The Common Lisp Object System

by

Linda G. DeMichiel

Lucid, Inc.
707 Laurel Street
Menlo Park, California 94025

1. 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.

2. 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 twenty months, the Common Lisp Object System design group 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 major participants in this design effort were Daniel Bobrow and Gregor Kiczales from Xerox, David Moon and Sonya Keene from Symbolics, and Richard Gabriel and Linda DeMichiel from Lucid.

The Common Lisp Object System has been proposed as a standard for ANSI Common Lisp. In 1987, the X3J13 committee endorsed an earlier, but incomplete, version of the specification, stating that it would almost certainly be adopted as part of the Common Lisp standard, and encouraged implementors to proceed with trial implementations. In June 1988, the X3J13 committee accepted the Common Lisp Object System Programmer

Interface, as defined in Document 88-002R, for inclusion into the Common Lisp language being specified by X3J13. This paper is a report on the specification of the Common Lisp Object System Programmer Interface that was adopted by X3J13 in June 1988.

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

3.1 *What the Common Lisp Object System Is*

The Common Lisp Object System is an object-oriented system that is based on the concepts of classes, 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. 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 multiple inheritance in a similar manner to CommonLoops and Flavors. Inheritance of methods and structure is based on a linearization of the class graph.

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 that is capable of supporting other object-oriented paradigms. In fact, the Object System itself can be implemented within this meta-object system.

3.2 *What the Common Lisp Object System Is Not*

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 by its position, 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.

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

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

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.

4. 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. It is not necessary for a class to have any instances.

The definition of a class allows a set of *superclasses* to be designated as classes from which the given class can inherit structure and behavior. A class whose definition specifies such a set of superclasses is said to be a *subclass* of each of those classes. A class can be neither a direct nor indirect subclass of itself. Thus, classes are organized into a *directed acyclic graph*. There is a distinguished class named **t** that is a superclass of every other class. The inheritance relationship that is defined by the subclass and superclass relationships among classes is transitive.

Classes are first-class objects that are themselves instances of classes. The class of the class of an object is called the *metaclass* of that object. The class of a class determines the representation of the instances of that class. The existence of metaclasses indicates that the structure and behavior of the class system itself is controlled by classes. Generic functions and methods are also objects and therefore are also instances of classes.

The Common Lisp Object System integrates the space of classes and the Common Lisp type space. For every class there is a corresponding type with the same name as the class. Many of the predefined Common Lisp type specifiers have a corresponding class of the same name as the type.

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 certain predefined classes with **make-instance** or to designate certain classes as superclasses.

Programmer-defined classes are instances of the class named **standard-class**. Instances whose metaclass is **standard-class** are like Common Lisp structures: they have named slots, which contain values. When we say that the structure of an instance is determined by its class and that that class is an instance of **standard-class**, we mean that the number and names of the slots are determined by the class, and we also mean that the means of accessing and altering the contents of those slots are controlled by the class.

4.1 *Defining Classes*

The macro **defclass** is used to define a class.

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 **defclass** form defines a *local precedence order* on the class and those superclasses. The local precedence order is used in determining inheritance. It is represented as a list consisting of the class followed by its direct superclasses in the order that they are mentioned in the **defclass** form.

A slot specifier consists of the name of the slot and zero or more slot options. The slot options of the **defclass** form allow for the following: providing initialization arguments for use in controlling the process of instance creation and initialization; providing a default initial value form for the slot; specifying that methods for generic functions are to be created for reading or writing the slot; controlling whether the slot is to be shared by all instances of the class or whether each instance is to have its own copy of the slot; and specifying the expected type of the slot contents.

A class option pertains to the class as a whole. The available class options allow for the following: providing default values for initialization arguments; specifying that the instances of the class are to have a metaclass other than the default; and providing documentation for the class.

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 () ())
```

The class **position** is useful if we desire to create other sorts of representations for spatial positions.

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

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**, we write:

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

To change the x-coordinate of that instance to the value *new-x*, we write:

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

4.2 *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 individual instances and slots that are shared by all the instances of a class. The kind of slot that is created is determined by the **:allocation** slot option to **defclass**.

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 that slot definition is shadowed. A class can shadow some or all of the slot options declared in the **defclass** form of one of its superclasses by providing its own description for that slot.

The **defclass** syntax allows for requesting that methods to read and write slots be automatically generated. The **:reader** slot option specifies that a *reader* method is to be created for reading the value of the slot. The **:writer** slot option specifies that a *writer* method is to be created for writing the value of the slot. The **:accessor** slot option specifies that both reader and writer methods are to be created for accessing and setting the value of the slot. All such methods are added to the appropriate generic functions. It is possible to modify the behavior of these generic functions by writing methods for them.

Slots can also be accessed by using the primitive function **slot-value**. The function **slot-value** can be used with a slot name to access a specific slot in an object, whether or not methods have been specified to read or write that slot. The function **slot-value** is used to implement reader and writer methods.

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 as if they were variables. The macro **with-accessors** provides an analogous functionality: it sets up a lexical environment in which certain slots are lexically available through their accessors as if they were variables.

4.3 *Class Precedence*

A *class precedence list* is associated with every class. The class precedence list is used for purposes of determining inheritance. It is a total ordering on the set of the given class and its superclasses in which the most specific classes precede the least specific.

The construction of the class precedence list at a class proceeds by topologically sorting the set of that class and its superclasses under the relation that 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. Therefore, 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. Because a partial ordering may be embedded in several total orderings, there is a rule used to select which total ordering to use. That rule has two main effects: simple chains of superclasses are preserved, and classes in relatively separated subgraphs are adjacent.

If the local precedence orders are inconsistent with each other, no class precedence list can be constructed, and an error will be signaled.

5. 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 its arguments. The operations of a generic function are defined by its *methods*. Thus, generic functions are objects that may be *specialized* by the definition of methods to provide class-specific operations. 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 can have a distributed definition, corresponding to the definitions of its methods. The definition of a generic function is found in a set of **defmethod** forms, possibly along with a **defgeneric** form that provides information about the properties of the generic function as a whole.

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

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. Default values are not supported by the generic function, but rather by individual methods.

The method combination type determines the form of method combination that is used with the generic function. The *method combination* facility determines which methods are available for execution, the order in which they are run, and the values that are returned by the generic function. The Common Lisp Object System provides a default method combination type that is appropriate for most user programs, as well as a number of other built-in method combination types. The **define-method-combination** macro is provided for declaring new types of method combination.

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.

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, and they are invoked using the same syntax. Thus, like an ordinary function, a generic function can be passed as an argument and be used as the first argument to **funcall** and **apply**.

5.1 *Defining Generic Functions*

Generic functions are defined by means of the **defgeneric** and **defmethod** macros.

The **defgeneric** macro is designed to 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 lambda-list of the generic function, the argument precedence order, declarations that pertain to the generic function as a whole, the method combination type, the class of the generic function, and the class of the methods of the generic function.

The Common Lisp Object System provides default values for these properties, so that the use of **defgeneric** is not essential. The **defgeneric** macro can also be used to specify a set of methods on the generic function. If no methods are specified, a generic function with no methods is created.

If a **defgeneric** form is evaluated and a generic function of the given name does not already exist, a new generic function object is created.

When a new **defgeneric** form is evaluated and a generic function of the given name already exists, the existing generic function object is modified. This modification may include replacement of existing methods on the generic function or the addition of new methods.

Local generic functions can be defined by using the **generic-flet**, **generic-labels**, and **with-added-methods** special forms. Anonymous generic functions are defined by using the **generic-function** macro.

6. 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 when the generic function is called, the class or identity of each argument to the generic function determines which method or methods are eligible to be invoked.

A method object contains a lambda-list, 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.

A parameter specializer is associated with each required formal parameter of a method. A method's parameter specializers are used to determine when that method can be invoked. A parameter specializer is either a class or a list of the form `(`**eql** *object*`)`.

A method can be selected for a set of arguments when each required argument satisfies its corresponding parameter specializer. Such a method is said to be an *applicable method* for those arguments. 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 `(`**eql** *object*`)` and the argument is **eql** to *object*.

A method all of whose parameter specializers are **t** is termed a *default method*. A default method is always applicable, but if it is shadowed by a more specific method, it may not be invoked.

Method *qualifiers* provide the method combination procedure a further means of distinguishing between methods. A qualifier can be any non-**nil** atom. By convention, qualifiers are usually 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**.

6.1 *Defining Methods*

The macro **defmethod** is used to create a method object. The **defmethod** form specifies the code that is to be run when the method that it defines is selected.

When a **defmethod** form is evaluated and no generic function of the given name already exists, a generic function is automatically created 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 created to be 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 **&rest** and **&key** arguments.

When a **defmethod** form is evaluated and a generic function of the given name already exists, the existing generic function object is modified to include the new method.

In addition to the method body, a method definition contains a *specialized lambda-list* and possibly one or more method qualifiers.

The specialized lambda-list specifies when that method can be selected for execution. A specialized lambda-list is like an ordinary lambda-list except that *specialized parameters* occur in the place of the names of required parameters. A specialized parameter is a list consisting of a variable name and a parameter specializer name. All required parameters in a specialized lambda-list must be specialized parameters. If some required parameter is simply a variable name, the corresponding specialized parameter is taken to be `(`*variable-name* **t**`)`.

A method definition can optionally specify one or more method qualifiers. A method qualifier is a non-**nil** atom that identifies the role of the method to the method combination type that is 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) (rho (pos x-y-position))
   (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) (theta (pos x-y-position))
   (let ((rho (position-rho pos)))
     (setf (position-x pos) (* rho (cos theta)))
     (setf (position-y pos) (* rho (sin theta)))))
```

To update the ρ-coordinate we write:

```
(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.

7. 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.

7.1 *Multiple Inheritance*

The Common Lisp Object System is a multiple-inheritance system; that is, it allows a class to directly inherit the structure and behavior of two or more otherwise unrelated classes. In a single inheritance system, if class C_3 inherits from classes C_1 and C_2, then either C_1 is a subclass of C_2 or C_2 is a subclass of C_1; in a multiple inheritance system, if C_3 inherits from C_1 and C_2, then C_1 and C_2 might be unrelated.

If no structure is duplicated and no operations are multiply defined in the several superclasses of a class, multiple inheritance is straightforward. If a class inherits two different operation definitions or structure definitions, it is necessary to provide some means of selecting which ones to use or how to combine them. The Object System uses the class precedence list for determining how structure and behavior are inherited among classes.

7.2 *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.

At most one slot with a given name is accessible in any instance of a class. If only one class in the class precedence list provides a slot description with a given slot name, inheritance is straightforward. If the slot is a local slot, 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.

If more than one class in the class precedence list of a class C provides a slot description with a given slot name, only a single slot of that name will be accessible in instances of class C. The properties of that slot result from a combination of the several slot descriptions.

7.3 *Inheritance of Methods*

In the Common Lisp Object System, generic functions are seldom associated unambiguously with a single class or instance; rather, they sit above a substrate of the class graph, and the class graph provides control information for the generic functions. It is thus more appropriate to think in terms of method applicability rather than the inheritance of methods.

Any method that is applicable to the instances of a class will also be applicable to all instances of any subclass of that class (assuming that all the other arguments to the generic function are the same).

8. Object Creation and Initialization

The Common Lisp Object System object creation and initialization protocol provides a flexible and powerful mechanism for the creation and initialization of class instances. The individual steps of the creation process are implemented by generic functions that are designed for customization. In addition, the creation and initialization process can be controlled by the use of *initialization arguments*.

8.1 *Instance Creation*

Instances are created by the generic function **make-instance**. Given a class and a series of initialization arguments, **make-instance** returns a new instance of the class.

The generic function **make-instance** checks the validity of the initialization arguments and invokes the generic function **allocate-instance** to allocate storage for the instance and the generic functions **initialize-instance** and **shared-initialize** to initialize the new instance.

8.2 *Instance Initialization*

Initialization arguments are symbols that are associated with slots and with methods for the generic functions **allocate-instance**, **initialize-instance**, and **shared-initialize**. Initialization arguments are declared as valid by means of the **:initarg** slot options to **defclass** and by their use in the lambda-lists of these methods. Default initial value forms for initialization arguments can be specified with the **:default-initargs** class option to **defclass**.

The generic function **make-instance** creates a defaulted initialization argument list by combining the initialization arguments and values supplied to it with default values for any other initialization arguments associated with the class and the applicable initialization methods. This defaulted initialization argument list has two purposes: to provide arguments for initialization methods and to fill slots with values.

The defaulted initialization argument list consists of the explicitly supplied initialization arguments and values, in the order that they were given to **make-instance**, followed by the defaulted initialization arguments, in an order that is determined by the order that the defaulted initialization arguments occur in the **defclass** forms of the class and its superclasses and in the class precedence list.

The generic functions that initialize instances, **initialize-instance** and **shared-initialize**, use the defaulted initialization argument list to fill slots. A slot is filled with the value of the first initialization argument in the defaulted initialization argument list that is associated with that slot. If the slot cannot be filled in this way, it is filled according to the **:initform** slot option of the **declass** form, if this has been specified.

Object creation and initialization can be further customized by the definition of additional methods on the generic functions **make-instance**, **allocated-instance**, **initialize-instance**, and **shared-initialize**.

For example, the following method for **initialize-instance** will be run before the system-supplied instance when an instance of class **x-y-position** is created. The initialization arguments **:rho** and **:theta** are declared as valid by their use in the lambda-list of the **:before** method; their values are supplied in the call to **make-instance**.

```
(defmethod initialize-instance :before
                              ((pos x-y-position)
                               &key ((:rho rho) 0.0 rho-supplied)
                                    ((:theta theta) 0.0 theta-supplied))
 (when (and rho-supplied theta-supplied)
   (setf (position-x pos) (* rho (cos theta)))
   (setf (position-y pos) (* rho (sin theta)))))


(make-instance 'x-y-position :rho 5.7 :theta (/ pi 4))
```

9. Method Combination

When a generic function is invoked, the code that is executed is one of the applicable methods of the generic function or a combination of several of them. This code is termed the *effective method*.

Computing 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 the function **call-next-method** is invoked; what value or values to return.

In order to compute the effective method, the set of applicable methods for the given arguments is first determined. These methods are then sorted according to precedence order, so that the most specific method occurs first. Method combination is then applied to the sorted list of methods to produce the effective method.

The effective method is called with the same arguments that were passed to the generic function. The values that the effective method returns are returned as the values of the generic function.

9.1 *Standard Method Combination*

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

A *primary method* defines the main action of the effective method. Primary methods have no method qualifiers. Standard method combination requires that if there are any applicable methods at all, there must be an applicable primary method.

An *auxiliary method* modifies the action of the primary method or of another auxiliary method. The auxiliary methods are **:before**, **:after**, and **:around** methods. In standard method combination, an auxiliary method can have exactly one qualifier that is either **:before**, **:after**, or **:around**.

In standard method combination, the applicable methods are called as follows:

If any **:around** methods exist, the most specific **:around** method is called. The function **call-next-method** can be used within the body of an **:around** method to call the next method. By convention, **:around** methods almost always use **call-next-method**. If **call-next-method** is not used, no other methods will be invoked.

When **call-next-method** is invoked within an **:around** method, the next most specific **:around** method is called, if one is applicable.

When there are no **:around** methods or when **call-next-method** is called from within the least specific **:around** method, the other methods are called as follows:

1. All the **:before** methods are called in increasing order of precedence, the most specific method first.

2. The most specific primary method is called. The function **call-next-method** can be used inside the body of a primary method to invoke the next most specific primary method.

3. All the **:after** methods are called in decreasing order of precedence, the least specific method first.

If any **:around** methods are called, the value or values that are returned by the most specific **:around** method will be those that are returned by the generic function invocation.

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

The values of all **:before** and **:after** methods are ignored.

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 order of the primary methods cannot be reversed.

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 :accessor general-window-initialized)
    (bitmap :type bitmap :accessor general-window-bitmap)
    (viewports :type list :accessor general-window-viewports)))
```

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 **viewports** 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 announcement-window-contents)))

(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. These methods are invoked in the following order: 1. the **:around** method, 2. the **:before** method, 3. the primary method, and 4. the **:after** method.

9.2 *Other Types of Method Combination*

In addition to standard method combination, the Common Lisp Object System provides the built-in method combination types +, **and**, **append**, **list**, **max**, **min**, **nconc**, **or**, and **progn**.

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

10. 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.

When a class is redefined, the existing class object is modified to reflect the new class definition, and changes are propagated to its instances and to instances of any of its subclasses. The updating process may modify a given instance, but it does not affect the identity of the instance as defined by the **eq** function. The updating process does not cause any new instances to be created.

Users can define methods on the generic functions **update-instance-for-redefined class** and **shared-initialize** to control the redefinition process. The generic function **update-instance-for-redefined-class** is invoked automatically by the system after **defclass** has been used to redefine an existing class.

Users can also explicitly request that the class of an instance be changed by invoking the function **change-class**. The generic function **update-instance-for-different-class** is invoked by **change-class**. Its behavior can be customized by the definition of additional methods.

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

Instances of **x-y-position** can be automatically updated by defining a method for **update-instance-for-different-class**. The method in the example below is a **:before** method so that it will not shadow the behavior of the system-supplied primary method on **update-instance-for-different-class**.

```
(defmethod update-instance-for-different-class :before
                                                ((old x-y-position)
                                                 (new rho-theta-position)
                                                &key)
  ;; Information is copied 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))))
```

The function **change-class** can now be used to change a particular instance of the class **x-y-position**, **p1**, to be an instance of **rho-theta-position**:

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

11. Meta-Objects

The Common Lisp Object System is implemented in terms of a set of objects that correspond to predefined classes of the system. These objects are termed *meta-objects*. The Common Lisp Object System *meta-object protocol* specifies a set of generic functions and methods on these objects and thus defines the behavior of the Object System itself.

The set of predefined metaclasses provided by the Common Lisp Object System includes **standard-class**, **structure-class**, **standard-method**, and **standard-generic-function**.

The class **standard-class** is the default class of classes defined by **defclass**. The class **structure-class** is the default class of all classes defined by **defstruct**.

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

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

There are also other, more general classes from which these metaclasses inherit.

12. References

Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon, *Common Lisp Object System Specification*, X3J13 Document 88-002R.

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.

Daniel G. Bobrow, and Gregor Kiczales, "The Common Lisp Object System Metaobject Kernel: A Status Report," ACM Lisp and Functional Programming Conference, 1988.

Linda G. DeMichiel and Richard P. Gabriel, "The Common Lisp Object System: An Overview," Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1987.

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

Sonya E. Keene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, Reading Massachusetts, 1988.

David A. Moon, "The Common Lisp Object-Oriented Programming Language Standard," in Won Kim and Fred Lochovsky, eds., *Object-Oriented Concepts, Applications, and Databases*, Addison-Wesley, Reading, Massachusetts, 1988.

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

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