Metamodeling for Multidimensional Reuse

Hafedh Mili[1] and François Pachet[2]
[1]Department of C. S., University of Québec at Montréal
P. O. Box 8888, Downtown Station
Montréal, Québec H3C 3P8, Canada
[2]Sony Computer Science Laboratory
6 rue Amyot, 75005 Paris, France
e-mail : hafedh.mili@uqam.ca, pachet@csl.sony.fr

Abstract

Metamodeling refers to the practice of representing objects at more instantiation levels than the usual two, instance and class. The need for representing computational entities in general, and objects in particular, at several levels of instantiation arises in many occasions. When properly recognized as a case of metamodeling, it may lead to a better understanding of the underlying application logic, and to a more reusable and efficient implementation. In this paper, we look at manifestations of metamodeling through the study of three classes of problems and describe design patterns that support the implementation of the most common instances of metamodeling. We conclude by highlighting directions for further research.

Keywords: ***Metamodeling, design patterns, metaclasses, domain models, process models.***

1. Introduction

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 has to be queried or otherwise manipulated need to explicitly represent the *representation* of objects [Diaz & Patton,1994] ; the act of representing the representation of objects is called *object metamodeling*. More generally, we use the term *metamodeling* to refer to the practice of representing computational entities at more instantiation levels than the usual two, instance and class. By computational entity we mean pure data, objects in the OO sense, tasks, processes, or any computational artifact that can be created, modified or otherwise manipulated during the execution of a program.

The need for metamodels is more frequent than one might first think, and has been practiced for some time. In relational database modeling, meta-data consists of table descriptors, which are system tables describing the data tables (their columns, domains for the columns, etc.) and integrity constraints, which are, by and large, semi-declarative constructs to be executed during updates. A typical computer-assisted manufacturing application needs two levels of instantiation/abstraction: one level to represent bills of material, e.g. describing the *composition* of different manufactured products, and a second level to describe *actual* manufactured products with stock numbers, locations in warehouses, etc. ; bills of materials are *representations* or *models* 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, i.e. *metaclasses.*

When not part of the problem (e.g. bills of material *and* inventory management in the same application), metamodeling is part of the solution: metamodeling is an *abstraction* mechanism in the sense that, much like classification, it enables us to factor out the common parts among a set of individual computational objects, and package their differences in a way that supports reuse, evolution, and interoperability [Mili et al., 1995]. Also, much like classification, it enables us to replace *extensions* (i.e. explicit occurrences of a concept) by *intensions* (the definition of the concept itself), and hence result into an economy of representation. Finally, when applied in actual software projects, metamodeling and

metaprogramming (programs that generate programs) 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.

Recognizing instances of metamodeling, and distinguishing them from generalization, is not easy, because generalization may, in simple cases, be used to implement metamodels. Further, implementing metamodels is difficult, both conceptually, and because of lack of support in existing languages. For instance, not only are metaclasses insufficient, but not all languages support metaclasses, and some that do (e.g. Smalltalk) offer a restricted implementation of metaclass programming.

In this paper, we first look at manifestations of metamodeling through the study of three classes of problems. In section 3, we describe design patterns that support the implementation of the most common instances of metamodeling. We conclude in section 4 by highlighting directions for further research.

2. Recognizing metamodeling

The need for metamodeling arises when we need to represent information at different *realms* of abstraction. Those « realms » could be different levels within the same domain of representation/abstraction, or could belong to different domains. We have classified occurrences of metamodeling into three groups, depending on the nature of the « realms »and their relationships.

2.1 Abstracting domain knowledge

Consider the example of a Savings & Loans Association whose member institutions offer different kinds of loans with various payment schedules. This is a simplified version of an actual modeling problem, for an actual such institution, on which one of the authors has worked. Generally speaking, different types of loans have different payment formulas. The association between types of loans and payment formulas is practically industry-wide. For example, all student loans have an initial deferment period, after which regular payments must be made at a minimum pace, but a student may pay the remaining balance in full at any point in time. It is also almost always the case that mortgages are to be paid back according to some regular payments, and full payments usually carry a penalty because lending institutions usually turn around and sell mortgages to potential investors with a guaranteed rate of annual return. For each one of these modes of payment, there are a number of parameters such as frequency of payment, ranges of allowable percentages, etc. Different member institutions of an S&L Association may offer different subsets of these ranges. For example, institution A will offer 6 month, 1, 2 and 3 year fixed-rate mortgages, while institution B will offer 1, 2, 3, and 5 year fixed-rate mortgages. An individual who walks into a particular institution (e.g. A) will get *a* mortgage with a *single value* from these parameters, e.g. a 3 year fixed-rate mortgage.

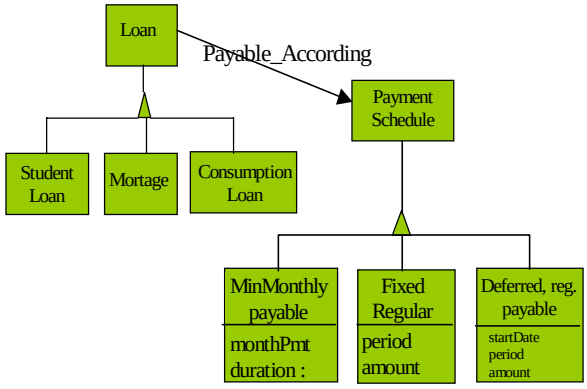

Figure 1. A simplistic model of loans and payment schedules.

A first-cut object model for this problem looks like Figure 1. This model says that « a loan » is payable according to « a payment schedule », and that there are many types of loans, and many types of payment schedules. What it fails to say is which schedules of payment are appropriate for which types of loans, *in general*, and *for individual institutions.* If we had one payment schedule per loan type, we could

represent this easily within this model by specializing the association « Payable According » into three (or more) more specific associations. However, the combinations can be numerous, and when we take into account the specifics of individual institutions, they become unwieldy.

What is really at issue here is the fact that we are attempting to represent information about two aspects, 1) about account *types*, and 2) about specific accounts of these types ; the latter have to abide by the constraints and parameters of the former. The object model of Figure 1 shows « a model of» the associations between *individual* accounts and *individual* payment schedules but very little about account types in general: if we restrict ourselves to a single institution, it may show which payment schedules apply to which accounts *for this particular institution*. We would need *a separate model for each institution*. The way of handling this consists of representing information explicitly about types of loans, types of payment schedules, and institutions, explicitly. Figure 2 shows such a model.

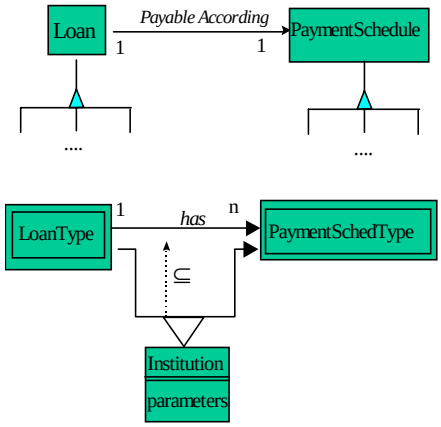

Figure 2. A complete model including a metamodel.

The upper part of the model is equivalent to the model of Figure 1. The lower part *is* the metamodel. The classes **LoanType** and **PaymentScheduleType** represent *types* or *classes* of loans and payment schedules, respectively, i.e. classes whose instances are themselves classes. The fact that a particular institution uses a specific subset of the set of payment schedules practiced by the industry at large is represented by the subset relationship (⊆ in Figure 2) between the association that is specific to individual institution, and the general one. The knowledge that would have been embodied in separate models, one per institution, is now represented by a single (meta)model (lower part of Figure 2), and some specific instances of that model (e.g. a table). The *explicit* presence of this information is valuable for the purposes of coding the desired behavior concisely and generically.

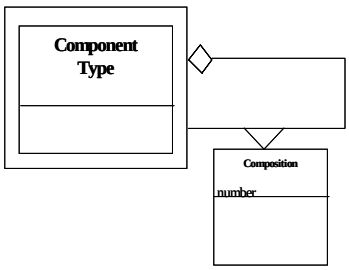

Figure 3. A (meta)model of bills of material.

A common instance of this general problem is the bill of materials problem in a manufacturing organization. For each non-elementary product, the organization would have a *recipe* for the product indicating its composition. For example, a **Table** is made of a **Board** and four **Leg**s. A **Chair** is made of two **Board**s, four **Leg**s, and two **Armrest**s. These recipes need to be stored and manipulated in the same way that we need information about LEG123402 and CHAIR2364315, in which warehouse they are stored, and whether LEG123402 is a component of CHAIR2364315. In this case, the recipes are

represented by *a* metamodel, and they are themselves models of the data (database). The metamodel is shown in Figure 3.

2.2 Representing the modeling language

The need for representing modeling languages arises under many circumstances. We identify two general categories of uses, (1) defining modeling notations and translating between them, and (2) abstracting processing tasks. While the first case is important for model sharing/exchange, and is a necessary part of any reuse infrastructure, we will highlight the major issues, and focus on the aspects related to design.

2.2.1 **Describing and translating between modeling notations**. Development methodologies include modeling notations that allow us to represent « worlds » using the constructs of a modeling language. Different methodologies use different notations, and it is important to understand the correspondences between these notations to allow the people and the tools that speak these modeling languages to exchange models and understand each other.

There are two strategies for addressing this problem. The first is to define a modeling language that is a join, of sorts, of the various modeling languages. A particular notation can be thus obtained as a subset and/or a specialization. A subset in the sense that only some constructs are used. A specialization in the sense that a specific notation ascribes additional semantics to a construct or imposes additional constraints to its use. In practice, this may lead to problems, as people who participate in standards committees can attest: the hardest cases occur when different notations use close enough constructs that they would be redundant, but not close enough that they are identical or that one subsumes the other.

The second approach seeks a *notation for describing notations*, or, more broadly, an ontology of modeling notations. Fewer constructs are needed to describe notations then there are constructs in the notations themselves. Figure 4 shows a *simplistic model* of a part of OMT ... in OMT. Generally speaking, the more atomic the constructs of the meta language, the fewer constructs we need. However, the descriptions tend to be more complex ; the challenge in choosing such a meta language is to find a reasonable compromise. It is interesting to note that UML combines both approaches described above. UML does have a set of constructs that is meant to handle 80% of the cases (whether that goal is attained or not is another matter). For the remaining 20%, a *metamodel* is provided to support the addition of new constructs, as needed.

Once the problem of defining notations has been resolved, comes the issue of translating between notations. If the translation is to be systematized, translation rules need to be expressed in terms of constructs of the source language, and constructs of the target language. One simple rule might say :

$$\mathbf{CLASS_{OMT}} \rightarrow \mathbf{TYPE_{ODELL}}$$

More complex rules may be written that transform a configuration of constructs from the source language to a configuration of constructs in the target language. For example :

$$\mathbf{TABLE}_{REL}(A) \wedge \mathbf{TABLE}_{REL}(B) \wedge \mathbf{TABLE}_{REL}(\{key(A), key(B)\}) \rightarrow$$
$$CLASS_{OMT}(A) \wedge CLASS_{OMT}(B) \wedge ASSOC\_M\_To\_M_{OMT}(A,B)$$

This rule translates a fragment of a relational conceptual model to an OMT model. A number of systems have been developed to migrate relational database technology towards object-oriented databases [Blaha et al., 1994], [Revault & Sahraoui,1995].

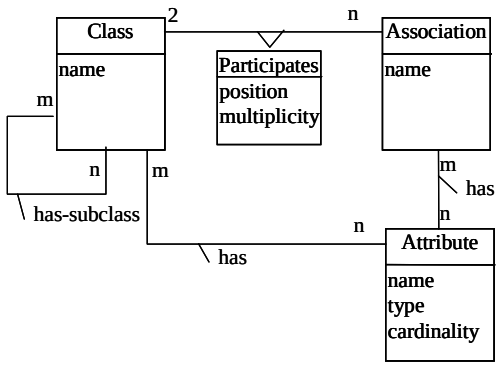

Figure 4. A metamodel of OMT in OMT.

For systems that need to contend with several modeling languages, an alternative approach to this language-to-language translation might use a common "esperanto" to which models are translated back and forth from their source language; the translation rules are written in terms of a common meta language that helps describe both the "esperanto" and all the languages (see e.g. [Missaoui et al., 1998]). This approach is more «economical» since it reduces the number of required translations from n^2 to 2n, but suffers from the limitations of the esperanto approach.

2.2.2 **Process abstraction for algorithm reuse**. Broadly speaking, class behavior that is dependent on the definitional structure of the class may be decomposed into a structure access part, and a computation part (see e.g. [Mili & Li, 1993]). The idea of collection class iterators may be seen as a simple example of this, where we separate the iteration part from the actual computation to be performed on the individual elements of the collection. The following C++ sample code illustrates the idea:

```
T Collection<T>::average() {
   CollectionIterator<T> it(this);
   T average;
   int n = 0 ;
   Node<T> current;
   while (! current = it.next()) {
         n = n + 1 ;
         average += current.value();}
   return average/n ;
}
```

In this case, the same function `average()` may be used on all the subclasses of Collection, independently of how they are structured, provided that each subclass provides an iterator. We could go one step further, if the structure of the class were available during run-time: even the iteration part may be written once for all. This approach may be used to define e.g. a generic shallow copy operation, a generic comparison operation, or a generic persistence operation. The following code excerpts show a generic shallow copy operation:

```
RootObject* RootObject::shallowCopy(){
   MyRootObject* cp = OneJustLikeMe();
      DataMemberIterator it(this);
      MemberNameType name;
      while ( ! name = it.next())
         cp->setValueOf(name, this->getValueOf(name));
      return cp;
}
```

We assume that `setValueOf(MemberNameType, void*)` and `getValueOf(MemberNameType)` will do the "right thing", and that users of `getValueOf(..)` will do the proper typecasting with the returned value, and so forth. Note the class `DataMemberIterator`, which knows how to access the list of data members of the current object.

In addition to reducing the amount of code that needs to be written, this approach may be almost mandatory in applications where new classes are added to the system during run-time. In non-typed languages such as Smalltalk and CLOS, new classes may be added, queried, instantiated, and manipulated during run-time, without any problem. With Java, new classes that satisfy a pre-compiled interface may be linked during run-time, but we cannot extend the interface during run-time. With C++, neither is possible, and a metamodeling approach is required. We will show in section 4 design patterns for implementing this for the case of C++ and like languages.

2.3 Behavioral metamodeling

Under behavioral metamodeling, we group descriptions of objects as computational entities, or *processes*, that may have a location (memory space), resources scheduled for it, priorities, etc. Behavioral metamodeling may manifest itself to varying extents depending on the variability in the computational properties of the objects, and the desired flexibility of the run-time environment. Generally speaking, the execution behavior of an application may be either embodied in the code of the application itself, or

stated declaratively for some run-time engine to interpret and enact. The trend has been to extract execution behavior from application code to run-time environment code. Nowadays, the transparent execution of distributed applications relies heavily on the explicit representation of run-time information, ranging from distribution information (e.g. the location of database tables) to information usable by ORBs (interfaces and locations of *individual* objects). Figure 5 illustrates this point.

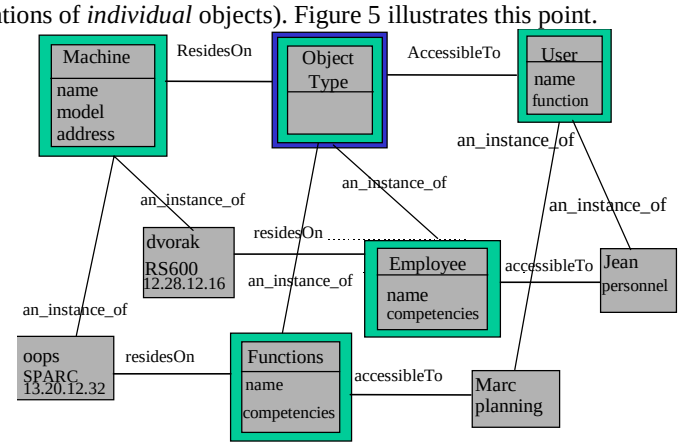

Figure 5. Representing location and access information explicitly.

A yet more advanced use of behavioral metamodeling is the explicit representation and execution of execution models. This is particularly valuable in distributed concurrent applications where different mechanisms for message sending, message handling, synchronization, resource allocation strategies, etc. (see e.g. [Briot,1996]) may be used or tested. Partially or fully reflective programming languages like Smalltalk make it possible to describe those mechanisms in the language itself (see e.g. Actalk [Briot,1989], and [Kuwabara et al.,1995]).

This approach has several advantages. By separating the execution model from the application logic, it considerably reduces the size of applications. Further, it yields reusable *executable models* which may be combined with other applications. It also yields far more reusable domain objects, which may then be used under different circumstances (single user, multi-user, multi-user dis-tributed, multi-user, multi-processor, client-server, etc.).

3. Designing metamodeling

3.1 Metamodeling = metaclasses ?

The origin of metaclass programming may be found in the ObjVlisp model [Cointe, 1987], which proposed a unified model for representing objects, classes and metaclasses. In this model, a user may define classes by freely specifying their superclass and metaclass, with no limitation. However, existing object-oriented languages are nowhere close to providing such a general scheme. Most typed languages do not provide any representation of classes at run time (C++, Eiffel). Java recently introduced a MOP in which ad hoc representations of classes are introduced, but which give limited user control. Among untyped languages, CLOS offers, in principle, unlimited access and control, but this lack of restriction makes it difficult to use in practice (see e.g. [Danforth & Forman, 1994]).

Regardless of whether languages support metaclasses, fully and safely, it is not clear that metaclasses are always needed to implement metamodeling, nor is it clear that they are always sufficient. In section 4.2, we will show alternative ways of implementing metamodels, including simulating metaclasses in languages that do not support them. In this section we address the necessity and sufficiency of metaclasses.

First, the issue of metaclass necessity. The representation of classes as objets is only useful if:

1. There is some useful behavior of class objects that can be invoked during run-time, either directly, or through instances of the class objects, and
2. The coding of that behavior benefits from class packaging, i.e. shareability between a bunch of similar (class) objects, encapsulation, and inheritance.

If class objects are only used as repositories for instance properties that apply to *all the objects of that class* then, clearly, the representation of classes as objects is *not* required.

However, there is more to metaclasses than the representation of class objects: *it also implies a «live»* (during run-time, stored or computed) *is-a link between objects and their classes*, i.e. *at least two levels of instantiation during run-time*. If the only use of that link is to to query an object about its class, that link can be implemented like any association (see e.g. section 3.2). If, on the other hand, that link is useful for general run-time behavior, then it is important that that link be built-in and inviolable ; this is the case for Smalltalk where the is-a link is used for run-time typing in general, and message handling in particular.

With regard to sufficiency, metaclasses are clearly not sufficient when it comes to the metamodeling of execution behavior: the metamodeling of execution behavior requires explicit representation of entities such as ***message***, ***parameter***, and ***context***, more than it needs the representation of classes as objects—albeit they constitute together the reflexive apparatus of reflexive languages [Briot,1996].

3.2 Structural metamodeling design patterns

3.2.1 **Representing two levels of instantiation**

Figure 7 shows the object model of a design pattern we used many times to represent class information explicitly during run-time. Implementation-wise, instances of **ObjectType** have two dictionaries, one to hold descriptions of attributes (accessible by attribute name, or 'attName'), i.e. instances of class **Property**, and one to hold components, i.e. other instances of the class **ObjectType**, accessible by component role name (e.g. 'leftLeg' and 'rightLeg', both of which could be instances of **Leg**). An actual object will be represented by an instance of the class **Object** or one of its subclasses. Each object (instance of **Object**) will also have two dictionaries, one to hold attribute values (instances of **PropertyValue**, accessible by 'attName'), and one to hold components (instances of **Object** or one of its subclasses, accessible by 'roleName'). Objects point to the instance of **ObjectType** that describes them. The class **Object** (and its subclasses) will implement a constructor that takes an instance of **ObjectType** as an argument, from which to initialize the dictionaries. If the classes of objects in the application have no distinguishing behavior except for the component and attribute access methods, all the classes can be represented by a single class, **Object**.

This pattern has proven useful for cases where we have to deal with several classes with complex data structures but whose behavior is no, or little, more than structure access: all the access methods can be coded generically as table access methods. It supports the addition of new attributes to existing classes during time, and the addition of new classes altogether. One application in which this pattern was used was a CAD application that manipulated 3-D objects drawn by the user with a drawing editor. The second application dealt with the processing (analysis and routing) of messages in an avionics message processing system for a US Airline, where incoming messages have unpredictable structure and contents.

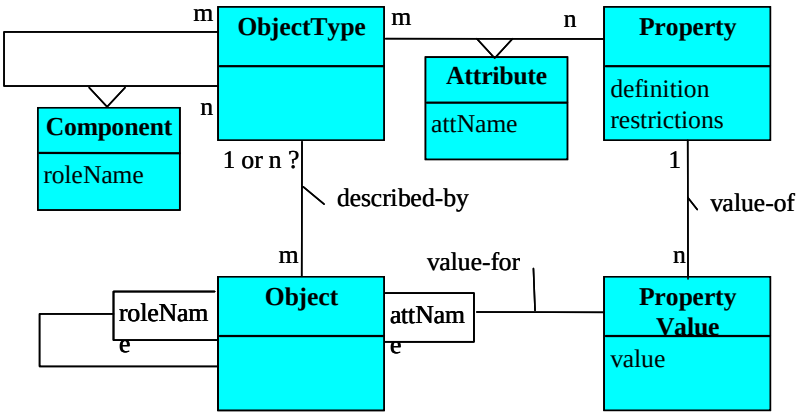

Figure 7. Simulating metaclasses in a classless language

An interesting extension of this model is the fact that an *object can have several descriptions*. We mentioned earlier the problem of multiple instantiation (see section 3.2). In essence, by decoupling

instances from the language built-in links to their « types » (since we use artificial types) we are able to implement multiple instantiation. Allowing an object to have multiple descriptors simply means that the constructor will build dictionaries that consist of the unions of the dictionaries originating from the individual descriptors—with the usual conflicts !

3.2.2 **The Zig-Zag pattern : implementing multi-level metaclasses**

This pattern comes from experiment with the MétaGen system [Revault&Sahraoui,1995]. MétaGen is a Smalltalk-based CASE tool that generates tools that transform models in a source description language (e.g. analysis model) to models in a target description language (e.g. design or implementation model). MétaGen takes as input a description of a source language, a description of a target language, and set of rules for transforming source language constructs to target language constructs. It outputs a graphical editor for the source language, a transformation procedure, and a graphical editor for the target language [Revault&Sahraoui,1995].

Smalltalk supports only one level of metaclasses. MetaGen requires several levels of instantiation so cannot be accommodated by Smalltalk. The basic idea of the ZIG-ZAG pattern (Figure 8) is to use a manual link to represent classes by regular instance-objects, which allow us in turn to go up an additional instantiation level. The representation of a class object by an instance is represented by an association "describes'' whereby class objects point to instance objects through a *metaclass instance variable* called "describedBy", and instance objects point back through an *instance variable* called "describes".

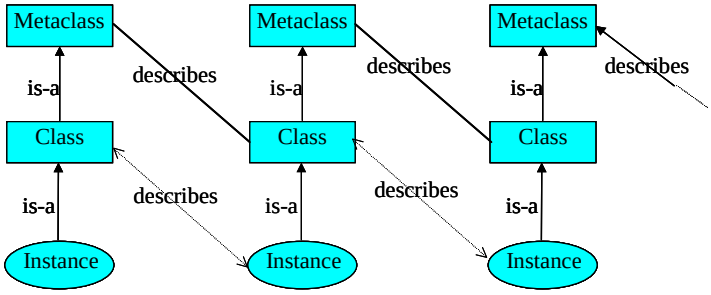

Figure 8. The ZIG-ZAG pattern.

3.3 Patterns for behavioral metamodeling

Behavioral metamodeling is mostly based on an explicit representation of message handling. In reflexive languages such as Smalltalk, message handling is already represented by Smalltalk objects, and the building blocks of behavioral modeling are there to build more complex message handling procedures either by adding new constructs or by specializing existing constructs [Briot,1996]. Depending on the desired flexibility with the computational models, designs can be fairly complex. For the purposes of this paper, we show a simple pattern allowing the interception and scheduling of method executions. This pattern is a simplification of the concurrency framework discussed in [Briot&Guerraoui,1997]. Figure 9 shows an object model for this pattern.

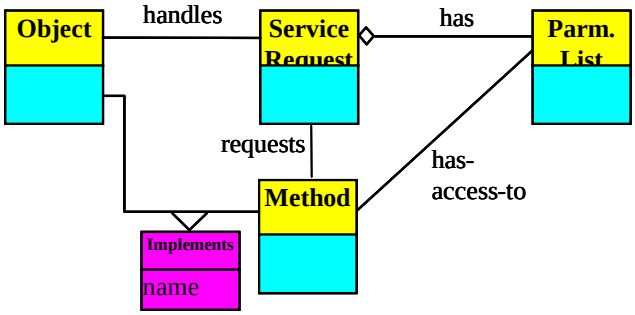

Figure 9. A simple message interception design pattern.

Implementing this pattern in Smalltalk is easy, since methods, parameters, and contexts are all Smalltalk classes, and message handling is done generically through a "perform:'' request which asks the object to perform a service request, as in Figure 9. Further, methods are represented by classes, and the "implements'' relationship shown in the model of Figure 9 is actually represented at the class level under the form of a *method dictionary*.

To implement this pattern in a language such as C++, some effort has to be spent. First, we would define classes the way we do normally, with data and member functions, and then add the following:

1. A dictionary of **Method** objects—call it *_methodTable*—indexed by name, which know how to call themselves on the object, and get the right parameter values from an instance of **ParameterList**,
2. A member function that handles service requests generically, by getting the proper **Method** object to do it.

The class **ServiceRequest** will have two data members, a method *name*, and an instance of **ParameterList** which could be either an actual heterogeneous list, or a record structure with properly typed fields. The class **Object** above would support the following method:

```
void* Object::handleRequest (ServiceRequest* sr, ParameterList* pl)
{
   Method* meth = _methodTable->at(sr->getName()) ;
   return meth->executeOnWith(this,pl);
}
```

The idea is that a particular instance of **Method** knows how to access the parameter list and invoke the actual method:

```
void* Method::executeOnWith(Object* op, ParameterList* pl) {
   T1 a1 = (T1)(pl->getElement(1));
   T2 a2 = (T2) (pl->getElement(2)) ;
   ...
#ifdef _SMART_IMPLEMENTATION
   return op->(*_method)(a1,a2,...) ;
#else
   return op->computeValue(a1,a2,...)
#endif
}
```

In the smart implementation, the class **Method** has a data member (_method) which is a function pointer whose type is the signature of the actual method of **Object** that will be called. In the other implementation, the name of the actual method is hardcoded, and we have to create or otherwise generate a subclass of **Method** for each method of each class (subclass of **Object**). The savvy C++ programmer can find a number of enhancements to this general scheme, including automating the generation of some of the code through the use of macros.

This scheme enables us to do lots of things with messages, starting with tracing, whereby the additional level of indirection can log the receiver, the parameter list, the caller (if we add an argument to handleRequest()), and the result. It may also be extended to assign processes to the various requests, and manage the concurrency of the tasks using the available process management environment.

Interestingly, application distribution infrastructure use similar schemes. For example, Java has a reflection package which is used within Java RMI, and is the basis for the Beans and EnterpriseJavaBeans architectures, which allow programs in a client-server architecture to interact with minimal knowledge of each other's interfaces. CORBA makes use of such patterns as well. CORBA was meant to address problems related to running "heterogeneous", and distributed applications. First, the standard had to abstract away issues related to the execution environment (location, platform, process management). Second, it had to abstract away issues related to the specific packaging of functionality, both in terms of paradigm—hence the paradigm-neutral notions of "module" and "interface"—and in terms of syntax—hence the interface registry idea, and the dynamic invocation interface.

4. Conclusion

Abstraction and separation of concerns are key strategies for managing the complexity of systems. Domain engineering is concerned with building a set of software constructs that can accommodate a set of application within an application domain, with different functional needs, different configurations, and different operational requirements. Abstraction comes in two flavors, which we might call *omission*, and

factorisation/*parametrization*. Abstraction by omission ignores the details that might differentiate between two instances of the abstraction; by doing so, using the abstraction requires putting that information back in, each time, i.e. costs engineering effort for each use. Abstraction through parametrization embodies variations in *concrete* interchangeable subcomponents. We showed in this paper a set of techniques, which we collectively call *metamodeling*, which lead to more reusable components, and more efficient implementations. We sketched the common theoretical foundations of these techniques, illustrated by a set of proven design solutions.

Acknowledgements: This work benefited from interactions within the OOPSLA'95 workshop on metamodeling (see http://www.info.uqam.ca/Labo_Recherche/Larc/metamodeling-wshop.html), and follow-up discussions with Jim Odell and Jean Bézivin, both privately, and moderated through the OA&DTF mailing list.

5. References

[Atzeni,1993] P. Atzeni and R. Torlone, ``A Meta Model Approach for the Management of Multiple Models and the Translation of Schemes, '' *Information Systems*, vol. 18, n 6, Pergamon Press Ltd., p. 349-362, 1993.

[Bezivin,1995] J. Bézivin, ``Technologie objet et ingénierie des besoins : une réconciliation nécessaire,'' *L'Objet*, vol. 1, n 1, p. 21-26, 1995.

[Blaha et al., 1994] M. Blaha, W. Premerlani, & H. Shen, "Converting OO Models into RDBMS Schema," *IEEE Software*, vol 11(3), May 1994, pp. 28-39

[Briot,1989] J.P. Briot, ``Actalk, a testbed for classifying and designing actor languages in the Smalltalk-80 environment'' in proceedings of the *European Conference on Object-Oriented Programming* (ECOOP'89), *Lecture Notes in Computer Science* (LNCS), Springer Verlag, 1989.

[Briot & Cointe,1989] J.P. Briot & P. Cointe, "Programming with Explicit Metaclasses in Smalltalk-80", *OOPSLA'89*, 419-432.

[Briot,1996] J.P. Briot and P. Cointe, "An experiment in Classification and Specialization of Synchronization Schemes'' in *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software* (ISOTAS'96), LNCS no 1049, 96, pp. 227-249.

[Cointe,1987] P. Cointe, "The ObjVLisp Kernel: A Reflexive Lisp Architecture to Define a Uniform Object-Oriented System'' in *Meta-Level Architectures and Réflexion*, edited by P. Maes and D. Nardi, pages 155-176, North-Holland, Amsterdam, 1987.

[Danforth&Forman,1994] S. Danforth and I. R. Forman, "Reflections on Metaclass Programming in SOM" in *proceedings of OOPSLA'94*, Portland, pp. 440-452.

[Diaz & Patton, 1994] O. Diaz and N. W. Paton, "Extending ODBMSs Using Metaclasses," *IEEE Software*, vol. 11(3), May 1994, pp. 40-47

[Graubé,1989] N. Graubé, "Metaclass compatibility", *OOPSLA'89*, New Orleans, pp. 305-316

[Kuwabara et al.,1995] K. Kuwabara, T. Ishida and N. Osato, "Agentalk : Coordination Protocol Description for Multiagent Systems" in *ICMAS'95*, 1995, pp. 455-461.

[Mili et Li, 1993] H. Mili and H. Li, "Data Abstraction in SoftClass, an OO CASE Tool for Software Reuse" in *Proceedings of TOOLS USA '93*, Santa Barbara, CA, Aug. 2-5, 1993, Prentice-Hall, Ed. Bertrand Meyer, pp. 133-149.

[Mili et al., 1995a] H. Mili, F. Mili, and A. Mili, "Reusing Software : Issues and Research Directions" *IEEE Transactions on Software Engineering*, June 1995, pp.

[Mili et al., 1995b] H. Mili, F. Pachet, I. Benyahya, and F. Eddy, "Report on the OOPSLA'95 Workshop on Metamodeling" *Addendum to the OOPSLA'95 proceedings*, ACM SIGPLAN notices.

[Missaoui et al.,1998] R. Missaoui, H. Sahraoui, and R. Godin, "Migrating to an Object-Oriented Database Using Semantic Clustering and Transformation Rules" to appear in *Knowledge and Data Engineering*, 1998.

[Revault & Sahraoui, 1995] N. Revault and H. Sahraoui, "A Metamodeling Technique: The MétaGen System" in *Proceedings of TOOLS Europe '95*, Versailles, France, 1995.

[Rivard, 1996] F. Rivard. Smalltalk: a Reflective Language. In REFLECTION'96, pages 21--38, San Franscico, USA, April 21-23 1996. Ed G. Kiczales, http://www.emn.fr/ deptinfo/rivard/ perso/informatique/reflection96/reflection96.htm