User-Level Language Crafting
Introducing the CLOS Metaobject Protocol

Andreas Paepcke

3.1 Introduction

The idea of open and modular systems is becoming more and more popular in the areas of networking and operating systems. In the former, services like packet transfer may be implemented in different ways without affecting the rest of the system [1]. In operating systems, attempts are made to open functions such as memory paging up to change [2]. CLOS carries this idea into the realm of language design which has traditionally been almost as closed as database implementations.

There are many reasons why language implementations should be open. One important reason is the ever increasing complexity of software development. Its management requires correspondingly more sophisticated tools which must obtain detailed language-internal information, such as class structure or information about methods. Traditionally designed languages often require implementation-specific modifications to compilers or run-time environments which are non-portable because that information is otherwise not obtainable. As the cost of software development rises, such inefficiencies in the creation of integrated environments become less and less tolerable.

The basic idea of the CLOS design is to specify a model for the language implementation and to standardize it. The inner workings of the implementation thereby become manipulable in a controlled manner. This internal model is called the CLOS Metaobject Protocol (MOP)[1].

The goal of this chapter is to explain the basic idea, the important principles and some design issues behind this part of the CLOS language. We make the reader understand why the approach is important and how it works. The material should be sufficient to provide intuition for deciding when the use of the Metaobject Protocol would be appropriate for some given application and how to go about its design.

This chapter shows selected highlights and is not a replacement for an eventual study of the specification in part two of [3], although it should make its consumption easier. We have tried to avoid the complexity caused by a formal specification without sacrificing important information on the material we cover. Part one of [3] is a detailed explanation of the principles of protocol-based design, while this is an *introduction* to the CLOS MOP.

Section 3.2 explains what the Metaobject Protocol is about, what it is trying to do and why it is interesting. Most of this material is kept at an abstract level and does not require deep knowledge of CLOS particulars.

Sections 3.4 and 3.5 are much more concrete. They present selected details of the MOP using an example that is introduced in section 3.3. These sections do assume knowledge of CLOS as explained in [4, 5].

The main body of the chapter closes with pointers to related work and the conclusion. Appendices provide some material about the MOP which is useful for the deeply interested reader but which are too detailed to include in the text.

3.2 The Metaobject Protocol

Beyond trying to be a powerful language in general, CLOS has two additional, unusual goals:

- Allowing users and external programs to *inspect* the internals of CLOS environments.
- Allowing external programs to *extend* the CLOS language itself without modifying existing implementation code and without affecting other, existing programs.

The first of these goals is particularly relevant for the construction of browsers that aid in software development, such as class hierarchy layout displays, and for the implementation of other system analysis tools, such as debuggers. Inspecting the internals includes, for example, the ability to programmatically determine the class structure of a program — without scanning and parsing source code, or to find out which methods are specialized on some class.

The ability to extend or modify the language is necessary to enable experimentation and adjustments to CLOS behavior which may be required to satisfy new applications or system environments. This might include control over how slots are accessed or how instances are made. Enabling non-instrusive modifications can significantly increase return on the investment of designing and building a language because it can be made applicable to a wider range of consumers.

Let us take a first-level look at how these objectives are addressed in CLOS. The approach is at this level quite applicable to designs of other systems that share these goals.

3.2.1 Design Premise and Challenges

When we study the basics of CLOS internals with focus on their openness and flexibility, it is convenient to separate static from dynamic aspects. This partitioning roughly reflects the two goals of the CLOS internal design we listed above and provides a way of organizing the material in our mind.

The static part of the CLOS design may be called its *metalevel architecture*. It describes the components of the system, its structural and procedural building blocks and how they are put together. Examples of major building blocks are the manifestations of classes, slots or methods in the language's implementation.

The dynamic part is described in terms of *protocols* which prescribe the manipulations of the building blocks that must be performed to effect the behavior of the language at run-time. For each 'behavior pattern' of the language, one or more protocols specify how the building blocks must change and interact. Example: everything that is supposed to happen in a CLOS system when a new class is defined is governed by the class *initialization* and *finalization* protocols. They specify the language-internal building blocks and run-time activities that together effect the definition process.

Thus we distinguish between the language itself and a 'metalevel' where its concepts are described abstractly and then implemented. This metalevel world has collectively become known as the CLOS *Metaobject Protocol* and is the focus of this chapter.

Every reasonably designed system has the characteristics we described so far: an external specification of behavior and an internal, hopefully modular model and its implementation. The step CLOS is attempting to take beyond this is to *export* the internal model, to standardize it and to make it part of the final product itself. This final step is what gives this language the desired flexibility and which makes it go beyond many other systems.

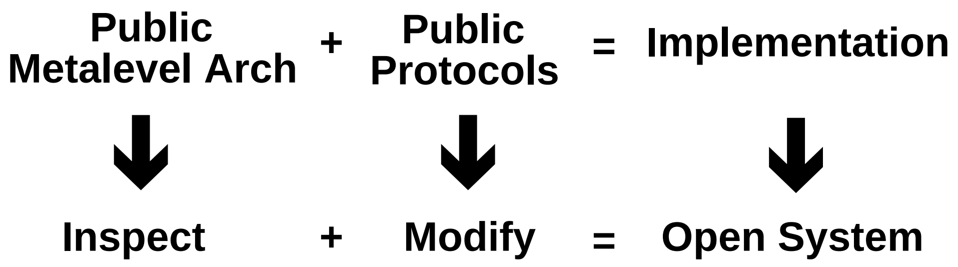

Figure 3.1: The Top-Level CLOS Design Premise

The means to modify CLOS have become **part of the language** and are therefore made as portable as the language itself.

Figure 3.1 tries to illustrate how the metalevel architecture representing statics, and the protocols representing dynamics together make up the CLOS design. The figure also suggests that the existence of these two explicit, standardized components enables us to inspect internals and to modify behavior, which in turn implies that CLOS is an open system — an unusual trait for a language.

All this sounds rather obvious and straight-forward. But designing such a system is difficult. The challenge begins when the proper break-down into building blocks must be decided. This break-down determines how cleanly the eventual implementations will be able to reflect the internal model. It can also determine how far modifications to one building block need to propagate through the system to other building blocks. These are, of course, crucial issues because the organized manipulation of the internal model are the way of modifying the implementation. Clean relationships between them are therefore important.

An even more difficult challenge than finding the proper break-down for the internal model is to find the level of detail to which protocols must be specified. An incorrect level not merely causes inconvenience, but it can lead to system failure. If too much detail is specified, implementations do not have enough room to introduce necessary optimizations. If too little is specified, it becomes unclear where and how modifications must be properly introduced to effect some desired change in behavior. This can lead to a loss in portability of the modifications.

When trying to find the proper balance for standardization questions like these, designers face the dilemma that sample applications are needed to find where the system's degrees of freedom should be placed. Obtaining a significant number of such applications, however, almost requires the a priori existence of the standard. Building an open system like this is therefore generally much more time consuming and frustrating than the construction of a more traditional design[2]. But the payoff is considerable.

3.2.2 Implementation of the Design Premise

We have so far spoken of 'building blocks' and 'behavior' in the abstract. What are these in the concrete case of CLOS? It is a very convenient characteristic of the 'CLOS-producing metalevel world' that it is itself written in CLOS. This unity of language is called *meta-circularity*. The language is itself a CLOS program which is manipulated through techniques of object-oriented programming. This is accomplished through appropriate bootstrapping facilities which do not need to concern us here. The ability to modify the language's implementation without leaving the realm of the language is called *reflection*.

Unity of language makes the life of the metalevel manipulator much easier. Instead of needing to learn a new configuration or implementation language, we can freely move between 'regular programming' and metalevel programming without having to switch languages and our way of thinking.

In particular, the metalevel architecture is defined and implemented as a CLOS class hierarchy. Instances of these classes implement elements of the CLOS object model at run-time. CLOS classes or methods, for instance, are themselves instances of classes at the metalevel. We will introduce these classes in section 3.4. Extensions to this *static* part of the CLOS implementation are made by subclassing the classes at the metalevel.

The *dynamics* of CLOS are captured in a set of generic functions and methods specialized on these classes. The protocols describe the main activities of these generic functions and explain which of them must be invoked to implement the behavior patterns of the language. Extensions and modifications of the dynamic part of CLOS are therefore usually implemented by defining new methods on existing system generic functions.

This uniformity of the metalevel and CLOS-level worlds does have the potential of causing confusion in that we must keep track of whether we are, for instance, talking about classes a regular programmer would define, or classes at the metalevel, which are pieces of the CLOS implementation.

The term *metaobject class* is used to denote a class at the metalevel. Instances of these classes are called *metaobjects*. A metalevel instance that implements a CLOS generic function or a CLOS programmer-level class is therefore a metaobject.

3.2.3 A More Detailed View

Let us pull together what we know so far about the CLOS design and its metalevel world and add some new pieces.

Figure 3.2 shows how we could view the system. A regular user of CLOS would be at the bottom of the figure 'looking up'. Regular programmers do not modify the language itself. They create metaobjects through definitional macros, such as the familiar `defclass`, `defmethod` or `defgeneric`. They unwittingly use these metaobjects by such activities as creating instances of classes and invoking generic functions.

Metalevel programmers perform the same activities, but they also handle metaobjects more consciously. In particular, they use mechanisms such as `find-class`, `find-method` or `symbol-function` which take a name and return an associated metaobject. `Find-class`, for example, takes the name of a programmer-level class and returns the metaobject that implements it.

Metalevel programmers also work with the static and dynamic parts of the language implementation by subclassing and by adding methods to system generic functions.

At the center of figure 3.2's upper portion we see the snapshot of a run-time collection of metaobjects which implement some running CLOS program. They are surrounded by the major design components which control them: the metaobject class hierarchy defining the static setup of the metalevel world. The exported, standardized system generic functions and methods which provide the implementation of the dynamic aspects and the protocol component which controls what the dynamic component does.

The next section explains some restrictions that are imposed on manipulations of the internal model.

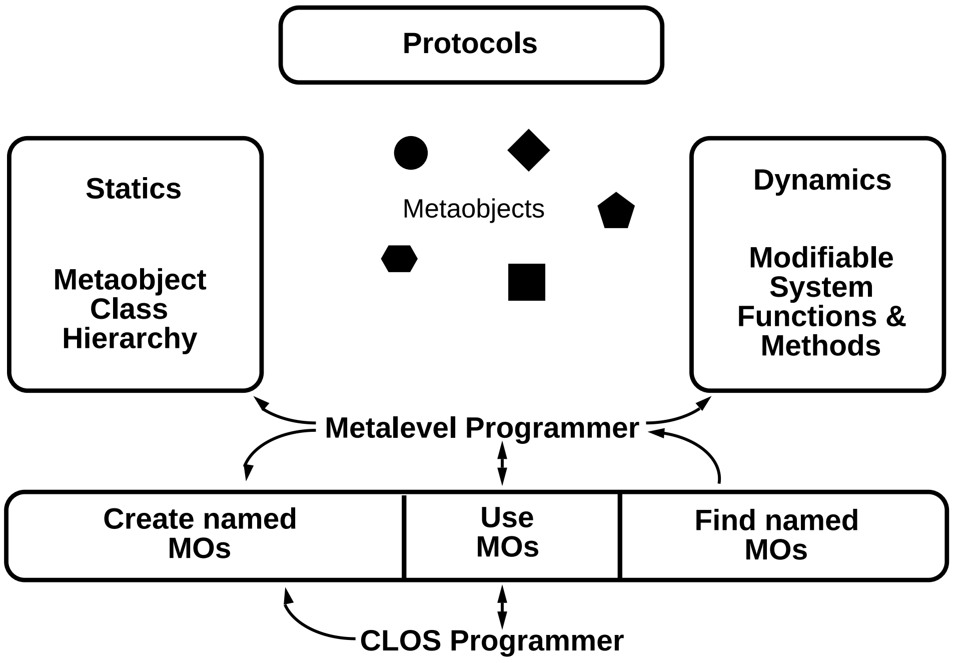

Figure 3.2: The Overall CLOS Design

3.2.4 Curbing Chaos

We have seen that the main tools of the metalevel programmer are subclassing and the definition of methods on exported system generic functions. Indiscriminate use of these tools can prevent a system from functioning properly. The problem lies in the fact that programmers build modules under the assumption that the language they work with is immutable. If the loading of one module changes the language, other modules can fail unless special care is taken.

The MOP does not include enforced safeguards against conflicts arising from metalevel manipulations. Instead, there are rules regarding these activities which are intended to ensure that extensions made at the metalevel are portable and do not destroy the system for other programs running in the same environment. One reason for such extreme openness is that radical modifications do have their place. One example has been the reduction of CLOS to a very small, fast, low-functionality delivery kernel after the completion of program development [6]. In general, however, programs will need to be portable, which means that they will need the ability to coexist with other, independently produced programs. This includes metalevel programs.

The rules regarding metalevel work all have the same purpose: To ensure that new behavior does not change *existing* system behavior that is relied upon by others. Appendix A contains a list of these rules.

Before we begin to introduce details of the Metaobject Protocol, we describe the skeleton of an application which we will use throughout the subsequent sections to illustrate how all the facilities can be put to use.

3.3 An Example Problem

As an example for the use of the Metaobject Protocol let us imagine that we want to add persistence to the objects in CLOS programs [7, 8].

We assume that objects may be either transient or persistent. The state of each persistent object is stored in a database and retrieved from there as needed. We make objects persistent by sending them the message `make-persistent`. This will cause the database to be prepared to receive the object's state and will then transfer the state there.

Objects may be cached, which means that their state is withdrawn from the database and stored in memory until it is explicitly returned to the database. Whether a persistent object is cached or not, it is always possible to send messages to it as if it were transient. There is to be no semantic difference between these object states, other than the persistence of values. If a slot of an uncached, persistent object is read, the slot value is retrieved from the database and returned as if it had been stored in memory. Slot updates are propagated to the database.

For reasons of efficiency and for some other technical reasons, it is desirable to allow individual slots to be transient. The value of a transient slot is not placed in the database but is always memory-resident, even if the object as a whole is made persistent. The programmer may declare individual slots to be transient when the class is being defined. In cases where some slot is provided by more than one superclass, we assert that transience is legal for the slot only if *all* superclasses have declared it to be transient. Otherwise it must be made persistent.

One tricky problem is caused by class redefinition, which CLOS makes easy to accomplish: we must create some appropriate schema in the underlying database which corresponds to the class hierarchy of the program that will generate the persistent instances. If this hierarchy changes, the schema will have to evolve as well. We will not cover how this can be accomplished in the database — that is a research issue in itself. We will merely point out how we can use the MOP to cause schema evolution to be initiated when necessary.

Given this problem description, how must we change the behavior of standard CLOS to accommodate a solution:

- We must be able to programmatically examine classes so that we can build appropriate schemas in the underlying database.
- The definition and redefinition of classes must be trapped to allow schema creation and evolution to be triggered.
- We need to manipulate the class inheritance.
- A new slot option must be introduced into the language to allow slots to be declared transient.
- Information about which slots are transient must be stored somewhere in the runtime system.
- Without the programmer being aware, internal information must be kept with each instance that is created. An important such piece of information is whether that particular instance is currently persistent or not.
- Additional information must be kept with each class. This might include information about how the database must be accessed or special caching policies for instances of that class.

- Slot access must be intercepted to implement faulting to the database.

Even a cursory glance at this list of requirements shows that these are significant modifications to any language and cannot be accomplished by working outside the language implementation. Our strategy will be to define a *class* metaobject class called `persistent-metalevel-class`. When a programmer defines a class whose instances are to have the potential of being persistent, she specifies that `persistent-metalevel-class` is to provide for that class' implementation.

We will define a programmer-level class `persistence-root-class` which provides some methods for persistent objects, such as `cache` and `make-persistent`. We will have `persistent-metalevel-class` take care of mixing that class into persistent user classes transparently.

Clearly, a full-scale persistent object system will need to do more than what we describe in this skeleton, but it turns out that this subset covers the language incisions that are necessary for such systems. It is therefore well suited to illustrate what we have to say about the details of the Metaobject Protocol.

In the following section we go into the details of the MOP's structural parts.

3.4 Metalevel Statics

We explained above that the structural part of the Metaobject Protocol reflects a breakdown of CLOS into basic concepts which is itself reflected in the metalevel class hierarchy. It is, of course, important to understand this hierarchy, as it is the key to making structural modifications and to accomplishing inspection of program internals.

The main building blocks are:

1. *Classes*
2. *Slots*
3. *Methods*
4. *Generic Functions*
5. *Method combination*

Each of these is represented by a class subtree at the metalevel whose terminals are the sources of the corresponding metaobjects.

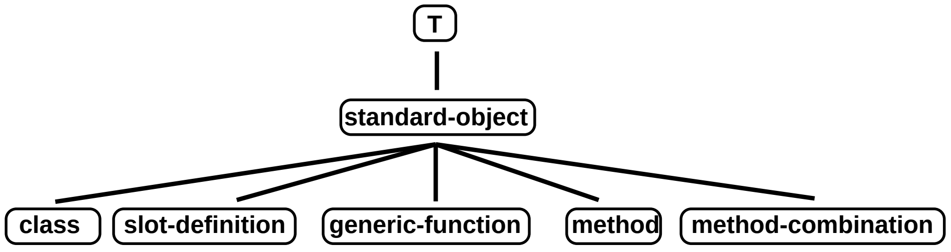

Figure 3.3: The Top-Level MOP Class Hierarchy

Figure 3.3 summarizes this.

In this section we will take several of these building blocks in turn and will explain their structural properties. Please note that we will not show the complete subtrees of a typical CLOS implementation. We try to extract the subclasses most likely to be of general interest to avoid confusion. It should not be necessary to understand more of the hierarchy.

Remember that the interface to the metalevel world provides us with powerful ways of finding out about the structural properties covered here. We can use `find-class <class-name-symbol>` to obtain instances of any of the *class* metaobjects we talk about. Using `describe` on those will reveal much useful information. Browsing the implementation in this way is indeed a very good way of getting acquainted with the system.

3.4.1 The *Class* Metaobject Class

The most frequently inspected and modified building block is the CLOS **class** since many important methods are defined on it and it contains a large amount of information useful for debugging and program maintenance. As a rule of thumb, if desired information is usually specified in a `defclass`, the resulting *class* metaobject is the place to find that information later on[3]. Standard CLOS comes with several *class* metaobject classes built in.

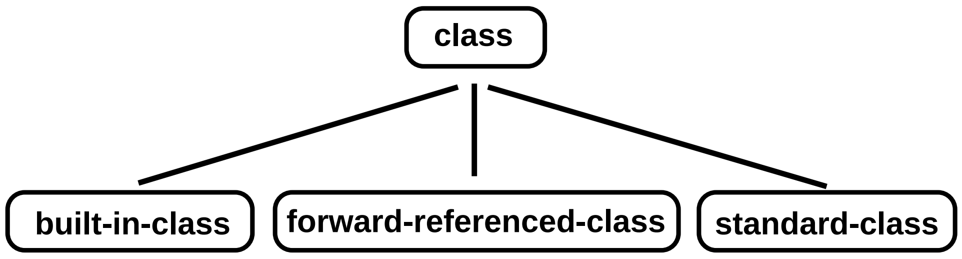

Figure 3.4: The *Class* Metaobject Class Subtree

Figure 3.4 shows some of these. The most important is `standard-class` since its instances are the metaobjects which by default implement the classes a programmer defines with the `defclass` macro. Most new metaclasses a user might want to write will be subclasses of `standard-class` and we concentrate on it here. But since most metalevel work tends to cause programmers to come across some of the others in passing, we mention their role briefly:

Instances of `built-in-class` implement classes that are not specified using `defclass` but are pre-constructed by CLOS implementations. Examples are classes that are made to correspond to standard CommonLisp types. `Built-in-class` metaobjects have various special properties, like the fact that they may not be redefined.

The `forward-referenced-class` is used when a programmer defines a class whose superclasses are not yet defined. In that case a metaobject of class `forward-referenced-class` is created to act as a 'place holder' until the superclass is defined later on.

The following information is kept in a `standard-class` metaobject. It is easy to see the correspondence between what a `defclass` specification contains and the information listed here. Indeed, the class metaobject is where most of the `defclass` entries end up. This information is available and we list the published reader function names for each of the items in parentheses.

As an example for the use of this information, assume the existence of a programmer-level class `train`. We could find its direct superclasses through:

```
(class-direct-superclasses (find-class 'train))
```

- The slots of the class are kept as a list of *slot* metaobjects. Reader `class-slots` returns all slots, including the inherited ones, `class-direct-slots` returns just the ones defined for this class explicitly.
- The super- and subclasses are stored as a list of *class* metaobjects (`class-direct-superclasses` and `class-direct-subclasses`).
- The class precedence list is recorded as a list of *class* metaobjects (`class-precedence-list`).
- The default initialization arguments for the class are kept. Reader `class-default-initargs` returns all initargs, including the ones inherited from superclasses while reader `class-direct-default-initargs` returns only the ones specified for the respective class directly.
- Information on whether the class has already been finalized is also available (This will be false if, for example, there were undefined superclasses at the time the *class* metaobject was created.) (`class-finalized-p`).

We can now introduce the first of the modifications our persistent object example requires: the storage of additional information in *class* metaobjects. We define a new metaclass:

```
(defclass persistent-metalevel-class (standard-class)
  ((checked-schema-congruence-p :initform NIL
                                :reader class-checked-schema-congruence-p)
   ))
```

It adds a new slot to *class* metaobjects which allows us to record whether we have checked that the structure of the class conforms with any schema we might have built earlier in the database to hold persistent objects of this class.

Now we can define our first persistent programmer-level class:

```
(defclass hypertext-node ()
  ((contents  :initform "" :accessor contents)
   (in-links  :initform NIL)
   (out-links :initform NIL))
  (:metaclass persistent-metalevel-class)
  )
```

This is a good time to make sure that easy-to-arise confusion between the metalevel and the regular CLOS level is avoided: at this point we have a programmer-level CLOS class called `hypertext-node` which contains the three slots `contents`, `in-links` and `out-links`. This class is all a regular CLOS programmer ever works with. If we now move into the metalevel world, we find out that this class is in reality a metaobject which is an instance of the *class* metaobject class called `persistent-metalevel-class`. Since that inherits from `standard-class`, it presumably has some slots we have no access to (the reader functions listed earlier provide all the information we are supposed to have). But we have added the additional slot for the schema congruence check whose value is available to us. This slot is therefore part of the metaobject, **not** part of any future programmer-level instances of `hypertext-node`.

With this clarified, let us get a hold of the class metaobject and find out some details about it (system responses are indented):

```
(setf hypertext-class-metaobject (find-class 'hypertext-node))

(class-direct-slots hypertext-class-metaobject)
  (#<Standard-Slot-Definition CONTENTS>
   #<Standard-Slot-Definition IN-LINKS>
   #<Standard-Slot-Definition OUT-LINKS>)

(class-precedence-list hypertext-class-metaobject)
  (#<Persistent-Metalevel-Class HYPERTEXT-NODE>
   #<Standard-Class STANDARD-OBJECT>
   #<Standard-Class T>)

(class-checked-schema-congruence-p hypertext-class-metaobject)
  NIL
```

We can also begin to add some behavior to our new metaclass which allows us to build a database relation based on the slots of the class and to record in the database some information about the class itself. We assume that we have a database object `*database*`. This object may be used for calls to generic functions that manipulate a database consisting of tables. We assume further that the database contains a special table called "master-class-table" which we initialized earlier and in which class-related information is stored. Tables can be searched by key and we can add and delete rows:

```
(defmethod create-schema ((class persistent-metalevel-class))
  (create-table *database* (class-name class) (class-slots class)))

(defmethod store-class-structure ((class persistent-metalevel-class))
  (unless (find-entry *database* 'master-class-table (class-name class))
    (add-row *database*
             'master-class-table
             (class-name class)
             (class-slots class)
             (class-precedence-list class))))
```

3.4.2 The *Slot-definition* Metaobject Class

Slots in CLOS and other object-oriented languages are more than a physical place to store a value. Issues of typing, initialization and accessability must be remembered and managed. This is why the second major building block of the Metaobject Protocol is the `slot-definition` metaobject. We use the `class-slots` generic function on the *class* metaobject to get a hold of `slot-definition` metaobjects. Recall that this returns a list of the class' slots.

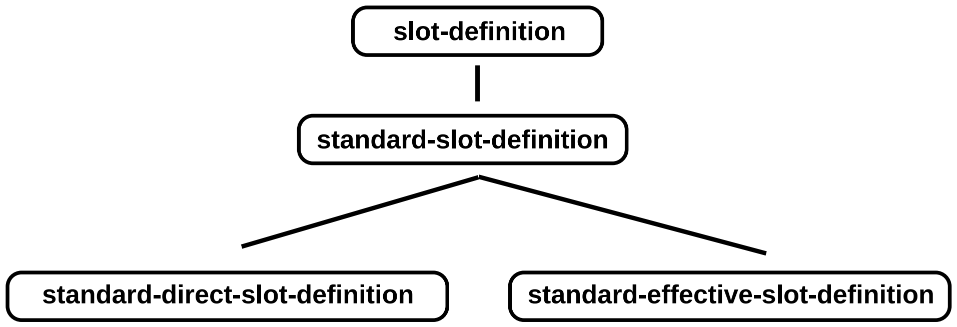
slot-definition
standard-slot-definition
standard-direct-slot-definition
standard-effective-slot-definition

Figure 3.5: The *Slot-definition* Metaobject Class Subtree

Figure 3.5 shows part of the relevant metaobject class subtree. We see that there are two main branches: `standard-direct-slot-definition` and `standard-effective-slot-definition`. Instances of the first hold the 'raw', 'untreated' slot-related information from the class definition form, while instances of the second hold information that reflects the actual, run-time properties of the slots after the CLOS inheritance rules have been applied. If, for instance, a slot is defined with the `:initarg` slot option[4]set to a value different from the same option in a slot it shadows, the `standard-direct-slot-definition` will show the child's initialization argument, while the `standard-effective-slot-definition` will show a list of the initialization arguments containing both the child's and the parent's specification.

Recall that we can extract a list of direct slot definitions and effective slot definitions from a *class* metaobject class by using the two accessors `class-direct-slots` and `class-slots` respectively. Once we have a slot definition metaobject in hand, we can extract the following information:

- The slot name, type and allocation may be obtained through `slot-definition-name`, `slot-definition-type` and `slot-definition-allocation`, respectively.

- The initialization form that was supplied in the `defclass` may be retrieved from a `slot-definition` by means of `slot-definition-initform`. If such an initform has been supplied, the initialization process of the class will also have provided a function with no arguments which returns the initform value. Thus, if a slot was defined with the `:initform` option `(+ 1 2)`, the method `slot-definition-initform` will return `(+ 1 2)`, while `slot-definition-initfunction` returns something like: `#<Interpreted-Function (LAMBDA NIL (+ 1 2)) 1238467>`. The form `(funcall (slot-definition-initfunction <slot-definition-metaobject>))` returns 3.

- The methods `slot-definition-initargs`, `slot-definition-readers` and `slot-definition-writers` return lists of the slot initialization argument(s) and reader/writer function specifier(s), respectively.

Note that the slot *value* is **not** stored in the slot definition metaobjects. Remember that there is only one such slot definition metaobject per slot per class. Since every instance has its own value for the slot, such an implementation would be incorrect.

Our persistent object system will augment the internal representation of slots by adding information on whether a slot is transient:

```
(defclass persistent-standard-direct-slot-definition
    (standard-direct-slot-definition)
  ((transientp :initform NIL :reader slot-definition-transient-p)))
(defclass persistent-standard-effective-slot-definition
    (standard-effective-slot-definition)
  ((transientp :initform NIL :reader slot-definition-transient-p)))
```

In section 3.5 we will see how the MOP may be influenced to use these classes instead of their parents when constructing one of our persistent classes.

3.4.3 The *Method* Metaobject Class

Methods are the next building block of the Metaobject Protocol. The metaobjects that implement them hold all the information associated with methods. This includes the information specified in the defining `defmethod`. We can get a hold of *method* metaobjects by using `find-method` as follows:

```
(find-method <generic-function-meta-object>
             <list-of-qualifier-keywords>
             <list-of-class-metaobjects>)
```

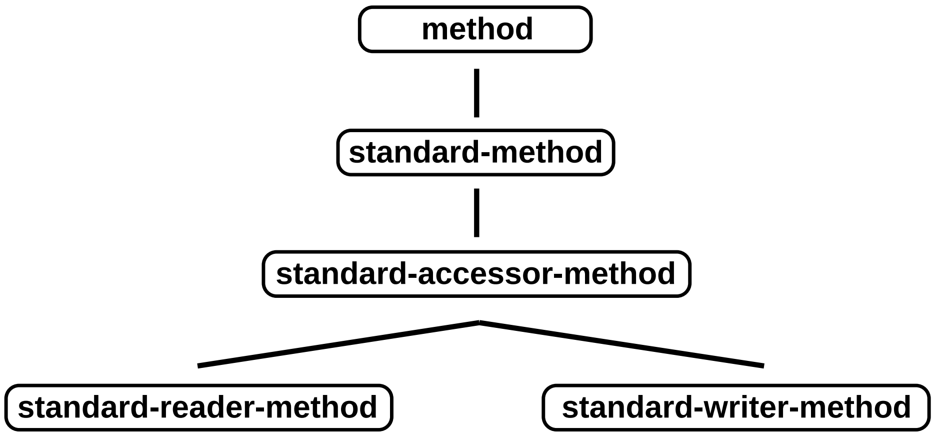

Figure 3.6: The *Method* Metaobject Class Subtree

Figure 3.6 shows part of the relevant metaobject class subtree. In order to illustrate the kind of information we can extract from method objects, let us define two hypothetical methods for the persistent hypertext class defined earlier on:

```
(defmethod linking ((source-node hypertext-node)
                    (destination-node hypertext-node))
  (push destination-node (slot-value source-node 'out-links))
  (push source-node (slot-value destination-node 'in-links)))
```

The following `:before` method allows us to observe the linking together of nodes at run-time:

```
(defmethod linking :before ((source-node hypertext-node)
                            (destination-node hypertext-node))
           (format t "Creating link from ~S to ~S.~%"
                   source-node destination-node))
```

Here is how we can obtain the method metaobjects that implement these two methods:

```
(let ((linking-gen-func (symbol-function 'linking)))
  (setq *primary-method* (find-method linking-gen-func
                                      nil
                                      (list (find-class 'hypertext-node)
                                            (find-class 'hypertext-node))))
  (setq *before-method* (find-method linking-gen-func
                                     '(:before)
                                     (list (find-class 'hypertext-node)
                                           (find-class 'hypertext-node)))))
```

Let us see some of what we can find out about these two methods:

- The generic function a method is currently associated with is returned by `method-generic-function` as a generic function metaobject.
- We can find the lambda list and the list of specializers of a method by using `method-lambda-list` and `method-specializers`. Both return lists. The first is a list of the argument names without any of the classes they are specialized to. The second is a list of *class* metaobjects. For both of our methods these would be:

  ```
  (SOURCE-NODE DESTINATION-NODE)
  (#<Persistent-Metalevel-Class HYPERTEXT-NODE>
   #<Persistent-Metalevel-Class HYPERTEXT-NODE>)
  ```

- The qualifiers of a method, finally, are obtained through `method-qualifiers`. This returns a list of qualifier specifications as they are used in the `defmethod` macro. Our primary method would return `NIL`, the `:before` method would return `(:before)`.

This concludes our look at the static part of the Metaobject Protocol. The information presented should be sufficient to extract a large amount of interesting information from the run-time environment of a CLOS program. In the next section we turn to the dynamics of the Protocol.

3.5 Metalevel Dynamics

When we want to go beyond inspection to modifying the behavior of the language, we will often modify the static part of the MOP by subclassing. Most of the time we will then need to modify parts of the dynamics as well. Many times this will involve initializing new information we keep in our metalevel subclasses. Sometimes there will be other run-time work to be taken care of as well. The goal of this section is to explain the sequences of events that take place to effect some of the major behavior patterns of CLOS. This information should be sufficient to locate where to 'hook in' to change these patterns.

As explained in section 3.2, the dynamics of the MOP are captured in a set of protocols. Here is a list of some major ones:

- The class initialization and class finalization protocols control what happens when a new class is defined.
- The instance initialization protocol describes what goes on when a new instance is created and readied for use.
- The dependent maintenance protocol helps in maintaining relationships among metaobjects. Examples are classes and their subclasses, or generic functions and their methods.
- The method lookup protocol determines how the correct method is found when a generic function is invoked.
- The instance structure protocol attempts to formalize just enough of the implementation of instance access to allow for organized modification, while leaving sufficient freedom for implementors.

We will begin with a tour through the process of defining a new class. The creation and initialization of *slot* metaobjects is part of this process. This will be followed by a description of how slot access works[5].

A theme common to most of metalevel CLOS initialization is that a user's definitional macros, such as `defclass` are checked for errors. Then the information supplied is brought into a canonical form that makes further processing easier. Once this has been accomplished, metalevel functions are invoked to create metaobjects and to initialize them.

After the error checking and canonicalization, these processes are the same whether they were initiated by the execution of a defining macro, or whether they were begun by programs. There is, for instance, a specific point in the processing of a class definition where programs wishing to define a new class would begin. We will point out those programmatic entry points into the metalevel machinery as we encounter them.

3.5.1 Class Definition

When CLOS classes are first defined, their superclasses need not necessarily have been defined yet. *Class* metaobjects must therefore be created and initialized as far as possible, without necessarily knowing all necessary details. Later, once all information is available, the inheritance of the class is *finalized.* The following two subsections explain how this happens.

3.5.1.1 Initialization

Figure 3.7 gives a simplified overview of what happens during class initialization. A full overview is available in appendix B. This figure, and similar ones later on in the text, list the various activities that must take place during the course of the protocol. A series of indented subactivities below an entry shows the steps necessary to accomplish that entry. For example, (re)initialization of the *class* metaobject involves the superclass compatibility check and the other subactivities at the same level of indentation. Successive levels of indentation thus represent increasing levels of detail.

Generic functions listed below an entry in parentheses are responsible for accomplishing that entry's task, often with help from the functions listed further down the list. When appropriate, we point out in the figures the functions through which programs may initiate protocols that are normally initiated through macros, such as `defclass`. The figures serve two purposes: to give a quick overview of a protocol and to show the 'hooks' available to effect changes.

DEFCLASS
1 Syntax error checking
2 Canonicalize information
3 Obtain *class* metaobject
`(ensure-class,` ⟵ **Programmatic entry**
`ensure-class-using-class)` ⟵ **Programmatic entry**
3.1 Find or make instance of proper *class* metaobject class
`(make-instance, the :metaclass option)` ⟵ **Programmatic entry**
3.2 (Re)initialize the *class* metaobject
`((re)initialize-instance)`
3.2.1 Check compatibility with superclasses
`(validate-superclass)`
3.2.2 Determine proper *slot-definition* metaobject class
`(direct-slot-definition-class)`
3.2.3 Create and initialize the *slot-definition* metaobjects
`(make-instance, initialize-instance)`
3.2.4 Maintain the 'subclasses' lists of superclasses
`(add-direct-subclass,remove-direct-subclass)`
3.2.5 Initiate inheritance finalization, if appropriate
`(finalize-inheritance)`

Figure 3.7: Summary of the Class Initialization Protocol

Let us touch on the main pieces of the class initialization process a step at a time.

`DEFCLASS` Expansion. The goal of the `defclass` macro expansion is to produce a call to the function `ensure-class` which will create the actual *class* metaobject. It is also used for redefining existing classes.

```
ensure-class <name> &key :environment
                    &allow-other-keys
```

Note that this is a regular function, not a generic one because when it is called we have no instance whose class we would specialize to. To illustrate the processing from `defclass` to `ensure-class`, consider the following subclass of our hypertext node:

```
(defclass monitored-hypertext-node (hypertext-node)
  ((access-count   :initform 0 :accessor access-count)
   (security-level :reader security-level))
  (:metaclass persistent-metalevel-class))
```

Here is roughly what we will end up with when this macro is expanded:

```
(ensure-class 'monitored-hypertext-node
              ':direct-superclasses '(hypertext-node)
              ':direct-slots (list (list ':name 'access-count
                                         ':initform '0
                                         ':initfunction #'(lambda () 0)
                                         ':readers '(access-count)
                                         ':writers '((setf access-count)))
                                   (list ':name 'security-level
                                         ':readers '(security-level)))
              ':metaclass 'persistent-metalevel-class)
```

We see that all but the name information from the `defclass` form is passed to `ensure-class` through keyword arguments. The specification of slots deserves special attention. It is the result of step 2 in figure 3.7 and takes the form of a list of *canonicalized slot specifications*. Each of these is itself a list of keyword-value pairs which will be used as keyword arguments when making *slot-definition* metaobjects later on.

This technique of preparing information into a form that can be used directly as an initialization argument later on is a common canonicalization method in the MOP and we will see other examples of it.

The `:initform` entry relays the form that was specified in the class definition. The `:initfunction` entry is a function that, when called, will return the proper initial value.

The next step in the initialization process happens in the generic function `ensure-class-using-class` which is the workhorse of `ensure-class` and is specialized to a particular *class* metaobject class or to `null`.

```
ensure-class-using-class <class> <name> &key :metaclass
                                             :direct-superclasses
                                             :environment
                                        &allow-other-keys
```

It is called either with a *class* metaobject bound to `<class>`, indicating that we wish to *redefine* a class, or with `NIL`, indicating that we are to create a new one.

`Make-instance` is used to create new *class* metaobjects and the regular CLOS instance initialization procedures are used to get the class ready for use: `initialize-instance` takes care of fixing up a new class, `reinitialize-instance` handles existing classes that are to be redefined. Here is what the class initialization protocol calls for when defining a new *class* metaobject.

Superclass Compatibility Check. The first job is to convert the superclass names from the `defclass` form into *class* metaobjects and to make sure there is no clash. This can happen, for instance, when the class being defined and one of its superclasses are of different metalevel classes. The compatibility check is done by the generic function:

```
validate-superclass <class-metaobject> <superclass-metaobject>
```

When constructing a new *class* metaobject class, the designer must decide whether a programmer-level class implemented by his new metalevel class and inheriting from a super that is implemented by a different metalevel class would lead to inconsistencies.

Unless we define a method on `validate-superclass`, the following will lead to an error because the proper `:metaclass` option was not specified and the system therefore defaulted to using `standard-class`:

```
(defclass simple-hypertext-node (hypertext-node)
  ((slot1)))
```

If we were sure that our new metaclass followed a protocol compatible with `standard-class`, we would provide:

```
(defmethod validate-superclass
           ((class persistent-metalevel-class)
            (superclass standard-class))
  t)
```

This would make the above class definition work. Note that incompatibilities can be a pervasive problem because they prevent the user from inheriting existing superclasses which are not under his control. If, for instance, someone else had provided an interesting 'text display' class facility that we want to reuse by mixing it in with `hypertext-nodes`, we must either certify compatibility in a `validate-superclass` method, or that provider must be asked to change his class to use the `:metaclass persistent-metalevel-class` option in his class definition.

Slot Definitions. Next in the process of class definition is the creation of an appropriate *slot-definition* metaobject for each slot which contains the 'untreated' information specified in the `defclass` definition. Recall that 'untreated' means that slot conflicts with inherited attributes have not been resolved yet. Once these metaobjects exist, dealing with the slots in the later stages of class initialization and finalization will be more convenient. The generic function `direct-slot-definition-class` is called with the *class* metaobject and the canonicalized slot definitions to find out which *slot-definition* metaobject class should be instantiated to implement each slot. This allows the slot implementation to be controlled either by the *class* metaobject class or by new slot options an implementor might introduce.

The choice of slot implementation is something we need to take care of in our persistence example. Recall that we defined a new `persistent-standard-direct-slot-definition` metaobject class and we must make sure that it is used, instead of `standard-direct-slot-definition`:

```
(defmethod direct-slot-definition-class
    ((class persistent-metalevel-class) initargs)
  (declare (ignore initargs))
  (find-class 'persistent-standard-direct-slot-definition))
```

This will ensure that all slots in persistent classes will be implemented with *our* slot metaobjects. The initargs contain the information about the slot that was provided in the `defclass` form, such as `:initform`, `:allocation` or `:type`. This information may be needed by some methods on this generic function to make their decision, though we do not require it for our purposes here.

Creating Slot Definitions. When `make-instance` is used to create a direct slot definition, all the slot options from the `defclass` form are passed in as initialization arguments. The `standard-direct-slot-definition` metaobject classes therefore have initialization arguments corresponding to each legal slot option. These arguments are then processed and installed in the direct slot definition metaobject as we have seen in section 3.4.

We will need to make some changes to introduce our new `:transient` slot option into the system. As things stand, a class definition, such as:

```
(defclass foo ()
  ((slot1 :transient t)))
```

would produce an error, such as:

```
>>Error: Invalid initialization argument :TRANSIENT for class
         STANDARD-DIRECT-SLOT-DEFINITION
```

In order to allow this new option, we modify our definition for *slot* metaobjects introduced in section 3.4.2 to include an `:initarg` option:

```
(defclass persistent-standard-direct-slot-definition
   (standard-direct-slot-definition)
  ((transientp :initform NIL
       :initarg :transient
       :reader slot-definition-transient-p)))
(defclass persistent-standard-effective-slot-definition
   (standard-effective-slot-definition)
  ((transientp :initform NIL
       :initarg :transient
       :reader slot-definition-transient-p)))
```

After the appropriate direct slot definition metaobject has been created and initialized for each slot specified in the `defclass`, the list is kept with the metaobject so that `class-direct-slots` can retrieve and return it.

Maintaining Class Hierarchy Pointers. Recall that we are to be able to ask for all direct super- and subclasses of any class. Since we validated and recorded the superclasses of our new class as part of this initialization process earlier, we know how the information for the former is obtained. But something must still be done to maintain the information for the latter. This is done through the generic functions:

```
add-direct-subclass <superclass-metaobject> <class-metaobject>
remove-direct-subclass <superclass-metaobject> <class-metaobject>
```

When a class is first defined, a call is made to `add-direct-subclass` for each of the new class' supers. In case of reinitialization, a combination of both functions is used to ensure that all class 'downpointers' are correct.

With this the initialization process of the new *class* metaobject is complete. At some point between now and the time the first instance is made, the final, inheritance-related issues must be resolved.

3.5.1.2 Inheritance Finalization

The class finalization protocol is responsible for controlling everything that has to do with a class' inheritance.

Figure 3.8 shows what needs to happen during the finalization of a class. A full overview is included in appendix B.

Let us go through the protocol a step at a time.

FINALIZE-INHERITANCE ⟵ **Programmatic entry**

1 Compute the class precedence list
  (`compute-class-precedence-list`)

2 Resolve conflicts among inherited slots with the same name

  2.1 Determine proper effective slot definition metaobject class
    (`effective-slot-definition-class`)

  2.2 Create the effective slot definition metaobjects
    (`make-instance`)

  2.3 Initialize the effective slot definitions
    (`initialize-instance, compute-effective-slot-definition`)

Figure 3.8: Summary of the Class Finalization Protocol

The Class Precedence List. The generic function:

```
compute-class-precedence-list <class-metaobject>
```

computes the linearized list of class metaobjects that are in the hierarchy above the class being finalized. The default methods do this according to the rules of official CLOS. We will make a small change here that causes all persistent classes to inherit a class which provides some persistence-related methods, such as `cached?`, `persistent?`, `make-persistent` and so on. We first define that class:

```
(defclass persistence-root-class ()
  ((persistent? :initform T)
   (cached? :initform NIL))
  (:metaclass persistent-metalevel-class))
```

We see that this service class also introduces some slots that are used for house keeping. This is our way of adding system information to **each instance** of our persistent world. Now let us 'sneak' this class into the class precedence list of every persistent class. We do this right when a class is defined.

The `member-if` statement in the following method looks at each superclass in turn and finds out whether any of them is a persistent class. If yes, that super already provides the service class through inheritance and we do nothing special. Otherwise we add our service class to the list of direct superclasses. The `apply` is necessary to make the keyword manipulation work:

```
(defmethod initialize-instance :around
  ((class persistent-metalevel-class)
   &rest all-keys
   &key direct-superclasses)
  (let ((root-class (find-class 'persistence-root-class))
        (pobjs-mc (find-class 'persistent-metalevel-class)))
    (if (member-if
         #'(lambda (super)
             (eq (class-of super) pobjs-mc)) direct-superclasses)
```

```
      (call-next-method)
      (apply #'call-next-method
             class
             :direct-superclasses (append direct-superclasses
                                          (list root-class))
             all-keys))))
```

The next major step in the class finalization is the coalescence of slots: The system needs to find the slots that are defined in multiple classes and must resolve any conflicts that arise in the details of their specifications, such as required value type or initialization.

Resolving Slot Inheritance Conflicts. The first entry point to the slot coalescence activity is:

```
compute-slots <class-metaobject>
```

Its final outcome is a list of `effective-slot-definition` metaobjects, each of which contains all information about one coalesced slot. `Compute-slots` first collects groups of all like-named direct slot definitions from the superclasses and then repeatedly calls the generic function:

```
compute-effective-slot-definition <class-metaobject>
                                  <slot-name>
                                  <direct-slot-definitions>
```

There is one call to this function for each group of conflicting slots. Each time, a single `effective-slot-definition` metaobject is created and returned. As explained earlier, the complete list of these is available through `class-slots` when the process is finished.

As an example, consider a class and its superclass which both provide a slot named 'contents'. The class initialization procedures of the two classes would each have produced one `direct-slot-definition` metaobject which would be kept with the respective class metaobject. During finalization of the subclass, `compute-slots` would construct a list of these two `direct-slot-definition` metaobjects and would call `compute-effective-slot-definition` with that list. The result would be a single `effective-slot-definition` metaobject that records the 'net' properties of the slot for instances of the subclass.

Analogous to the mechanism that allowed `ensure-class-using-class` to create proper direct slot definition metaobjects, the generic function `effective-slot-definition-class` is used to determine which metaobject class should be used for effective slot definition metaobjects. Recall that we defined `persistent-standard-effective-slot-definition` earlier on and we need to ensure that the system uses this class instead of the default:

```
(defmethod effective-slot-definition-class
    ((class persistent-metalevel-class) initargs)
  (declare (ignore initargs))
  (find-class 'persistent-standard-effective-slot-definition))
```

Now we need to ensure that our inheritance rules regarding slot transience will be enforced: A slot will be treated as transient only if all classes in the inheritance chain that define a slot with that name agree that it should be transient. Otherwise the slot will be persistent.

```
(defmethod compute-effective-slot-definition :around
           ((class persistent-metalevel-class)
            slot-name
            direct-slot-definitions)
  ;; Let default system do its work first:
  (let ((slotd (call-next-method)))
    (setf (slot-value slotd 'transientp)
      (every #'slot-definition-transient-p direct-slot-definitions))
    slotd))
```

This example also illustrates how *class* metaobject class incompatibilities discussed in section 3.5.1.1 may introduce subtle problems: our persistence example was written to use `persistent-standard-direct-slot-definitions` for the slots of `persistent-metalevel-class`. All its *slot* metaobjects therefore have a method `slot-definition-transient-p` defined for them. If all classes involved in the inheritance used our metaobject class, the code above would therefore work. If, on the other hand, some of the supers were not using `:metaclass persistent-metalevel-class`, some `direct-slot-definitions` would not have `slot-definition-transient-p` defined on them and the code would fail. To ensure compatibility, we would have to define a default method on `slot-definition-transient-p` that returned `nil`.

We have one more problem to solve in the context of our persistent object system: Whenever a class is defined or redefined to change the number of slots, we must create or modify a corresponding piece of database schema. This can happen through changes to the class itself or through modifications of one of its superclasses. We can handle this conveniently by using the 'chokepoint' introduced in this section:

```
(defmethod finalize-inheritance :after
  ((class persistent-metalevel-class))
  (maintain-schema class))

(defmethod maintain-schema ((class persistent-metalevel-class))
  (if (schema-exists-p class)
    (rework-database-schema class)
    (progn
      (create-schema class)
      (store-class-structure class))))
```

This concludes our look at the creation and redefinition of classes. A full treatment of method definition and invocation would overload this introductory text, but we include the protocol outlines in appendix B.

3.5.2 Slot Access

The last piece of CLOS dynamics we will consider here is the setting and retrieving of slots. When making modifications in this area, the implementor should keep a small checklist of issues in mind:

- The different built-in CLOS slot allocations must be considered (e.g. `:instance` vs. `:class` allocation).
- There is a group of slot access related built-in generic functions that must be kept synchronized: Changes to one could require changes in the other. We will point to examples below.

All the 'official', programmer-level slot traffic goes through the `slot-value` function. This will not generally be true for code generated automatically for reader or writer methods. The entry point for such code is the generic function `slot-value-using-class` and its `setf` dual which are the main point for slot access modifications:

```
slot-value-using-class <class-metaobject>
                       <instance>
                       <effective-slot-definition-metaobject>
(setf slot-value-using-class) <new-value>
                              <class-metaobject>
                              <instance>
                              <effective-slot-definition-metaobject>
```

Figure 3.9 shows the protocol for accessing slots.

SLOT-VALUE-USING-CLASS ⟵ **Programmatic entry**
1 Check for existence of slot
`(slot-exists-p, slot-missing)`
2 Check for slot being bound
`(slot-boundp-using-class, slot-unbound)`
3 Retrieve the value

Figure 3.9: Summary of the Slot Reading Protocol

Apart from the generic functions listed in the figure, `slot-makunbound-using-class` should be considered if changes are made to the slot access process.

It is an error to attempt access to a non-existent slot. The Metaobject Protocol allows metalevel programmers to control what happens when this condition is encountered. That enables the programmer to react in a way that makes sense in his modified CLOS context. This control is exercised by defining methods on:

```
slot-missing <class-metaobject> <instance> <slot-name> <operation>
             &optional new-value
```

The `operation` parameter is one of the symbols `slot-value`, `setf`, `slot-bound` or `slot-makunbound`. These can be used to provide a helpful error message.

We need to intercept slot access for our persistent objects to work correctly. The main problem is that we must fault to the database if the object is persistent and not currently cached. In all other cases, we will defer to the built-in way of accessing slots[6].

This brings up a subtle problem that exemplifies the potential dangers of metalevel programming: recall that we record with each instance whether it is persistent and whether it is cached. We did this by causing persistent classes to inherit from `persistence-root-class`, which adds the slots `persistent?` and `cached?`. In order to find out whether an instance is cached or persistent, we therefore need to perform a slot access. Since we must do this to accomplish slot access in the first place, there will be infinite recursion whenever a slot is read, unless we take special precautions. We take care of this in the following code for reading a slot for persistent classes[7]:

```
(defmethod slot-value-using-class :around
  ((class persistent-metalevel-class)
   object
   (slotd persistent-standard-effective-slot-definition))
  (let (
        (slot-name (slot-definition-name slotd))
        (persistent?-slotd
         (find-if #'(lambda (slotd)
                      (eq (slot-definition-name slotd) 'persistent?))
                  (class-slots class)))
        (cached?-slotd
         (find-if #'(lambda (slotd)
                      (eq (slot-definition-name slotd) 'cached?))
                  (class-slots class))))
    (if (and (not (eq slot-name 'persistent?))
             (not (eq slot-name 'cached?))
             (slot-value-using-class class object persistent?-slotd)
             (not (slot-value-using-class class object cached?-slotd))
             (not (slot-definition-transient-p slotd)))
        (slot-value-from-database class slotd)
        (call-next-method))))
```

This concludes our summary of the Metaobject Protocol dynamics. We have seen that each protocol attempts to specify just enough detail about some piece of the CLOS operation to allow controlled modifications to be made. We have covered the process around creating and initializing new classes and the access to slots. Let us now move on to putting the approach into perspective with earlier work.

3.6 Related Work

The concept of making languages extensible concentrated initially on syntactic extension and the creation of new types [9]. Opening languages up for deep semantic changes is a more recent development. This requires the kind of architectural considerations introduced in this chapter.

The idea of making seemingly fundamental components of systems in reality be elements of a meta-level 'world' has been explored in various earlier systems.

Like CLOS, Smalltalk [10] includes the notion of metaclasses. But the concept, though equal in name, is quite different in the two languages: Each Smalltalk class is an instance of exactly one metaclass which in turn may only have that one class as its instance. A class thereby acts like 'regular', program-level objects in the sense that it responds to messages whose effects are determined by its metaclass. In particular, the metaclass controls the initialization of class variables and also manufactures the class' instances. But in contrast to CLOS, the programmer cannot modify metaclasses and use object-oriented programming at the metalevel to produce special effects.

ObjVlisp [11], which is very similar to CLOS [12], has worked on introducing a full metalevel class mechanism into Smalltalk-80 [13]. This has led to a kind of 'metaclass workbench' called *Classtalk* which helps with the construction of metaclass libraries and provides a metaclass browser.

An interesting angle to metalevel architectures is added by [14] which shows how the principle can be used in the construction of operating systems.

There is a rapidly accumulating body of literature about CLOS and its uses. The first, second and third "CLOS Users and Implementors Workshops" of 1988-1990 are good sources for information on a wide spectrum of CLOS aspects. Another report on the use of the Metaobject Protocol can be found in chapter five of this volume.

3.7 Conclusion

This chapter has attempted to introduce the CLOS programmer to the world that lies beyond the confines of the language proper. This world is defined and controlled by the Metaobject Protocol which makes the mechanisms for changing CLOS part of the language definition and thereby renders it portable.

We have introduced the basic notions of this 'metalevel world', giving the reader enough understanding to appreciate the concepts and to read the somewhat more formal specification for more in-depth information.

We believe that the tendency of making systems open should extend beyond areas like networking to the realm of language implementation, operating systems and databases. The CLOS Metaobject Protocol approach is an important step in this direction. Experience during its development has shown that it is difficult to find the correct balance between standardized degrees of freedom and the needs for optimization, between openness and safety, between flexibility and portability. Writing modular systems is more difficult initially than building monoliths. Making systems be open and portable is an additional dimension which requires additional care and sophistication. The payoff, however, is worth the investment because the system covers much more ground than it could with more conventional approaches.

A word of caution is in order at this point. Metalevel programming is still systems programming. Increased power bears with it additional dangers. Research is needed to understand which design rules and conventions can be added to the object-oriented programming style to introduce the necessary measure of safety. We have introduced the rules that were developed for controlling the use of the Metaobject Protocol. More experience is needed to find out whether these rules are necessary and sufficient. More probing still is needed to understand whether they have any universal applicability.

Designing protocols is another area that needs further investigation. The current specification of the Metaobject Protocol is only slightly more formal than our presentation in this chapter. Are there good formal ways of specifying behavior at the right level of detail? Are there indeed formal or informal ways of finding the correct level of detail in the first place? What definitely has to be specified to ensure portability of modifications and what must be left open to allow for optimizations? Understanding the nature of protocol design would go a long way towards making the idea of the CLOS design approach applicable to systems other than languages, a goal that seems intriguing after seeing the CLOS Metaobject Protocol as a datapoint.

3.8 Acknowledgements

This work benefited greatly from detailed and insightful suggestions by Daniel G. Bobrow and Gregor Kiczales. Robin Jeffries contributed through comments on earlier drafts of this chapter.

3.9 Appendix A: Rules for Metalevel Extensions

Different rules apply for implementors of the system and programmers wishing to create portable code which manipulates the metalevel. We do not address restrictions for implementors here but concentrate on the ones applying to portable programs. The following rules all have the same underlying reason: To ensure that new behavior does not modify *existing* system behavior that is relied upon by others:

- For a metalevel program to be portable it must not redefine existing metaobject classes, generic functions, methods or method combinations which are explicitly specified by the MOP.

  In syntactic terms this implies that every new metalevel method must have at least one specializer in its parameters which is not one of the built-in metaobject classes. This means that writing a :before, :after or :around method which specializes only on existing metaobject classes can render a program non-portable. Violating this rule could inadvertently destroy a method provided by the system, or it could cause unexpected side effects for programs using the default implementation. The programmer must produce his own metaobject class and specialize on it.

  Allowing the destructive modification of the existing stock of behavior could also lead to a kind of race condition in which two programs make a modification to the same piece of behavior. The order in which the programs are loaded would then determine the final behavior, which is unacceptable.

- Unless explicitly forbidden by the underlying generic function, it is always legal to *extend* the behavior of an existing method by writing a new one which specializes to relevant subclasses as explained above. But the arrangement must ensure that the original, less specific method will be called. For standard CLOS this means that the new method must be a :before or :after method, or that it is a primary or :around method which calls `call-next-method`. This ensures that any new behavior is *added* to the default behavior, as opposed to replacing it.

- Only if a generic function explicitly allows it, may methods be overridden, that is replaced completely by primary or :around methods that do not use `call-next-method` in their body.

  Note that MOP generic functions often come in 'groups' which must be kept consistent. When overriding one, consistency with the others must be ensured. One example is the group `add-dependent`, `remove-dependent` and `map-dependent`. These groupings are not always explicitly defined in the MOP.

3.10 Appendix B: Protocol Overviews

In order to make the MOP specification easier to follow we include here the full summaries of various protocols that were shortened in the text or are not covered there at all. We explained in section 3.5.1.1 how these figures are to be read.

Class Definition Protocol:
DEFCLASS
1 Syntax error checking
2 Canonicalize information
3 Obtain *class* metaobject
  (`ensure-class`, ⟵ **Programmatic entry**
   `ensure-class-using-class`) ⟵ **Programmatic entry**
  3.1 Find or make instance of proper *class* metaobject class
    (`make-instance`, **the** `:metaclass` **option**) ⟵ **Programmatic entry**
  3.2 (Re)initialize the *class* metaobject
    (`(re)initialize-instance`)
    3.2.1 Default unsupplied keyword arguments/error checking
    3.2.2 Check compatibility with superclasses
      (`validate-superclass`)
    3.2.3 Associate superclasses with this new class metaobject
    3.2.4 Determine proper *slot-definition* metaobject class
      (`direct-slot-definition-class`)
    3.2.5 Create and initialize the *slot-definition* metaobjects
      (`make-instance, initialize-instance`)
    3.2.6 Associate them with this new class metaobject
    3.2.7 Check `default-initargs`
    3.2.8 Maintain the 'subclasses' lists of superclasses
      (`add-direct-subclass`,`remove-direct-subclass`)
    3.2.9 Initiate inheritance finalization, if appropriate
      (`finalize-inheritance`)
    3.2.10 Create reader/writer methods
    3.2.11 Associate them with this new class metaobject

Slot Reading Protocol:
SLOT-VALUE-USING-CLASS ⟵ **Programmatic entry**
1 Check for existence of slot
  (`slot-exists-p`, `slot-missing`)
2 Check for slot being bound
  (`slot-boundp-using-class`, `slot-unbound`)
3 Retrieve the value

Class Finalization Protocol:
FINALIZE-INHERITANCE ⟵ **Programmatic entry**

1 Compute the class precedence list
  (`compute-class-precedence-list`)
2 Resolve conflicts among inherited slots with the same name
  2.1 Determine proper effective slot definition metaobject class
    (`effective-slot-definition-class`)
  2.2 Create the effective slot definition metaobjects
    (`make-instance`)
  2.3 Initialize the effective slot definitions
    (`initialize-instance`, `compute-effective-slot-definition`)
  2.4 Associate them with the class metaobject
3 Enable/Disable slot access optimizations
  (`slot-definition-elide-access-method-p`)

Method Lookup Protocol:
Generic Function Call

1 Invoke the generic function's discriminating function
  1.1 Find out which methods are applicable for the given arguments
    (`compute-applicable-methods-using-classes`, `compute-applicable-methods`)
  1.2 Combine the methods into one piece of code
    (`compute-effective-method`)
  1.2 Run the combined methods
    (`method-function-applier`)

Method Definition Protocol:
DEFMETHOD

1 Syntax error checking

2 Obtain target *generic function* metaobject
  (`ensure-generic-function`, ⟵ **Programmatic entry**
  `ensure-generic-function-using-class`) ⟵ **Programmatic entry**

  2.1 Find or make instance of proper *generic function* metaobject class
    (`make-instance`, `:generic-function-class` **from** `defgeneric` **form**)

  2.2 (Re)initialize the *generic function* metaobject
    (`(re)initialize-instance`)

    2.2.1 Default unsupplied keyword arguments/error checking
    2.2.2 Check lambda list congruence with existing methods
    2.2.3 Check argument precedence order spec against lambda list
    2.2.4 (Re)define any old 'initial methods'
    2.2.5 Recompute the generic function's discriminating function
        (`compute-discriminating-function`)

3 Build method function
  (`make-method-lambda`)

4 Obtain *method* metaobject

  4.1 Make instance of proper *method* metaobject class
    (`make-instance`, `generic-function-method-class`)

  4.2 Initialize the *method* metaobject
    (`initialize-instance`)

    4.2.1 Default unsupplied keyword arguments/error checking

5 Add the method to the generic function
  (`add-method`)

  5.1 Add method to the generic function's method set
  5.2 Recompute the generic function's discriminating function
    (`compute-discriminating-function`)
  5.3 Update discriminating function
  5.4 Maintain mapping from specializers to methods
    (`add-direct-method`)

Bibliography

[1] International Organization for Standardization. Basic reference model for open systems interconnection, 1984.

[2] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proc. 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*. Computer Society Press, 1987.

[3] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[4] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. Technical Report 88-002R, X3J13 Standards Committee, 1988. (Also published in SIGPLAN Notices, Vol. 23, special issue, Sept. 1988, and in Guy Steele: Common Lisp, The Language, 2nd ed., Digital Press, 1990.).

[5] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley Publishing Company, 1989.

[6] James Bennett, John Dawes, and Reed Hastings. Cleaning CLOS applications with the MOP. In Gregor Kiczales, editor, *Proceedings of the Second CLOS Users and Implementors Workshop*, 1989.

[7] Andreas Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, Springer Verlag, 1988.

[8] Andreas Paepcke. PCLOS: A Critical Review. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1989.

[9] ECL programmer's manual. Center for Research in Computing Technology, Harvard University, TR-23-74, December 1974.

[10] Adele Goldberg and David Robinson. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, 1983.

[11] Pierre Cointe. Metaclasses are first class: The ObjVlisp model. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*. Association of Computing Machinery, 1987.

[12] P. Cointe and N. Graube. Programming with metaclasses in CLOS. In *Proceedings of the First CLOS Users and Implementors Workshop*, 1988.

[13] Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1989.

[14] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In *Proceedings of the European Conference on Object-Oriented Programming*, 1989.

Notes:

[1]The MOP is not part of the official CLOS standard at this time. Its current state is documented in part two of [3].

[2]CLOS was developed using a reference implementation (PCL) which was distributed, critiqued and improved many times before commercial implementations began to emerge.

[3]Notable exception: details about slots are kept in another kind of metaobject which is covered in section 3.4.2 but which is also accessed indirectly through *class* metaobjects.

[4]Recall that an *initarg* is a name that may be associated with a slot and that may later be used in calls to `make-instance` to specify an initial value for that slot.

[5]Interesting protocols we do not cover here include the definition process for generic functions and methods and the addition of methods to generic functions.

[6]This assumes that we make cached objects look like regular CLOS objects. This is actually a very useful way of dealing with caching.

[7]For efficiency, the `persistent?` and `cached?` *slot definition* metaobjects should not be searched for during every slot access as is done by the `find-if` calls in the example. They would be cached in a real system.

Biography:

Andreas Paepcke has been with Hewlett-Packard Laboratories since 1982, working on a wide range of projects, including an infrared network for terminals and workstations, the integration of telephone service into workstation environments, transparent persistence of CLOS objects on a variety of databases and access to information services through object-oriented views. Mr. Paepcke received his BS and MS degrees from Harvard University and his Ph.D. in Computer Science from the University of Karlsruhe, Germany.