Common Lisp Object System

by

Linda G. DeMichiel and Richard P. Gabriel

Lucid, Inc.
707 Laurel Street
Menlo Park, California 94025
(415)329-8400
LGD@SAIL.STANFORD.EDU RPG@SAIL.STANFORD.EDU

with Major Contributions by

Daniel Bobrow, Gregor Kiczales, and David Moon

Abstract

The Common Lisp Object System is an object-oriented system that is based on the concepts of generic functions, multiple inheritance, and method combination. All objects in the Object System are instances of *classes* that form an extension to the Common Lisp type system. The Common Lisp Object System is based on a meta-object protocol that renders it possible to alter the fundamental structure of the Object System itself. The Common Lisp Object System has been proposed as a standard for ANSI Common Lisp and has been tentatively endorsed by X3J13.

Common Lisp Object System

by

Linda G. DeMichiel and Richard P. Gabriel

with Major Contributions by

Daniel Bobrow, Gregor Kiczales, and David Moon

1. History of the Common Lisp Object System

The Common Lisp Object System is an object-oriented programming paradigm designed for Common Lisp. Over a period of eight months a group of five people, one from Symbolics and two each from Lucid and Xerox, took the best ideas from CommonLoops and Flavors, and combined them into a new object-oriented paradigm for Common Lisp.

This combination is not simply a union: It is a new paradigm that is similar in its outward appearances to CommonLoops and Flavors, and it has been given a firmer underlying semantic basis.

The Common Lisp Object System has been proposed as a standard for ANSI Common Lisp and has been tentatively endorsed by X3J13.

2. The Common Lisp Object System View of Object-Oriented Programming

2.1 *What the Common Lisp Object System Is*

The Common Lisp Object System is an object-oriented system that is based on the concepts of generic functions, multiple inheritance, method combination, and meta-objects. All objects in the Object System are instances of *classes* that form an extension to the Common Lisp type system.

A *generic function* is a function whose behavior depends on the classes or identities of the arguments supplied to it. The *methods* associated with the generic function define the class-specific behavior and operations of the generic function.

The Common Lisp Object System supports multiple inheritance in a similar manner to CommonLoops and Flavors. Inheritance of methods and structure is based on a linearization of the class graph.

§ 2 The Common Lisp Object System View of Object-Oriented Programming

In the Common Lisp Object System classes and generic functions are first-class objects with no intrinsic names. Thus it is possible and useful to create and manipulate anonymous classes and generic functions.

The Common Lisp Object System supports a mechanism for method combination that is both more powerful than that provided by CommonLoops and simpler than that provided by Flavors.

The Common Lisp Object System is founded on a meta-object system which is capable of supporting other object-oriented paradigms. In fact, the Object System itself can be implemented within this meta-object system.

2.2 *What the Common Lisp Object System Is Not*

The Object System is not a single inheritance language. As such it is much more like Flavors than like Smalltalk-80.

The Object System is not a message-passing language. If the behavior of generic functions depended on the class of exactly one argument, where that argument was distinguished positionally, then it would be isomorphic to a message-passing language. But the behavior of a generic function can depend on the classes of several arguments simultaneously.

The Object System does not attempt to solve problems of encapsulation or protection. The inherited structure of a class depends on the names of internal parts of the classes from which it inherits. The Object System does not support subtractive inheritance. Within Common Lisp there is a primitive module system that can be used to help create separate internal namespaces.

3. Goals of the Common Lisp Object System.

The primary goals that the design of the Common Lisp Object System attempts to meet are the following:

1. To fit into Common Lisp in a natural way, both in terms of using functional notation and in terms of extending the Common Lisp type system.

2. To provide a smooth growth path and easy transition for current users of Flavors and CommonLoops.

3. To provide a layered approach in which each layer provides functionality at an appropriate level according to the sophistication of user needs.

4. To provide both a language to form the basis of a powerful programming environment as well as a platform for the efficient delivery of applications written within the Object System.

4. Design Criteria

The design of the Common Lisp Object System was founded on several design principles.

1. Use a set of levels to separate programming language concerns from each other. The first level provides a system of object-oriented programming that meets the needs of most serious users, with a syntax that is crisp and understandable. The second level provides a functional interface into the heart of the Object System for the programmer who is writing very complex software or a programming environment; this level is completely consistent with the first level. The first level is written in terms of this second level. The third level provides the tools for the programmer writing his own object-oriented language. It allows access to the primitive objects and operators of the Common Lisp Object System. It is this level in which the Object System itself can be written.

2. Make as many things as possible within the Object System first-class. Therefore, classes and generic functions are first-class objects, whereas generic functions, for example, are first-class objects in neither Flavors nor CommonLoops.

3. Provide a unified language and syntax. It is tempting, when designing a programming language, to invent additional languages within the primary programming language, where each additional language is suitable for some particular aspect of the overall language. In Common Lisp, for example, the **format** function defines a language for displaying text, but this language is not Lisp nor is it like Lisp.

4. Make the specification of the language as precise as possible. Though this may seem an obvious design criterion, this criterion was not followed during the design of Common Lisp.

5. Be willing to trade off complex behavior for conceptual and expository simplicity. Though it may appear that the effect of this criterion on the Object System was subtle, it led to radical simplifications to proposed functionality.

5. Classes

A *class* is an object that determines the structure and behavior of a set of other objects, which are called its *instances*. It is an important feature of the Common Lisp Object System that every Common Lisp object is an instance of a class. The class of an object determines the set of operations that can be performed on that object.

Classes are represented by first-class objects that are themselves instances of classes. The class of the class of an object is termed the *metaclass* of that object. The metaclass determines the form of inheritance used by the classes that are its instances and the representation of the instances of those classes. The Common Lisp Object System provides a default metaclass that is appropriate for most programs. The Common Lisp Object System meta-object protocol allows for defining and using new metaclasses.

A class can inherit structure and behavior from other classes. A class whose definition refers to other classes for the purpose of inheriting from them is said to be a *subclass* of each of those classes. The classes that are designated for purposes of inheritance are said to be *superclasses* of the inheriting class. The inheritance relationship is transitive.

Classes are organized into a *directed acyclic graph*. There is a distinguished class named **t**. The class **t** is a superclass of every other class.

The Common Lisp Object System maps the Common Lisp type space into the space of classes. Many but not all of the predefined Common Lisp type specifiers have a class associated with them that has the same name as the type. For example, an array is of type **array** and of class **array**. Every class has a corresponding type with the same name as the class.

A class that corresponds to a predefined Common Lisp type is called a *standard type class*. Each standard type class has the class **standard-type-class** as a metaclass. Users can write methods that discriminate on any primitive Common Lisp type that has a corresponding class. However, it is not allowed to make an instance of a standard type class with **make-instance** or to include a standard type class as a superclass of a class.

5.1 *Defining Classes*

The macro **defclass** is used to define a new class. This macro is on the first of the three levels. The function **make-instance** when applied to an appropriate metaclass provides the same functionality on the second level.

The definition of a class consists of the following: its name, a list of its direct superclasses, a set of slot specifiers, and a set of class options.

The direct superclasses of a class are those classes from which the new class inherits structure and behavior. When a class is defined, the order in which its direct superclasses are mentioned in the defining form is important. Each class has a *local precedence order*, which is a list consisting of the class followed by its direct superclasses in the order mentioned in the **defclass** form.

Each slot specifier includes the name of the slot and zero or more slot options. A slot option pertains only to a single slot. The slot options of the **defclass** form allow for the following: providing a default initial value form for the slot; requesting that methods for appropriately named generic functions be automatically generated for reading or writing the slot; controlling whether one copy of a given slot is shared by all instances or whether each instance is to have its own copy of that slot; and specifying the type of the slot contents.

Each class option pertains to the class as a whole. The available class options allow for the following: requesting that methods for appropriately named generic functions be automatically generated for reading or writing all slots defined by the new class; requesting that a constructor function be automatically generated for making instances of the class; and indicating that the instances of the class are to have a metaclass other than the default.

For example, the following two classes define a representation of a point in space. The class **x-y-position** is a subclass of the class **position**:

```
(defclass position () ())

(defclass x-y-position (position)
    ((x :initform 0)
     (y :initform 0))
  (:accessor-prefix position-))
```

The class **position** is useful if we desire to create other sorts of representations for spatial positions. The x- and y-coordinates are initialized to 0 in all instances unless explicit

values are supplied for them. To refer to the x-coordinate of an instance of the class **x-y-position**, **position**, one writes:

```
(position-x position)
```

To alter the x-coordinate of that instance, one writes:

```
(setf (position-x position) new-x)
```

5.2 *Slots*

Classes and class instances have named slots. The name of a slot is a symbol that could be used as a Common Lisp variable name.

There are two kinds of slots: slots that are local to an individual instance and slots that are shared by all instances of a given class. The **:allocation** slot option to **defclass** controls the kind of slot that is defined.

In general, slots are inherited by subclasses. That is, a slot defined by a class is also a slot implicitly defined by any subclass of that class unless the subclass explicitly shadows the slot definition. A class can also shadow some of the slot options declared in the **defclass** form of one of its superclasses by providing its own description for that slot.

Slots can be accessed in two ways: by use of generic functions defined by the **defclass** form and by use of the primitive function **slot-value**.

The **defclass** syntax allows for generating methods to read and write slots. If a slot *accessor* is requested, a method is automatically generated for reading the value of the slot, and a **setf** method is also generated to write the value of the slot. If a slot *reader* is requested, a method is automatically generated for reading the value of the slot, but no **setf** method for it is generated. Readers and accessors can be requested for individual slots or for all slots. Reader and accessor methods are added to the appropriate generic functions. It is possible to modify the behavior of these generic functions by writing methods for them.

The function **slot-value** can be used with any of the slot names specified in the **defclass** form to access a specific slot in an object of the given class. Readers and accessors are implemented by using **slot-value**.

Sometimes it is convenient to access slots from within the body of a method or a function. The macro **with-slots** is provided for use in setting up a lexical environment in

which certain slots are lexically available. It is also possible to specify whether the macro **with-slots** is to use the accessors or the function **slot-value** to access slots.

5.3 *Class Precedence*

Each class has a *class precedence list*. The class precedence list is a total ordering on the set of the given class and its superclasses for purposes of inheritance. The total ordering is expressed as a list ordered from most specific to least specific.

The class precedence list is used in several ways. In general, more specific classes can *shadow*, or override, features that would otherwise be inherited from less specific classes. The method selection and combination process uses the class precedence list to order methods from most specific to least specific.

The class precedence list is always consistent with the local precedence order of each class in the list. The classes in each local precedence order appear within the class precedence list in the same order. If the local precedence orders are inconsistent with each other, no class precedence list can be constructed, and an error will be signaled.

6. Generic Functions

The class-specific operations of the Common Lisp Object System are provided by generic functions and methods.

A *generic function* is a function whose behavior depends on the classes or identities of the arguments supplied to it.

Like an ordinary Lisp function, a generic function takes arguments, performs a series of operations, and returns values. An ordinary function has a single body of code that is always executed when the function is called. A generic function might perform a different series of operations and combine the results of the operations in different ways, depending on the class or identity of one or more of its arguments. Like other operations in Lisp, invoking a generic function requires examining the classes of its arguments at runtime, and, though a compiler might be able to optimize away some of this runtime typing, method selection and combination is a runtime affair.

The operations of a generic function are defined by its methods. The behavior of the generic function results from which methods are selected for execution, the order in which the selected methods are called, and how their values are combined to produce the value or values of the generic function.

Thus, unlike an ordinary function, a generic function has a distributed definition, corresponding to the definition of its methods. The definition of a generic function is found in a set of **defmethod** forms, possibly along with a **defgeneric-options** form that provides information about the properties of the generic function as a whole. Evaluating these forms produces a generic function object.

Generic functions are first-class objects in the Common Lisp Object System. They can be used in the same ways that ordinary functions can be used in Common Lisp. A generic function is a true function that can be passed as an argument, used as the first argument to **funcall** and **apply**, and stored in the function cell of a symbol. Ordinary functions and generic functions are called with identical syntax.

6.1 *Generic Function Objects*

A generic function object comprises a set of methods, a lambda-list, a method combination type, and other information.

The methods associated with the generic function define the class-specific behavior and operations of the generic function. Thus, generic functions are objects that may be *specialized* by the definition of methods to provide class-specific operations.

The lambda-list specifies the arguments to the generic function. It is an ordinary function lambda-list with these exceptions: No **&aux** variables are allowed; optional and keyword arguments may not have default initial value forms nor use supplied-p parameters. The generic function passes to its methods all the argument values passed to it, and only these; default values are not supported.

The method combination type determines the form of method combination that is used with the generic function. The *method combination* facility controls the selection of methods, the order in which they are run, and the values that are returned by the generic function. The Common Lisp Object System offers a default method combination type that is appropriate for most user programs. The Common Lisp Object System also provides a facility for declaring new types of method combination for programs that require them.

The generic function object also contains information about the argument precedence order (the order in which arguments to the generic function are tested for specificity when selecting executable methods), the class of the generic function, and the class of the methods of the generic function. While the Common Lisp Object System provides default

classes for all generic function, method, and class objects, the programmer may choose to implement any or all of these using classes of his own definition.

6.2 *Defining Generic Functions*

Generic functions are defined by means of the **defgeneric-options** and **defmethod** macros, on the first level. On the second level, the function **make-instance** is used to create a generic function.

If a **defgeneric-options** form is evaluated and a generic function of the given name does not already exist, a new generic function object is created. This generic function object is a generic function with no methods. The **defgeneric-options** macro may be used to specify properties of the generic function as a whole—sometimes referred to as the "contract" of the generic function. These properties include the argument precedence order, the method combination type, the class of the generic function, and the class of the methods of the generic function.

When a new **defgeneric-options** form is evaluated and a generic function of the given name already exists, the existing generic function object is modified. This does not modify any of the methods associated with the generic function.

The **defmethod** form is used to define a method. If no generic function of the given name already exists, however, it automatically creates a generic function with default values for the argument precedence order, the generic function class, the method class, and the method combination type. The lambda-list of the generic function is congruent with the lambda-list of the new method. In general, two lambda-lists are congruent if they have the same number of required parameters, the same number of optional parameters, and the same treatment of **&allow-other-keys**.

When a **defmethod** form is evaluated and a generic function of the given name already exists, the existing generic function object is modified to contain the new method. The lambda-list of the new method must be congruent with the lambda-list of the generic function.

7. Methods

The class-specific operations provided by generic functions are themselves defined and implemented by *methods*. A generic function can have several methods associated with it, and the class or identity of each argument to the generic function indicates which method or methods to use.

7.1 *Method Objects*

A method object contains a method function, an ordered set of *parameter specializers* that specify when the given method is applicable, and an ordered set of *qualifiers* that are used by the method combination facility to distinguish among methods. Each required formal parameter of each method has an associated parameter specializer, and the method is expected to be invoked only on arguments that satisfy its parameter specializers.

A parameter specializer is either a class or a list of the form (**quote** *object*).

A method can be selected for a set of arguments when each required argument satisfies its corresponding parameter specializer. An argument satisfies a parameter specializer if either of the following conditions holds:

1. The parameter specializer is a class and the argument is an instance of that class or an instance of any subclass of that class

2. The parameter specializer is (**quote** *object*) and the argument is **eql** to *object*.

A method all of whose parameter specializers are **t** is a *default method*; it is always part of the generic function but often shadowed by a more specific method.

Methods can have *qualifiers*, which give the method combination procedure a way to distinguish between methods. A method that has one or more qualifiers is called a *qualified* method. A method with no qualifiers is called an *unqualified method*. A qualifier is any object other than a cons, that is, any non-**nil** atom. By convention, qualifiers are usually keyword symbols.

In standard method combination, unqualified methods are also termed *primary* methods and qualified methods have a single qualifier that is either **:around**, **:before**, or **:after**.

7.2 *Defining Methods*

The macro **defmethod** is used to create a method object. A **defmethod** form contains the code that is to be run when the arguments to the generic function cause the method that it defines to be selected. If a **defmethod** form is evaluated and a method object corresponding to the given generic function name, parameter specializers, and qualifiers already exists, the new definition replaces the old.

Each method definition contains a *specialized lambda-list*, which specifies when that method can be selected. A specialized lambda-list is like an ordinary lambda-list except

that a *parameter specifier* may occur instead of the name of a parameter. A parameter specifier is a list consisting of a variable name and a parameter specializer name. Every parameter specializer name is a Common Lisp type specifier, but the only Common Lisp type specifiers that are parameter specializers names are type specifier symbols with corresponding classes and type specifier lists of the form **(quote** *object*). The form **(quote** *object*) is equivalent to the type specifier **(member** *object*).

Only required parameters can be specialized, and each required parameter must be a parameter specifier. For notational simplicity, if some required parameter in a specialized lambda-list is simply a variable name, the corresponding parameter specifier is taken to be (*variable-name* **t**).

A future extension to the Object System might allow optional and keyword parameters to be specialized.

A method definition may optionally specify one or more method qualifiers. A method qualifier is a non-**nil** atom that is used to identify the role of the method to the method combination type used by the generic function of which it is part.

Generic functions can be used to implement a layer of abstraction on top of a set of classes. For example, the class **x-y-position** can be viewed as containing information in polar coordinates.

Two methods are defined, called **position-rho** and **position-theta**, that calculate the ρ and θ coordinates given an instance of the class **x-y-position**.

```
(defmethod position-rho ((pos x-y-position))
   (let ((x (position-x pos))
         (y (position-y pos)))
     (sqrt (+ (* x x) (* y y)))))

(defmethod position-theta ((pos x-y-position))
   (atan (position-y pos) (position-x pos)))
```

It is also possible to write methods that update the 'virtual slots' **position-rho** and **position-theta**:

```
(defmethod-setf position-rho ((pos x-y-position)) (rho)
   (let* ((r (position-rho pos))
          (ratio (/ rho r)))
     (setf (position-x pos) (* ratio (position-x pos)))
     (setf (position-y pos) (* ratio (position-y pos)))))
```

```
(defmethod-setf position-theta ((pos x-y-position)) (theta)
   (let ((rho (position-rho pos)))
     (setf (position-x pos) (* rho (cos theta)))
     (setf (position-y pos) (* rho (sin theta)))))
```

To update the ρ-coordinate one writes:

```
(setf (position-rho pos) new-rho)
```

which is precisely the same syntax that would be used if the positions were explicitly stored as polar coordinates.

8. Class Redefinition

The Common Lisp Object System provides a powerful class-redefinition facility.

When a **defclass** form is evaluated and a class with the given name already exists, the existing class is redefined. Redefining a class modifies the existing class object to reflect the new class definition.

When a class is redefined, changes are propagated to instances of it and to instances of any of its subclasses. The updating of an instance whose class has been redefined (or any of whose superclasses have been redefined) occurs at an implementation-dependent time, but will usually be upon the next access to that instance or the next time that a generic function is applied to that instance. Updating an instance does not change its identity as defined by the **eq** function. The updating process may change the slots of that particular instance, but it does not create a new instance.

Users may define methods on the generic function **class-changed** to control the class redefinition process. The generic function **class-changed** is invoked automatically by the system after **defclass** has been used to redefine an existing class.

For example, suppose it becomes apparent that the application that requires representing positions uses polar coordinates more than it uses rectangular coordinates. It might make sense to define a subclass of **position** that uses polar coordinates:

```
(defclass rho-theta-position (position)
    ((rho :initform 0)
     (theta :initform 0))
  (:accessor-prefix position-))
```

The instances of **x-y-position** can be automatically updated by defining a **class-changed** method:

```
(defmethod class-changed ((old x-y-position)
                          (new rho-theta-position))
  ;; Copy the position information from old to new to make new
  ;; be a rho-theta-position at the same position as old.
  (let ((x (position-x old))
        (y (position-y old)))
    (setf (position-rho new) (sqrt (+ (* x x) (* y y)))
          (position-theta new) (atan y x))))
```

At this point we can change an instance of the class **x-y-position**, **p1**, to be an instance of **rho-theta-position** using **change-class**:

```
(change-class p1 'rho-theta-position)
```

9. Inheritance

Inheritance is the key to program modularity within the Common Lisp Object System. A typical object-oriented program consists of several classes, each of which defines some aspect of behavior. New classes are defined by including the appropriate classes as superclasses, thus gathering desired aspects of behavior into one class.

9.1 *Inheritance of Slots and Slot Description*

In general, slot descriptions are inherited by subclasses; that is, slots defined by a class are usually slots implicitly defined by any subclass of that class unless the subclass explicitly shadows the slot definition. A class can also shadow some of the slot options declared in the **defclass** form of one of its superclasses by providing its own description for that slot.

In the simplest case, only one class in the class precedence list provides a slot description with a given slot name. If it is a local slot, then each instance of the class and all of its subclasses allocate storage for it. If it is a shared slot, the storage for the slot is allocated by the class that provided the slot description, and the single slot is accessible in instances of that class and all of its subclasses.

More than one class in the class precedence list can provide a slot description with a given slot name. In such cases, at most one slot with a given name is accessible in any

instance, and the characteristics of that slot involve some combination of the several slot descriptions.

Methods that access slots know only the name of the slot and the type of the slot's value. Suppose a superclass provides a method that expects to access a shared slot of a given name and a subclass provides a local description of a local slot with the same name. If the method provided by the superclass is used on an instance of the subclass, the method accesses the local slot.

9.2 *Inheritance of Methods*

A subclass inherits methods in the sense that any method applicable to an instance of a class is also applicable to instances of any subclass of that class (all other arguments to the method being the same).

The inheritance of methods acts the same way regardless of whether the method was created by using **defmethod** or by using one of the **defclass** options that cause methods to be generated automatically.

10. Class Precedence List

The class precedence list is a linearization of the subgraph consisting of a class C and its superclasses. The **defclass** form for a class provides a total ordering on that class and its direct superclasses. This ordering is called the *local precedence order*. It is an ordered list of the class and its direct superclasses. A class precedes its direct superclasses, and a direct superclass precedes all other direct superclasses specified to its right in the superclasses list of the **defclass** form. For every class in the set of C and its superclasses, we can gather the specific relations of this form into a set, called R.

R may or may not generate a partial ordering, depending on whether the relations are consistent; we assume they are consistent and that R generates a partial ordering. This partial ordering is the transitive closure of R.

To compute the class precedence list at C, we topologically sort C and its superclasses with respect to the partial ordering generated by R. When the topological sort algorithm must select a class from a set of two or more classes, none of which are preceded by other classes with respect to R, the class selected is chosen deterministically. The rule that was chosen for this selection process is designied to keep chains of superclasses together in the class precedence list. That is, if C_1 is the unique superclass of C_2, C_2 will immediately precede C_1 in the class precedence list.

We require that an implementation of Common Lisp Object System signal an error if R is inconsistent, that is, if the class precedence list cannot be computed.

11. Method Combination

When a generic function is called with particular arguments, it must determine what code to execute. This code is termed the *effective method* for those arguments. The effective method can be one of the methods of the generic function or a combination of several of them.

Choosing the effective method involves the following decisions: which method or methods to call; the order in which to call these methods; which method to call when **call-next-method** is invoked; what value or values to return.

The effective method is determined by the following three steps:

1. Selecting the set of applicable methods;

2. Sorting the applicable methods by precedence order, putting the most specific method first;

3. Applying method combination to the sorted list of applicable methods, producing the effective method.

When the effective method has been determined, it is called with the same arguments that were passed to the generic function. Whatever values it returns are returned as the values of the generic function.

The Common Lisp Object System provides a default method combination type, *standard method combination*, that is designed to be simple, convenient, and powerful for most applications.

11.1 *Standard Method Combination*

Standard method combination is the default method combination type. Standard method combination recognizes four roles for methods, as determined by method qualifiers.

Primary methods define the main action of the effective method, while *auxiliary methods* modify that action in one of three ways. A primary method has no method qualifiers. The auxiliary methods are **:before**, **:after**, and **:around** methods.

The semantics of standard method combination are:

If there are any **:around** methods, the most specific **:around** method is called. Inside the body of an **:around** method, **call-next-method** can be used to immediately call the next method. When the next method returns, the **:around** method can execute more code. By convention, **:around** methods almost always use **call-next-method**.

If an **:around** method invokes **call-next-method**, the next most specific **:around** method is called, if one is applicable. If there are no **:around** methods or if **call-next-method** is called by the least specific **:around** method, the other methods are called as follows:

1. All the **:before** methods are called, in most specific first order. Their values are ignored.

2. The most specific primary method is called. Inside the body of a primary method, **call-next-method** may be used to pass control to the next most specific primary method. When that method returns, the first primary method can execute more code. If **call-next-method** is not used, only the most specific primary method is called.

3. All the **:after** methods are called in most specific last order. Their values are ignored.

If no **:around** methods were invoked, the most specific primary method supplies the value or values returned by the generic function. Otherwise, the value or values returned by the most specific primary method are those returned by the invocation of **call-next-method** in the least specific **:around** method.

An error is signaled if **call-next-method** is used in a **:before** or **:after** method or if **call-next-method** is used and there is no next method remaining.

In standard method combination, if there are any applicable methods at all, then there must be an applicable primary method. In cases where there are applicable methods, but no primary method, an error is signaled.

Standard method combination allows no more than one qualifier per method.

If only primary methods are used, standard method combination behaves like CommonLoops. If **call-next-method** is not used, only the most specific method is invoked; that is, more general methods are shadowed by more specific ones. If **call-next-method** is used, the effect is the same as **run-super** in CommonLoops.

If **call-next-method** is not used, standard method combination behaves like **:daemon** method combination of New Flavors, with **:around** methods playing the role of whoppers, except that the ability to reverse the order of the primary methods has been removed.

The use of method combination can be illustrated by the following example. Suppose we have a class called **general-window**, which is made up of a bitmap and a set of viewports.

```
(defclass general-window ()
   ((initialized :initform nil)
    (bitmap :type bitmap)
    (viewports :type list))
   (:accessor-prefix general-window-))
```

The viewports are stored as a list. We presume that it is desirable to make instances of general windows but to not create their bitmaps until they are actually needed. Thus we see that there is a flag, called **initialized**, that states whether the bitmap has been created. The **bitmap** and **viewport** slots are not initialized by default.

We now wish to create an announcement window that will be used for messages that must be brought to the user's attention. When a message is to be announced to the user, the announcement window is exposed, the message is moved into the bitmap for the announcement window, and finally the viewports are redisplayed.

```
(defclass announcement-window (general-window)
   ((contents :initform "" :type string))
   (:accessor-prefix announcement-window-))

(defmethod display :around (message (w general-window))
   (unless (general-window-initialized w)
    (setf (general-window-bitmap w) (make-bitmap))
    (setf (general-window-viewports w)
          (list (make-viewport (general-window-bitmap w))))
    (setf (general-window-initialized w) t)))

(defmethod display :before (message (w announcement-window))
   (expose-window w))

(defmethod display :after (message (w announcement-window))
   (redisplay-viewports w))

(defmethod display ((message string) (w announcement-window))
   (move-string-to-window message w))
```

To make an announcement, the generic function **display** is invoked on a string and an annoucement window. The **:around** method is always run first; if the bitmap has not been set up, this method takes care of it. The primary method for **display** simply moves the string (the announcement) to the window; the **:before method** exposes the window; and the **:after** method redisplays the viewports. When the window's bitmap is initialized, the sole viewport is made to be the entire bitmap. The order in which these methods are invoked is: 1. the **:around** method, 2. the **:before** method, 3. the primary method, and 4. the **:after** method.

11.2 *Defining Other Types of Method Combination*

The programmer can define new forms of method combination by using the **define-method-combination** macro.

There are two forms of **define-method-combination**. The short form is a simple facility for the cases that have been found to be most commonly needed. The long form is more powerful but more verbose. It resembles **defmacro** in that the body is an expression that computes a Lisp form, usually using backquote. Thus, arbitrary control structures can be implemented. The long form also allows arbitrary processing of method qualifiers.

12. Meta-Objects

The Common Lisp Object System provides the predefined meta-objects **standard-method** and **standard-generic-function**.

The class **standard-method** is the default class of methods defined by **defmethod** or **defmethod-setf**.

The class **standard-generic-function** is the default class of generic functions defined by **defmethod**, **defmethod-setf**, **defgeneric-options**, or **defgeneric-options-setf**.

The Common Lisp Object System also provides the standard method combination type, which is not implemented as a meta-object, but as a method.

13. References

Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, and David A. Moon, *Common Lisp Object System Specification*, X3J13 Document 87-002.

Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel, "CommonLoops: Merging Lisp and Object-Oriented Programming," ACM OOPSLA Conference, 1986.

Adelle Goldberg and David Robson, *Smalltalk-80: The Language and its Immplementation*, Addison-Wesley, Reading, Massachusetts, 1983.

Guy L. Steele, *Common Lisp: The Language*, Digital Press, 1984.

Reference Guide to Symbolics Common Lisp: Language Concepts, Symbolics Release 7 Document Set, 1986.