Metaclass Mechanisms: a CLOS perspective for Reifying Relationship *

Manuel Kolp † Alain Pirotte † Esteban Zimányi ‡

† Université catholique de Louvain, IAG-QANT
1 Place des Doyens, 1348 Louvain-La-Neuve, Belgium,
e-mail: kolp@qant.ucl.ac.be, pirotte@info.ucl.ac.be,

‡ Ecole Polytechnique Fédérale de Lausanne,
DI-LBD, IN-Ecublens, CH-1015 Lausanne, Switzerland.
e-mail: Esteban.Zimanyi@epfl.ch

1 Introduction

Since the recognition of the software crisis in the mid-70's, a huge amount of work has been done to devise modeling tools that help analysts to develop systems of ever-increasing complexity. While "structured methods" were appropriate for describing systems only driven by data transformations, they were not appropriate to cope with more sophisticated technologies. These include OO database systems, World Wide Web and network applications, hypertext, multimedia and CAD/CAM systems, distributed computing or sophisticated man-machine interfaces.

The object-oriented paradigm (software development methods [3, 11] and programming languages [1, 5, 12]) aim at remedying the situation. This led to a new way of thinking about problems with complementary models organized around real-world concepts: object models describe the structure of objects in a system; dynamic models represent aspects of a system concerned with time, control, and behavior. In these models, the fundamental construct is the object, organized in classes of similar objects (for instance, we have a class employee and a class department when modeling a company, which combines both data structure and behavior in a single entity).

In object-oriented applications, objects and classes are connected and cooperate to perform collective tasks and maintain some invariants. From a structural point of view, such connections are generally modeled in object models by semantic relationships.

Intuitively, a relationship describes a group of physical or conceptual connections between objects. For example, an employee works for a departement. All the connections in a relationship connect objects from the same classes. Relationships often appear as verbs

*This work is part of the YEROOS (Yet another project on Evaluation and Research on Object-Oriented Strategies) project, principally based at the Universities of Louvain. (http://yeroos.qant.ucl.ac.be)

in a problem statement and describe a set of potential connections in the same way that a class describes a set of potential objects.

1.1 Significant Problems of Relationship Implementation

Most elements of object models have OO language counterparts: the class mechanism models abstract data types, while attributes and methods support, respectively, properties and behaviors, const and reference types enforce immutability, inheritance implements generalization, virtual and generic functions correspond to polymorphism, and templates realize genericity. But relationships, other than generalization, are not supported by OO languages and systems. Consequently, developers have to implement this capability themselves.

Generally, relationships are implemented by unidirectional or bidirectional pointer-based mechanisms or using attributes containing references to related objects. However pointer-based implementations are *adhoc* structures which do not carry special semantics, not even referential integrity that must be explicitly enforced.

This leads to various problems, including the duplication of information among participating objects, the dispersion of the relationship knowledge, and the need to know beforehand all the relationships that an object can potentially be part of. Moreover, since relationship mechanisms are embedded, sometimes buried, in class definitions, no attribute or method can be specifically attached to them. Developers cannot thus uniformly send messages or apply operations to a relationship as a whole; they can only single out one of the objects in the relationship as the target of a method. The extension of a relationship cannot be changed by operations to add or delete elements; it cannot be queried by methods to test membership of objects, to select a subset of objects satisfying some conditions, or to iterate over the entire extension.

1.2 Reifying Relationships with Metaclasses

An alternative solution is thus to view a relationship between objects as a full-fledged object storing all relevant information [10, 6]. The ability to apply operations to the entire relationship rather than to related objects in it allows many expressions to be written concisely. Hence, relationships between classes must be viewed and implemented in turn as full-fledged classes managing relationship objects [4]. This leads to reify *relationship* as a first-level construct, i.e, to give it the same status and features as application classes: abstraction and encapsulation of the relationship semantics, independence from other classes, access to relationship properties and behavior through public interface and independent definition.

We first define a model of relationships including *generic relationship*, *aggregation (part relationship)*, *aggregated relationship*, *derived relationship*, and *materialization*. A relationship describes a group of connections between objects; an aggregation relates composite objects to component objects; an aggregated relationship describes a relationship participating in another relationship; a derived relationship is inferred from a join of other relationships; a materialization describes a relationship between an abstract class and a concrete class.

We then propose a Metaobject Protocol allowing to reify relationships between application classes so that they can be integrated into a reflective system (e.g., CLOS [1], Dylan, ObjVLisp [2]).

A reflective system (CLOS [1], Dylan, ObjVLisp [2]) incorporates structures representing static and dynamic aspects of itself [8]. The sum of these structures is called the self-representation of the system. This self-representation makes it possible for the system to answer questions about itself and support actions on itself.

Reflection is based on three principles.

(1) The system consists of a *two-level architecture*:
   - the *base level* contains the elements of the system and provides primitives for creating and manipulating objects;
   - the *metalevel* represents the model of the reflected system. Concepts of the metalevel are managed by a *metaobject protocol*. This protocol can be defined as:
     - a set of public hierarchies of metaclasses whose instances implement the concepts of the system;
     - a protocol that describes when metaobjects are created, which messages must pass between them, and which parameters are to be used.

(2) The two levels are *causally connected* so that any change in the metalevel leads to a corresponding effect upon the base level.

(3) This two-level architecture can be generalized to more than two levels; each level is reflected at the next higher metalevel.

The metaclass mechanisms allow to tailor a system to the needs of a particular application. Specifically, each kind of semantic relationship defined in our model is integrated through the definition of a metaclass which endows its instances (i.e, relationship classes between application classes) and their instances (relationship objects between application objects) with properties and behavior befitting the desired semantic relationship [7]. These different types of relationship are implemented as specializations of the *generic relationship* from which they inherit a default semantics.

This reflective meta-architecture using the capability of the meta-interpreter of CLOS also allows us to consider relationships between classes and objects in a schema evolution since developers should be able to dynamically manipulate (define, modify, redefine, and delete) a relationship or application class independently of other classes.

References

[1] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common lisp object system specification, x3j13 document 88-002r. In *Common Lisp, The language*, chapter 28, pages 770–864. Digital Press, second edition, 1990.

[2] P. Cointe. Metaclasses are first class: the ObjVlisp model. In Meyrowitz [9], pages 156–167. ACM SIGPLAN Notices 22(12), 1987.

[3] D. Coleman, F. Hayes, and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Trans. on Software Engineering*, 18(1):9–18, Jan. 1992.

[4] S. Ducasse. *Intégration réflèxive de dépendances dans un modèle à classes.* PhD thesis, Université de Nice-Sophia Antipolis, France, 1997.

[5] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[6] M. Halper, J. Geller, and Y. Perl. An OODB part relationship model. In Y. Yesha, editor, *Proc. of the 1st Int. Conf. on Information and Knowledge Management, CIKM'92*, LNCS 752, pages 602–611, Baltimore, USA, Nov. 1992. Springer-Verlag.

[7] M. Halper, J. Geller, Y. Perl, and W. Klas. Integrating a part relationship into an open OODB system using metaclasses. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proc. of the 3rd Int. Conf. on Information and Knowledge Management, CIKM'94*, pages 10–17, Gaithersburg, Maryland, Nov. 1994. ACM Press.

[8] P. Maes and D. Nardi, editors. *Meta-level architectures and reflection.* North-Holland, 1988.

[9] N. Meyrowitz, editor. *Proc. of the 2nd Conf. on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'87*, Orlando, Florida, Dec. 1987. ACM SIGPLAN Notices 22(12), 1987.

[10] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In Meyrowitz [9], pages 466–481. ACM SIGPLAN Notices 22(12), 1987.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[12] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1993.