Metalevel Programming in CLOS

Giuseppe Attardi, Cinzia Bonini, Maria Rosaria Boscotrecase, Tito Flagella, Mauro Gaspari

Artificial Intelligence Division, DELPHI SpA, via Vetraia 11, I-55049 Viareggio, Italy

Abstract

The facilities of the Metaobject Protocol are explored to incorporate various paradigms of object-oriented programming within the Common Lisp Object System. The basic principles of metalevel programming are sketched and the technique is applied to develop atomic objects, persistent objects and graphic objects for CLOS.

1. Introduction

The COMMON LISP Object System (CLOS) is the new standard object-oriented extension for COMMON LISP , derived from the experiences with two well known object languages integrated with Lisp: LOOPS [Bobrow 83] and Flavors [Moon 86].

One of the most innovative ideas of CLOS is the provision of a metalevel facility, which allows to extend the object system with new specific features or functionalities, while still remaining within a common framework. The advantage of this approach is that different varieties of object systems can coexist and be profitably combined.

Smalltalk [Goldberg 86] and LOOPS had explored the idea of metaclasses but the lack of a uniform object model [Cointe 87] and of a visible metalevel mechanism has limited the potential power of the tecnique.

In an object system the behavior of an object is determined by its class, and that of a class by its metaclass. CLOS has adopted a uniform model, where each object in the language is an instance of a class and where classes and metaclasses are first class elements [Cointe 87]. Therefore the basic elements of CLOS itself are represented as instances of CLOS classes, which are dealt in the same way as those objects that a programmer is expected to manipulate. A programmer then can build its own system objects, to use instead of the existing ones. However, by exploiting a fundamental facility of object-oriented programming, i.e. specialization of generic functions, one can make the new system objects to coexist with the standard ones. Since specialization preserves the functional interface to the objects, instances derived from the new and the standard

classes may be freely mixed in programs.

This technique provides a novel approach to language extensibility.

The classical approaches to extend a programming language like LISP are interpreter layering and macro expansion. Recently some attention has been paid to a third technique: reification and reflection [de Rivieres 88]. Reification allows to expose some of the internal data structures of the interpreter, so that programs can manipulate them. Reflection is the process by which these updated structures are re-installed in the interpreter to affect its subsequent behavior. In a language providing reification and reflection, new constructs can then be defined by programs which access and update the run time state of the language.

We claim that a metalevel implementation of an object system is the dual of a reflective interpreter: a reflective interpreter provides a way to extend the control facilities of a language, similarly a metaobject system provides a way to extend its data structuring facilities.

2. CLOS

CLOS [CLOS 88] is based on the concepts of class, metaclass, multiple inheritance, generic function, method and method combination.

A basic building block of CLOS-based programs is the *class*. A class is an object that determines the structure and the behaviour of a set of other objects, which are called its *instances*. In CLOS all objects are instances of a class. A class can inherit structure and behaviour from one or more other classes. The conflicts deriving from *multiple inheritance* are resolved by the class precedence list associated to each class, which is a total ordering on the set of the given class and its superclasses.

A class is an instance of another class, called *metaclass*, which controls its behaviour as an object. The metaclass determines the representation of the instances of that class. CLOS provides a set of predefined metaclasses (CLOS kernel) that will be described in the following section. A user can define new metaclasses.

Generic functions are an extension of ordinary Lisp functions which are able to perform operations in ways which depends on the type of its arguments. The *methods* associated with the generic function define the class specific operations of the generic function. The behaviour of a generic function depends on which methods are selected for execution, on the order in which the selected methods are called and how their values are combined to produce the value of the generic function. This mechanism is called *method combination*.

There is a mapping from the class system defined by CLOS into the COMMON LISP type

space. Many of the standard COMMON LISP types have a corresponding class with the same name as the type. The hierarchical relationships among the Common Lisp type specifiers are reflected by relationships among the classes corresponding to those types.

The COMMON LISP structures are integrated in the class graph. In fact each structure type created by `defstruct` without using the `:type` option has a corresponding class.

3. The Object Model

The basic object model adopted in CLOS assumes the existence of two relations between objects: the *inheritance* relation and the *instance-of* relation. The instance-of relation plays a role in the way new objects are created, while the inheritance relation describes how an object will inherit its structure or which methods are applicable to it.

Since CLOS is integrated with COMMON LISP , all entities belong to some class. CLOS is a uniform system in the sense that each object in the system is an instance of some class. A class is an instance of another class, its metaclass. The only difference between a class and an instance is that a class can create instances of itself, while an instance can't. As a consequence there is no type distinction among classes, metaclasses and instances.

In the following we present the CLOS predefined classes that are relevant to our discussion, discarding all the others. There are two metaclasses that encompass the primitive data types of COMMON LISP (built-in and structure types). The metaclass *standard-class* is the class from which all other classes originate and is therefore the root of the instantiation tree. The class `T` is the root of the inheritance lattice and corresponds to the CL type specifier `T`. The class *class* describes the basic behaviour of all classes and it is not intended to be instantiated.

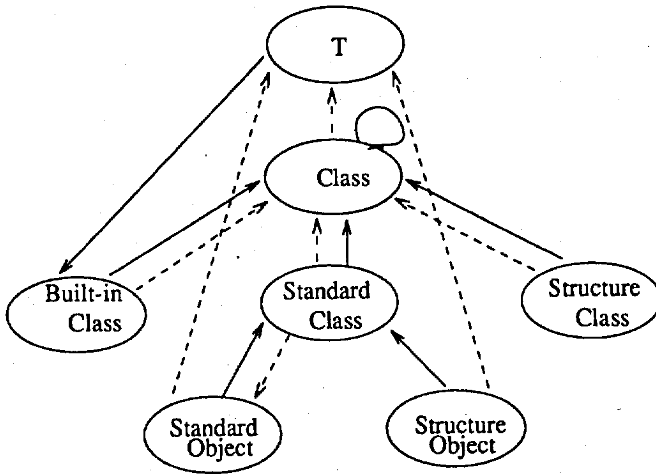
standard-class and *standard-object* represent the default kind of objects which are generated in CLOS, in particular by the constructs in the *macro interface*. Similarly, the classes *structure-object* and *structure-class* represent the objects created by `defstruct`.

Figure 1 shows the relationship among the classes in the CLOS kernel according to our implementation [Attardi 88c]: solid arrows represent instance-of relations, dashed arrows represent inheritance. Here *class* is designed as the minimum structure capable of describing its own structure and common to any class [Cointe 87]. Therefore it is the class of all the classes in the kernel except `T`.

4. Metaobject Protocol

CLOS consists of two levels: the *programmer interface* and the *metaobject protocol.*

FIGURE 1



The programmer interface provides the primitives for creating and manipulating objects. The programmer interface is split into two parts: a *functional interface* and a *macro interface*. The *functional interface* consists of a collection of functions implementing the basic object operations: since everything in CLOS is an object, including classes and methods, this is all is needed to build an object system. The *macro interface* consists in a set of macros which provide a convenient syntax for defining classes and methods, which is all most users need to know to write their programs.

The second level (CLOS Metaobject Protocol) describes CLOS itself as an object oriented program. It specifies the classes, methods and generic functions which implement CLOS. The kernel classes describe the data structure used to represent the CLOS program elements (metaobjects), the generic functions and methods define the basic operations of the kernel (metaobject protocol or MOP).

5. Introducing Object-Oriented Paradigms

In CLOS the behaviour of the object system is specified in terms of a set of generic functions for which methods are predefined for the classes in the kernel. By defining new classes and methods for them, a programmer can build extensions to the basic CLOS framework.

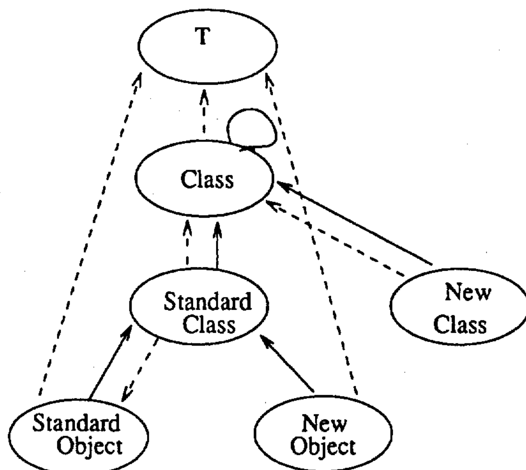
Each variety of object system, including CLOS itself, can be defined as deriving from a

pair of classes: an object class *NewObject* and a metaclass *NewClass*. The object class *NewObject* is the root of the inheritance tree for the particular object system and so it describes the default behaviour of the objects of such system. The metaclass *NewClass* is the root of the instantiation tree for that object system and so it describes the structure and the behaviour of the classes in such system. In particular for the kernel of CLOS itself, this pair of classes consists of *standard-object* and *standard-class*.

A new object-oriented paradigm can be introduced according to the model shown in figure 2.

Whenever one wants to define a new kind of object system, one defines such a pair of classes and the appropriate specializations of the generic functions (methods) for those classes, which implement the required behavior. Being *class* the root of the instantiation tree, which describes only the basic behaviour of all classes, these methods could be defined completely from scratch. Therefore the behavior of the new object system can be completely different from the default CLOS behavior embodied in *standard-object* and *standard-class*. However in many cases one only needs limited variations to this behavior, so the pair of classes describing the object system can be related to the standard ones. The importance of the metaobject protocol is to make the implementation of the standard object system open and visible to programmers. Since the standard system is implemented in terms of *generic functions*, these functions can be specialized where necessary while the standard versions can be relied upon in the other cases.

FIGURE 2



We exploited metaobject programming techniques in CLOS to build a number of facilities:

(1) synchronization facilities based on atomic objects;

(2) provision of persistent objects which are stored in an entity-relationship database;

(3) support for graphic objects.

As a further example of the possibility of integrating various object system within the unified CLOS framework, the Knowledge Representation System (KRS) of Luc Steels has been built in CLOS [Simi 88].

In the following sections we sketch how these constructions were performed.

5.1. Atomic Objects

In order to deal with concurrency in CLOS, we defined a metaclass, named *atomic-class*, and a superclass, named *atomic-object*. Instances of each class which is an instance of *atomic-class* and which inherits from *atomic-object* will have the behaviour of atomic objects.

In the following code a brief explanation of the CLOS syntax is provided in comments.

```
(defclass atomic-class              ; define a new class called atomic-class
  ()                                ; inheriting from no other class, except standard-object
  ()                                ; having no slot
  (:metaclass class))               ; and being an instance of class class

(defclass atomic-object             ; define a new class called atomic-object
  ()                                ; inheriting from no other class, except standard-object
                                    ; having the following slots:
  ((lock                            ; lock: set to the thread owning the lock on the object
       :initform nil                ; initialized to nil
       :accessor lock)              ; and accessible with a method called lock
   (request-queue                   ; request-queue: queue of threads waiting to access
       :initform nil                ; the object, initialized to nil
       :accessor requests)))        ; and accessible with a method called requests
```

Atomic objects are meant to support synchronization by mutual exclusion between conflicting operations on the same object. In general, operations which update the content of a slot in the same atomic object must be synchronized. In CLOS, the primitive for accessing the slot of an instance is `slot-value`, and the same primitive, in conjunction with the operator `setf`, is used to update it.

In the following example, we access and modify the content of the slot `balance` for

the object `my-account`, created as an instance of the class `account`.

```
(defclass account (atomic-object)
      ((balance :initform 0))
      (:metaclass atomic-class))

(setq my-account (make-instance 'account))          ; create an account

(slot-value my-account 'balance)                    ; access slot balance

(setf (slot-value my-account 'balance) 500)         ; update the value of the slot
```

In CLOS, all accesses to slots go through the function `slot-value`, and updates through the function `(setf slot-value)`. These functions are implemented using the generic functions `slot-value-using-class` and `(setf slot-value-using-class)` respectively. By specializing these generic functions with suitable methods for a specific metaclass, it is possible to change the basic behavior of objects.

For instance, to ensure that an operation requiring exclusive access to an atomic object is not carried out concurrently with an update, we can redefine the `(setf slot-value-using-class)` method for them. This method is specialized on the second and third arguments (`class` and `object`) which are required to be of class *atomic-class* and *atomic-object* respectively.

```
(defmethod (setf slot-value-using-class)    ; name of the method
                                            ; with the following arguments:
          (value
          (class atomic-class)
          (object atomic-object)
          slot-name)
                                            ; and the following body:
    (unwind-protect                         ; ensure that no control transfers might
                                            ; elude the unlocking of the object
      (progn
        (lock-object object)
        (call-next-method))                 ; invoke the standard method to perform the update
      (unlock-object object)))
```

The lock and unlock operations use the slots *lock* and *requests-queue* for recording respectively which thread is accessing and which ones are waiting to access the object. They can be implemented for instance using the primitives of the Multithread COMMON LISP [Attardi 87].

We call *action* a method which requires exclusive access to an object. To definine an action we provide for convenience the macro `defaction`. For instance, the above method can be written using `defaction` as follows:

```
(defaction (setf slot-value-using-class)
           (value (class atomic-class) (object atomic-object)
            slot-name)
   (call-next-method))
```

The specialized definition of `(setf slot-value-using-class)` for atomic objects ensures that all access operations which modify a slot are executed atomically, i.e. in mutual exclusion from other similar atomic operations, thus avoiding inconsistencies in slot values.

Normal methods can also be defined on atomic objects and they will not interact with the mutual exclusion protocol. For example, since we did not specialize the generic function `slot-value` for atomic objects, reading the value of a slot is performed by a normal method. Therefore it may occurr concurrently with other accesses to the same object, which do not require locking.

An alternative solution for defining actions would have been to exploit the technique of method combination, which allows one to describe how the various methods for a generic function are to be combined to form the effective method. In this case the combination could have been performed by wrapping all the applicable methods for an action with the lock/unlock operations. This solution however would have not worked for a generic function like `slot-value`, whose method combination type is already specified as `standard` in CLOS.

5.2. Persistent Objects

Persistent and shared objects can be integrated in CLOS as an additional object-oriented paradigm. Persistent objects reside on persistent memory support and survive across applications. In order to provide persistent objects we have built an interface between CLOS and the Object Management System (OMS) of PCTE (Portable Common Tool Environment) [PCTE 85]. The OMS is a specialized Data Base Management System of the entity-relationship family designed and aimed at storing objects (entities) that need to be managed in a software engineering environment.

The implementation of our interface consists of a metaclass called *persistent-class*, which defines the behavior of persistent classes. Instances of persistent classes will be persistent and will be sharable among various programs that access the same database. The

common features of all persistent objects are provided by the class *persistent-object*.

```
(defclass persistent-object ()
  ((object-id :accessor object-id)))         ; reference to the OMS object

(defclass persistent-class (standard-class)
  ((class-id :accessor class-id)             ; reference to the OMS class object
   (objects :initform (make-hash-table)))    ; table of objects handles in memory
  (:metaclass class))
```

Normal CLOS operations are applicable to persistent objects, even though the objects are not stored in core, since their implementation is specialized for them. A persistent object is located in secondary memory but programs refer to it through an *object handle*, which is a CLOS instance which contains a reference to the real object. Since persistent objects can be reached from other persistent objects or by associative access, a hash table is used to ensure that only a single handle exists for an object, in order to preserve object identity.

Since whenever a new persistent class is created, we must create a corresponding representation of it in the OMS, we specialize the protocol for creating persistent classes. This is done by defining the MOP generic function `ensure-class-using-class`, like this:

```
(defmethod ensure-class-using-class
          ((class persistent-class) name superclasses slots)
   (call-next-method)
   (setf (class-id class)
        (create-persistent-class name)))
```

The function `create-persistent-class` defines an OMS type for the class, creates an OMS object to represent the class, which will contain links to all the instances of the class and returns a pathname which can be used to access the object. Whenever a new persistent object is created, it is added to the list of instances in its class object, and an object handle is created and returned referring to the object. The association between the object handle in memory and the corresponding object in the database is recorded in the hash table of the class. To perform these additional operations for object creation, we must specialize the generic function `make-instance` to take this into account.

Finally, the generic functions `slot-value-using-class` and `(setf slot-value-using-class)` are redefined for persistent objects so that they are turned into access and update operations onto the fields of persistent objects in the database.

```
(defmethod slot-value-using-class ((class persistent-class)
                                   (object persistent-object)
                                   slot-name)
  (if (eq slot-name 'object-id)
      (call-next-method)
      (get-persistent-slot (object-id object) slot-name)))
```

Persistent objects are stored within OMS, and one must open an OMS schema to work with persistent objects. This is done with `open-schema` to which we supply the name of the schema. In the next example we show how to define a persistent class, create instances and manipulate them.

```
(setq schema (open-schema "/usr/oms/persons"))

(defclass persons (persistent-object)
   ((name :type string :initarg :name)
    (age :type fixnum :initarg :age))
   (:metaclass persistent-class))

(make-instance 'person :name "Mauro" :age 26)

(close-schema schema)

(bye)
```

When a new session starts we can reopen the schema and access the objects it contains or create new objects. A simple construct is available to scan the objects in a class.

```
(setq schema (open-schema "/usr/oms/persons"))

(make-instance 'person :name "Beppe" :age 38)

(for-each (p person)
   (print (slot-value p 'name)))

 Beppe
 Mauro
```

5.3. Graphic Objects

The COMMON LISP User Interface Environment (CLUE) [Oren 88] is a tool for programming user interfaces with X Windows. It provides a basic set of graphic object classes, and facilities for creating objects and using them to control the dialog between an application and its user. CLUE is implemented in terms of CLOS and CLX [Scheifler 88].

In CLUE the basic components of a user interface are *contacts*, which are objects responsible for presenting application information to the user and for notifying the application of the input from the user through the keyboard or an interactive input device.

A user may need to specify or modify values of certain features of an interface, (e.g. colors, fonts, sizes, etc.). Such features are called *resources* and a capability is provided by CLX to store and retrieve resources from a resource database.

A hierarchical naming mechanism is provided to refer to such resources. This allows one to specify a single value for a resource to be shared across instances of separate contact classes. For instance a single resource may describe the font to be used in all contacts which make up a complex graphic interface. By changing this resource, one can change in one shot the font in all the contacts composing the interface.

Each class of contact may use a different set of resources. The names and additional information about them must be specified for each class. A class slot is needed in each contact class to store such information. The standard model of inheritance of class slots for CLOS however specifies that a class slot is shared not only among the instances of the class but also among the instances of its subclasses. So if we were to define a class slot for the class *basic-contact*, which is the root of inheritance for contacts, we would have just one slot and all the contacts would have the same unique description for their resources.

To override the CLOS mechanism of inheritance for class slots, we introduce a special

metaclass, called *clue*, from which contact classes will be created. The metaclass *clue* defines a slot called `resources`. Therefore each instance of metaclass *clue*, i.e. each contact class, will have such a slot, and so each contact class can have a different value for it. Given a specific contact instance, one can access its resource list by applying the `clue-resources` method to its class.

This alternative model of inheritance for class slots is exactly the one proposed in [Cointe 87].

```
(defclass clue (standard-class)
  ((resources :type list :initform nil
              :reader clue-resources))
  (:metaclass standard-class))

(defmethod resources ((bc basic-contact))
  (clue-resources (class-of bc)))
```

6. Conclusions

The CLOS object model proved to be simple and flexible. It allows to easily extend CLOS in order to provide new features useful to deal with AI applications.

Persistent, graphic and atomic objects have been implemented using the metalevel facilities of CLOS.

It is remarcable, expecially in the case of persistent objects, that the results were obtained with a fairly small implementation effort. It was sufficient to specialize only a small set of the MOP generic functions, and piggy-back on the ordinary object operations for most of the rest.

The work reported here has been implemented in DELPHI COMMON LISP (DCL), an implementation of COMMON LISP that includes CLOS, a highly tuned version of CLX, the COMMON LISP programmer's interface to X Windows, and Multithread COMMON LISP , an extension that provides low level primitives for handling concurrency.

A general metalevel paradigm was used to implement these three extensions. The possibility of extending DCL at the metalevel makes it a very general and powerful high level tool for AI applications.

7. References

[Attardi 86] Attardi G, Simi M., "A Description Oriented Logic for Building Knowledge Bases", Proceedings of IEEE, Vol 74, No 10, October 1986.

[Attardi 87] Attardi G., Diomedi S., "Multithread Common Lisp", ESPRIT MADS TR-87.1, July 1987.

[Attardi 88a] Attardi G., Flagella T., "A proposal for efficient automatic synchronization in Concurrent Object Oriented Programming", ESPRIT MADS TR 88.1, June 1988.

[Attardi 88b] Attardi G., Gaspari M., "PCLOS Shared and Persistent Object Hierarchies" ESPRIT MADS TN-88.3, November 1988.

[Attardi 88c] Attardi G., Boscotrecase M.R., "An implementation of CLOS" ESPRIT MADS TN-88.2, November 1988.

[Bobrow 83] Bobrow D.G. and M. Stefik, "The LOOPS Manual", Intelligent Systems Laboratory, Xerox, 1983.

[Bobrow 88] Bobrow D.G., Kiczales G., "The Common Lisp Object System Metaobject Kernel: A Status Report", Lisp and Functional Programming 1988, Proceedings of the ACM Conference on Lisp and Functional Programming, Snowbird, Utah, July 1988.

[CLOS 88] Bobrow D.G. et al., "Common Lisp Object System Specification", X3J13 standards committee document 88-003 (ANSI COMMON LISP), June 1988.

[Cointe 87] Cointe P., "Metaclasses are First Class: the ObjVlisp model", OOPSLA Florida, USA, October 1987.

[Cointe 88a] Cointe P., "Towards the Design of a CLOS Metaobject Kernel: ObjVlisp as a First Layer", Proceedings of the International Workshop on Lisp Evolution and Standardization, Paris, F, February 1988.

[Cointe 88b] Cointe P., "The ObjVlisp Kernel: a reflective Lisp Architecture to define a Uniform Object Oriented System", in "Meta-level Architectures and Reflection" Maes P., Nardi D., (eds), North-Holland, 1988.

[de Rivieres 88] de Rivieres J., "Control-Related Meta-Level Facilities", in "Metalevel Architectures and Reflection", Maes P., Nardi D. (eds), North-Holland 1988.

[Goldberg 86] Goldberg, A., "Smalltalk 80", Addison-Wesley 1986.

[Kiczales 88] Kiczales G., "Proceedings of the First CLOS Users and Implementors Workshop", Palo Alto, California, USA, October 1988.

[Monn 86] Moon, D., "Object-Oriented Programming with Flavors", Proceedings of the ACM OOPSLA Conference, 1986.

[Oren 88] Oren LaMott, K. Kimbrough, "Common Lisp User Interface Environment", Texas Instruments, 1988.

[PCTE 85] "PCTE: A Basic for a Portable Common Tool Environment" Functional Specification BULL et al, 1985.

[Scheifler 88] Scheifler R., "CLX - Common LISP Language X Interface", 1988.

[Simi 88] Simi M., "A Naive Implementation of KRS in CLOS: Report on an Experiment" ESPRIT MADS TR-88.3, November 1988.

[Steele 84] Steele, Guy L. Jr., "Common Lisp: The Language", Digital Press, 1984.