much the result of what the organizers wanted discussed as it was a result of what "participants" were willing to discuss! That result is shown next. We first attempt to define metamodeling in the context of object-oriented software engineering, identifying major issues. Next, we discuss each one of those issues in some detail.

2. General issues

Object-oriented structural modeling uses classes to represent the structure of similar application objects, and associations to represent patterns of connections between application objects. Applications where the representation of objects needs to be queried or otherwise manipulated need to explicitly represent the representation of objects [Diaz,1994]; we call the representation of the representation of application objects *metamodeling*.

The need for metamodels is more frequent than one might first think. In relational database modeling, meta-data consists of table descriptors, which are system tables (system dictionary) describing the data tables (their columns, domains for the columns, etc) and integrity constraints, which, by and large, are semi-declarative constructs to be executed during database updates. [1] A typical computer-assisted manufacturing application needs two levels of abstraction: 1) one level to represent bills of material, e.g., describing the compositions of different manufactured products, 2) a second level to describe actual manufactured products in stock. Bills of material are representations of manufactured products. However, to the extent that they can be created, consulted, and modified, they too need to be represented by a construct that describes their structure; these are called *metaclasses*.

When not a part of the problem (e.g. a bill of materials *and* inventory application), metamodeling is part of the solution: metamodeling is an *abstraction* mechanism in the sense that, much like classification, it enables us to differentiate identity-dependent behavior from state-dependent behavior, hence making some sense of the variations in the world to be modeled. Also, much like classification, by replacing *extensions* (explicit occurrences of a given concept) by *intensions* (the definition of the concept itself), metamodeling results into *cognitive economy*. Finally, when applied in actual software projects, metamodeling can result in great savings in development effort, and in much better reuse potential by going one additional step from component-oriented reuse towards generation-oriented reuse.

For the purposes of OO software engineering, when we talk about metamodeling, we have to talk about at least three sets of issues:

1) An anthology of metamodeling: what does it mean, what kinds of metamodeling "dimensions" are there, how many are really needed for practical software engineering, etc.,
2) A process for metamodeling: how to perform metamodeling, how to recognize and/or buiild metamodels from application/business requirements, etc., and
3) Implementing metamodels and meta-programming: how to represent metamodels at the analysis, design, and implementation levels, what additional notational constructs are needed, what kind of support do programming languages need to provide, etc.

The position papers submitted and the discussion that took place during the workshop focused mostly on the first (anthology) and third (implementation/support) sets of issues. These are discussed in more detail below.

3. An anthology of metamodeling

Three dimensions of metamodeling emerged from the papers submitted:

1) Metamodeling as the modeling of modeling/representation languages (see e.g. [Artsy., 1995],
2) Metamodeling as the multiple instantiation levels of application knowledge (see e.g. [Odell, 1995a] and [Odell, 1995b]), and
3) Metamodeling as the modeling of information about how to use and manipulate application models (see e.g. [Anquetil & Vaucher, 1995], and to some extent, some aspects of system described by Demphlous [Demphlous & Lebastard, 1995]) as *computational objects*.

It was clear to us that all three kinds of information were useful and may be combined in any one application area. We attempted to identify a common framework within which the three dimensions and arbitrary combinations thereof could be studied

One way of looking at this considers computer application models as the cross-product of three models or *domains*: 1) the business/application domain, call it D_A, 2) the computational model, call it D_C, and 3) the representation model, call it D_R. The three dimensions mentioned above correspond to abstraction along one of three domains, where relationship between a model and its meta equivalent is that between an instance and a set (or a class). In other words, if the realm (set) of computer application models is the cross product $D_A \times D_R \times D_C$, then one metamodeling *direction* might be $D_A \times 2^{D_R} \times D_C$, where 2^S is the *power set* of set S. In this first example, we were "going meta" along the representation/notation language dimension; another might be $2^{D_A} \times D_R \times D_C$, i.e. "going meta" along the application knowledge.

We could also iterate several times along the same dimension. For example, we can "go meta-meta" along the representation notation/language dimension, to get the realm $D_A \times 2^{2^{D_R}} \times D_C$. Yet another meta-meta direction might go one step in the representation language direction, and one in the application domain direction, yielding the modeling realm $2^{D_A} \times 2^{D_R} \times D_C$. And so forth.

Much of the discussion about the anthology of metamodeling and the above formalization benefited from two position papers ([Odell,1995a] and [Odell, 1995b]) and two presentations by Jim Odell. The remainder of the workshop was to go over the three dimensions mentioned above, before moving on to issues of supporting metamodeling.

4. Metamodeling as the modeling of representation languages

The one dimension that received the most attention is the first one, with various applications thereof, including maintaining consistency between object models [Sarver,1995], implementing CASE tools [Sahraoui et al.,1995], a uniform way of supporting persistence [Demphlous et Lebastard, 1995], data interchange between tools (e.g. [Odell, 1995a] and [Artsy,1995].

Artsy, who leads a standards group working on establishing a common data interchange format (CDIF) between tools, gave the introductory talk. The approach taken by the standards group consists of expressing the various modeling languages used by the various tools and methodologies in a single meta-modeling language, so that correspondences and transformations could be applied between models expressed within (exported by) the various methodologies (tools). He discussed some of the practical issues in finding such a CDIF, including the fact that most methodologies and tools have no formal (or reliable or consistent) description of their semantics. He reported that the group has finished work on data (static) models and that ongoing efforts continue towards bridging dynamic/behavioral modeling notations.

Sahraoui described the use of metamodeling for generative programming. METAGEN, a system developed at the LAFORIA lab of the University of Paris VI, supports generative programming by supporting the description of analysis/modeling notations, of design notations, and of correspondence rules (mappings) between them. Specific application models are input as instances of the user modeling language, and the computational model (design model) is output as an instance of the design language through the triggering of transformations. He described his experiences using METAGEN to replicate a database modeling and programming tool called INTERSEM, developed by Prof. Missaoui of the Univ. of Québec, which inputs extended E/R schemas, and generates ObjectStore schemas and access functions.

The general discussion addressed general issues such as: 1) is there really a (meta-) modeling esperanto, 2) what kind/family of languages is/are guaranteed to not be outdone by the Booch notation, otherwise known as the APL of design notations, 3) can we avoid having to fall back on GP representation languages such as logic, 4) how to build extensibility in such a language so that new constructs can be introduced without invalidating earlier specifications. Artsy stressed that the goal of his CDIF group is a best effort approach, i.e., they accept loss of information during interchange; they just wish to minimze it and understand its nature.

5. Metamodeling of application knowledge

The second dimension of metamodeling has received little attention as a subject of study per se in the community, and this was reflected in the submissions. Jim Odell introduced the notion of *powertypes*, which inspired much of the formalization of metamodeling discussed earlier. A *powertype* is to a type what a power set is to set, i.e. a power type is the type (class) whose instances are subtypes (subclasses) of another type. We come across powertpes very frequently in OO modeling, as when we have something to say about term deposit accounts, rather than about a particular kind of term deposit account of a particular customer. Odell proposes a way of representing types and powertypes in the same class (type, we should say) diagram, helping to clarify object models and to maintain consistency between "type" and "meta-types".

Fred Eddy, formerly of GE Corporate Research and Development, described an avionics application in which they were mandated to develop a monitoring and control application that could accommodate variations in aircraft models. He discussed two alternatives, parameterization and meta-modeling. Parameterization accommodates data differences quite handily, but does not account for process differences as easily or as cost-effectively as meta-modeling. In this case, "going meta" meant: 1) developing a conceptual computational model for the family of applications at hand, 2) designing an architecture for simulating and executing such applications, and 3) implementing such an architecture in the form of a development platform/framework. The gains in terms of productivity and time-to-release were immeasurable[1]

Michael Mohammed, of PanCanadian Petroleum, described ongoing efforts at PanCanadian to re-engineer its information infrastructure. A cornerstone of that strategy consists of identifying commonalities between its various information bases and processes. Metamodeling was used to package the description of the common information structures and processes.

A number of issues remain to be addressed with regard to the use of meta-modeling to model different levels of abstraction of the application domain. First, from a practical standpoint, how do we distinguish between a type property/attribute (e.g. Smalltalk's metaclass instance variables) and a property whose value is shared by the instances of a class (Smalltalk's class variables, C++'s static data members). We also need behavioral metamodeling (a definition and a notation), along the lines of the experiences reported by Fred Eddy. It is interesting to note that standardization efforts have so far succeeded in developing a meta-level object model, but work continues on behavioral models [Artsy,1995].

6. Metamodeling as a user manual for objects

The third dimension of metamodeling corresponds to the concern addressed by Anquetil and Vaucher [Anquetil and Vaucher, 1995]. In that paper, they look at the problem of expressing usage constraints on reusable components, and more specifically, on the representation of negation or mutual exclusion between classes; while mutual exclusion between sibling classes is represented in some OO models, it is not general enough to handle unrelated classes. Further, existing OO languages and library environments do not support such exclusion.

Stephane Demphlous, of INRIA (France), presented their approach to manage persistence. It is based on a meta-modeling language which has its own meta-object protocol for managing persistence. Different users using different modeling notations will express their modeling notation in the meta-modeling language, and specialize the persistence meta-object protocol.

Other kinds of usage constraints discussed during the workshop have to do with the run-time "programmatic behavior" of objects. For example, in a distributed environment, we may want to say that classes of a given type are remote and that some updates have to be delayed, and/or queued, or that some objects shouldn't be loaded into the environment unless they are needed, etc. Such information can be represented and manipulated. Ira Forman raised the distinction between distribution (or any other programmatic paradigm) as part of the requirements, or as design choices made for performance or fault-tolerance, and wondered whether the distinction made a difference in the way the information is represented.

7. Providing support for metamodeling

7.1. Background

The most tangible meta-modeling achievements were historically made by work on reflection in programming languages, also referred to as computational reflection. Historically, work on computational reflection dates back to Brian Smith's seminal 3-Lisp system, which inspired an entire research

[1] i.e., nobody remembers :-)

branch on reflection in the functional programming community. This work gave then birth to various attempts at computational reflection in OO languages, mainly Smalltalk and CLOS (see the numerous OOPSLA workshops on reflection in OO languages, and the recent IJCAI workshop on meta-level architectures in AI). Other attempts at computational reflection were made in the community of logic programming (see e.g. the recent Godel language).

These efforts have identified two kinds of reflection, **structural reflection** and **behavioral reflection**. **Structural reflection**, which can be defined as the "ability of a language to provide a complete reification of both the program currently executed as well as a complete reification of its abstract data types". Structural reflection is the basis for a number of metaclass models. The purest (and minimal one, in a canonical sense) one is the ObjVlisp model. The most practical one is Smalltalk. CLOS is in the middle; in practice, it is the simplest one to implement. **Behavioral reflection** can be defined as "the ability of the language to provide a complete reification of its own semantics (processor) as well as a complete reification of the data it uses to execute the current program". Behavioral reflection is more difficult to implement. Work in OO behavioral reflection showed that in order to give access to the main phases of message interpretation, message execution can be divided in two phases: method lookup and application of messages. Each phase can then be redefined by the user to control message interpretation. In CLOS, this is done via redefinition of function apply-generic-function. In Smalltalk this is not possible in practice, but the language can be easily extended to do so (see work on ClassTalk (Briot&Cointe).

For the purposes of this workshop, two sets of issues are of interest: 1) implementing meta-models using reflective OOPLs, and 2) implementing meta-models without reflection.

7.2. Metaclasses and metaprogramming: major issues

From a practical standpoint, the utility of metaclass models, when they are available, is still problematic. The most visible problem is perhaps the "metaclass compatibility problem", raised by N. Gaube (OOPSLA 89). This problem comes from the fact that metaclasses, in reflective OO models, control their direct instances (classes), AND also their 2nd generation instances (instances of their instances), in an implicit way, which is not supported by the language. Hence, the programmer may define classes which are inconsistent, if he/she does not respect these compatibility constraints.

With both Pierre Cointe (ClassTalk) and Ira Forman (SOM) in attendance, we anticipated a shoot out. Fortunately (or unfortunately) François Pachet's dispassionate and to the point introduction to the topic, drew the battle lines in such a way that confrontation took place. Ira Forman presented SOM's solution to this problem, based on the notion of "derived metaclasses", which are dynamically created metaclasses, that ensure compatibility and *cooperation* between parent classes; he described a technique for combining before- and after- methods in a systematic and semantically meaningful way.

There then followed some discussion about the semantics of metaclasses, and of subclassing between metaclasses. Pierre Cointe mentioned the case where one metaclass provides a "self shouldNotImplement" implementation of a class method, which could be combined with a more constructive implementation of that method from another metaclass. Ira suggested that his approach is meant to work in cases of "positive attributes" (behaviors that classes do have to support), and not in cases involving cancellation/negation. Hafedh Mili ventured a set theoretic interpretation of Ira's metaclass compatibility constraints by treating metaclasses as powersets.

7.3. Doing without metaclasses and metaprogramming

Most of the widely used OO languages (C++, Eiffel) do not have support for metaclass programming. This raises two questions: 1) the question of the adequacy of the metaclass programming to implement metamodels, and 2) When metaclasses are not available, what kind of mapping can be done to simulate metaclass programming?

In his introductory presentation, François Pachet mentioned that experience has shown that, notwithstanding the incompatibility problems mentioned above, metaclass programming is difficult. Further, a fewer languages than we might think actually support unrestricted metaclass programming: he reminded Smalltalk hawks that Smalltalk supports only one level of metaclasses, but that real applications may involve several meta-iterations, to which Jim Odell nodded very forcefully. François mentioned an ongoing effort at the LAFORIA lab (University Paris VI) in which a "zig zag" approach was used to emulate a multi-level meta-architecture: metaclasses delegate their behavior to regular classes, enabling developers to program an arbitrary number of instantiation levels. He mentioned that, surprisingly, they didn't run into as many problems as with languages supporting unrestricted metaclasses. Hafedh Mili hypothesized that the reason was that developers had to be explicit about what behavior to carry across meta-class links, since they couldn't rely on built-in mechanisms.

Todd Blanchard described a C++ implementation in which they emulated metaclass programming. His approach is similar in spirit to François's "zig zag" approach in the sense that, unable to add an additional instantiation level for active objects to the language, we use low-level object to stand for higher-level ones, and provide programmed (ad-hoc) links between the two. What I really mean is, he represented classes by instances of some C++ class called (say) Class, hence, providing support for querying classes, during run-time, about their attributes and their relations to other classes (sorry Todd, I can't remember how you maintained consistency between actual C++ classes and their representation, or how you managed dispatching). He mentioned the danger of slipping towards developing yet another programming language.

8. General impressions

We came out with the impression that:

1) There is widening recognition for the need for metamodeling, and for integrating it into modeling methodologies, tools, and programming practices,
2) It is far from clear that *metaclasses* (not *metaprogramming*, which can be done without metaclasses) is the way to support metamodeling and metaprogramming,
3) There seems to be precious little investigation into the semantics of metaclasses from a type theory perspective, and quiet a few PhD theses can probably be had in this area.

9. Bibliography

Due to space limitations, we only list outside publications. Workshop submissions can be accessed on-line through the web URL: http://saturne.info.uqam.ca/Labo_Recherche/Larc/metamodeling-wshop.html. We should point out that the URL listed in OOPSLA's WEB pages was wrong (our mistake).

[Briot & Cointe, 1989] J.-P. Briot and Pierre Cointe, ''Programming with explicit metaclasses in Smalltalk80,'' *Proceedings of OOPSLA'89*, pp. 419-431.

[Diaz, 1994] O. Diaz and N. Paton, ''Extending ODBMSs using metaclasses,'' *IEEE Software*, vol. 11, no. 3, pp. 40-47, May 1994.

[Graube, 1989] N. Graube, ''Metaclass compatibility,'' *Proceedings of OOPSLA'89*, (October 1-6, 1989), pp. 305-316.