The Common Lisp Object System: An Overview

by

Linda G. DeMichiel and Richard P. Gabriel
Lucid, Inc.
Menlo Park, California

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

2. History of the Common Lisp Object System

The Common Lisp Object System is an object-oriented programming paradigm designed for Common Lisp. The lack of a standardized object-oriented extension for Common Lisp has long been regarded as a shortcoming by the Common Lisp community. Two separate and independent groups began work on an object-oriented extension to Common Lisp several years ago. One group is Symbolics, Inc. with New Flavors, and the other is Xerox PARC with CommonLoops. During the summer of 1986, these two groups met to explore combining their designs for submission to X3J13, a technical working group charged with producing an ANSI standard for Common Lisp.

At the time of the exploratory meetings between Symbolics and Xerox, the authors of this paper became involved in the technical design work. The major participants in this effort were David Moon and Sonya Keene from Symbolics, Daniel Bobrow and Gregor Kiczales from Xerox, and Richard Gabriel and Linda DeMichiel from Lucid.

By March 1987 this three-way collaborative effort had produced a strong draft of a specification for the bulk of the Object System. X3J13 has voted an endorsement of that specification draft, stating that it would almost certainly be adopted as part of the standard and encouraging implementors to proceed with trial implementations. This paper is a report on the specification that was presented to X3J13.

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

Several aspects of the Object System stand out upon inspection: a) it is a layered system designed for flexibility; b) it is based on the concept of generic functions rather than on message-passing; c) it is a multiple inheritance system; d) it provides a powerful method combination facility; e) the primary entities of the system are all first-class objects.

3.1 *The Layered Approach*

One of the design goals of the Object System is to provide a set of layers that separate different programming language concerns from one another.

The first level of the Object System provides a programmatic interface to object-oriented programming. This level is designed to meet the needs of most serious users and to provide a syntax that is crisp and understandable. The second level provides a functional interface into the heart of the Object System. This level is intended for the programmer who is writing very complex software or a programming environment. The first level is written in terms of this second level. The third level provides the tools for the programmer who is writing his own object-oriented language. It allows access to the primitive objects and operators of the Object System. It is this level on which the implementation of the Object System itself is based.

The layered design of the Object System is founded on the *meta-object protocol*, a protocol that is used to define the characteristics of an object-oriented system. By using the meta-object protocol, other functional or programmatic interfaces to the Object System, as well as other object systems, can be written.

3.2 *The Generic Function Approach*

The Common Lisp Object System is based on *generic functions* rather than on message-passing. This choice is made for two reasons: 1) there are some problems with message-passing in operations of more than one argument; 2) the concept of generic functions is a generalization of the concept of ordinary Lisp functions.

A key concept in object-oriented systems is that given an operation and a tuple of objects on which to apply the operation, the code that is most appropriate to perform the operation is selected based on the classes of the objects.

In most message-passing systems, operations are essentially properties of classes, and this selection is made by packaging a message that specifies the operation and the objects to which it applies and sending that message to a suitable object. That object then takes responsibility for selecting the appropriate piece of code. These pieces of code are called *methods*.

With unary operations, the choice of a suitable object is clear. With multiary operations, however, message-passing systems run into problems. There are three general approaches to the problem of selecting a suitable object to which to send a message: currying, delegation, and distribution of methods.

Currying is a technique for turning a multiary operation into series of unary operations. In message-passing systems, this is accomplished by having the objects in question send messages among themselves to gather the contributions of each object to the final result of the operation. For example, adding a sequence of numbers can be done by asking each number to add itself to some accumulated total and then to send a message to the next object for it to do the same. Every object must know how to start and end this process. Currying may result in a complicated message-passing structure.

Delegation is a technique whereby an object is defined to handle an operation on a number of other objects. For example, to sum a sequence of numbers, there can be an object that will accept a message containing the identities of the numbers and then perform the addition. Thus, every object that can be involved in a multiary operations must know how to further delegate operations.

Distribution of methods is a technique whereby every object that can be involved in a multiary operation can be outfitted with enough information to directly carry out the operation.

In the generic function approach, objects and functions are autonomous entities, and neither is a property of the other. Generic functions decouple objects and operations upon objects; they serve to separate operations and classes. In the case of multiary operations, the operation is a generic function, and a method is defined that performs the operation on the objects (and on objects that are instances of the same classes as those objects).

Generic functions provide not only a more elegant solution to the problem of multiary operations but also a clean generalization of the concept of functions in Common Lisp. Each of the methods of the generic function provides a definition of how to perform an operation on arguments that are instances of particular classes or of subclasses of those classes. The generic function packages those methods and selects the right method or methods to invoke.

Furthermore, in Common Lisp, arithmetic operators are already generic in a certain sense. The expression

```
(+ x y)
```

does not imply that x and y are of any particular type, nor does it imply that they are of the same type. For example, x might be an integer and y a complex number, and the + operation is required to perform the correct coercions and produce an appropriate result.

Because the Object System is a system for Common Lisp, it is important that generic functions be first-class objects and that the concept of generic functions be an extension and a generalization of the concept of Common Lisp functions. In this way, the Object System is a natural and smooth extension of Common Lisp.

A further problem with message-passing systems is that there must be some way within a method of naming the object to which a message was sent. Usually there is a pseudovariable named something like "self." With generic functions, parameters are named exactly the same

way that Common Lisp parameters are named—there is conceptual simplicity in using the fewest constructs in a programming language.

Some message-passing systems use a functional notation in which there is a privileged argument position that contains the object to which a message is to be sent. For example

```
(display x window-1)
```

might send a message to x requesting that x display itself on the window `window-1`. This expression might be paraphrased as

```
(send 'display x window-1)
```

With generic functions there are no such privileged argument positions.

In addition to these advantages over message-passing, there are three other fundamental strengths of generic functions:

1. It is possible to abstract the definition of a generic function into parts that are conceptually independent. This is accomplished by splitting the definition of a generic function into separate parts where each part is the partial function definition for a particular set of classes. This leads to a new modularization technique.

2. It is possible to spread the definition of a generic function among the places where the partial definitions make the most sense. This is accomplished by placing the appropriate **defmethod** forms where the relevant classes are defined.

3. It is possible to separate the inheritance of behavior from the placement of code. Generic functions select methods based on the structure of the class graph, but the generic functions are not constrained to be stored within that graph.

In a message-passing system, the class graph and the instances are truly central because the methods are associated with a particular class or instance. It makes sense in this setting to think of the methods as part of the structure of a class or an instance.

In a generic-function system, the generic functions provide a very different view of methods. Generic functions become the focus of abstraction; they are rarely associated unambiguously with a single class or instance; they sit above a substrate of the class graph, and the class graph provides control information for the generic functions.

Despite the advantages of generic functions, there may at times be reasons for preferring a message-passing style; such a style can be implemented by means of generic functions or by means of the Common Lisp Object System Meta-Object Protocol.

3.3 *The Multiple Inheritance Approach*

Another key concept in object-oriented programming is the definition of structure and behavior on the basis of the *class* of an object. Classes thus impose a type system—the code that is used to execute operations on objects depends on the classes of the objects. The subclass mechanism allows classes to be defined that share the structure and the behavior of other classes. This subclassing is a tool for modularization of programs.

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 a linearized *class precedence list* for determining how structure and behavior are inherited among classes.

3.4 *The Method Combination Approach*

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.

Method combination is used to define how the methods that are applicable to a set of arguments can be combined to provide the values of a generic function. In many object-oriented systems, the most specific applicable method is invoked, and that method may invoke other, less specific methods. When this happens there is often a combination strategy at work, but that strategy is distributed throughout the methods as local control structure. Method combination brings the notion of a combination strategy to the surface and provides a mechanism for expressing that strategy.

A simple example of method combination is the common need to surround the activities of a method's invocation with a prologue and an epilogue: the prologue might cache some values in an accessible place for the method, and the epilogue will write back the changed values.

The Object System provides a default method combination type, *standard method combination*, that is designed to be simple, convenient, and powerful for most applications. Other types of method combination can easily be defined by using the **define-method-combination** macro.

3.5 *First-Class Objects*

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

The concept of "first-class" is important in Lisp-like languages. A first-class object is one that can be explicitly made and manipulated; it can be stored anywhere that can hold general objects.

Generic functions are first-class objects in the 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.

3.6 *What the Common Lisp Object System Is Not*

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 not necessary for a class to have any instances, but all objects are instances of some class. The class system defined by the Object System and the Common Lisp type system are tightly integrated, so that one effect of the Object System is to define a first-class type system within Common Lisp. The class of an object determines the set of operations that can be performed on that object.

There are two fundamental sorts of relationships involving objects and classes: the subclass relationship and the instance relationship.

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.

A typical situation is that one class, C_1, represents a possibly infinite set of objects, and a second class, C_2, represents a subset of that set; in this case C_2 is a subclass of C_1.

An object is an instance of a class if it is an example of a member of that class. If the class represents a set, then an instance is a member of that set.

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.

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

The 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.

All programmer-defined classes are instances of the class named **standard-class**, which is an instance of the class named **class**; the class named **class** is an instance of itself, which is the fundamental circularity in the Object System.

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 new 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 represented as a list consisting of the class followed by its direct superclasses in the order mentioned in the **defclass** form.

A slot specifier includes the name of the slot and zero or more slot options that pertain to that particular slot. The name of a slot is a symbol that could be used as a Common Lisp variable name. 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.

There are two kinds of slots: slots that are local to an individual instance and slots that are shared by all instances of the 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 syntax of **defclass** provides several means for generating methods to read and write slots. Methods can be requested for all slots or for particular slots only. Methods can be requested to read and write slots or to read slots only. 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 as variables. It is also possible to specify whether the macro **with-slots** is to use the accessors or the function **slot-value** to access slots.

A 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 specifying 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 want 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)
```

The macro **defclass** is part of the Object System programmatic interface and, as such, is on the first of the three levels of the Object System. When applied to an appropriate metaclass, the function **make-instance** provides the same functionality on the second level.

4.2 *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.

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 the arguments supplied to it. The *methods* associated with the generic function define the class-specific operations of the generic function.

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 is able to perform different series of operations and to combine the results of the operations in different ways, depending on the class or identity of one or more of its arguments.

The operations of a generic function are defined by its methods. Thus, generic functions are objects that can 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 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.

In addition to a set of methods, a generic function object comprises a lambda-list, a method combination type, and other information. In Common Lisp a lambda-list is a specification of the parameters that will be passed to a function. The syntax of Common Lisp lambda-lists is complex, and the Object System extends it further.

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 and 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 Object System offers a default method combination type that is appropriate for most user programs. The 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 Object System provides default classes for all generic function, method, and class objects, the programmer may choose to implement any or all of these by using classes of his own definition.

Generic functions in the Object System are nearly indistinguishable from ordinary functions: they can be applied, stored, and manipulated exactly as ordinary functions are. In this way, the Object System is smoothly integrated into the Common Lisp framework, and there is no sense in which Common Lisp programs are partitionable into the functional parts and the object-oriented parts.

5.1 *Defining Generic Functions*

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

The **defgeneric-options** macro is designed to allow for the specification of properties that pertain to the generic function as a whole, and not just to individual methods.

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—this is sometimes referred to as the "contract" of the generic function. These properties include the following: the lambda-list of the generic function; a specification of the order in which the required arguments in a call to the generic function are to be tested for specificity when selecting a particular method; declarations that pertain to the generic function as a whole; the class of the generic function; the class of all the methods of the generic function; and the method combination type to be used with this generic function. The Object System provides a set of default values for these properties, so that use of the **defgeneric-options** macro is not essential.

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 there is no generic function of the given name, however, it automatically creates a generic function with default values for the argument precedence order (left-to-right, as defined by the lambda-list), the generic function class (the class **standard-generic-function**), the method class (the class **standard-method**), and the method combination type (standard method combination). 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.

6. Methods

The class-specific operations provided by generic functions are themselves defined and implemented by *methods*. The class or identity of each argument to the generic function indicates which method or methods are eligible to be invoked.

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.

Method qualifiers give the method combination procedure a further means of distinguishing 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*.

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. 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. By convention, qualifiers are usually keyword symbols.

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)
```

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

7. 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 that has

a superclass that has been redefined) occurs at an implementation-dependent time; this 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. 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** by using **change-class**:

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

8. Inheritance

Inheritance is the key to program modularity within the 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.

8.1 *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.

8.2 *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.

9. 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 is preceded by other classes with respect to R, the class selected is chosen deterministically. The rule that was chosen for this selection process is designed 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.

It is required that an implementation of the Object System signal an error if R is inconsistent, that is, if the class precedence list cannot be computed.

10. 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 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 Object System provides a default method combination type, *standard method combination*. The programmer can define other forms of method combination by using the **define-method-combination** macro.

10.1 *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; *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 given as follows:

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.

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 not to create their bitmaps until they are actually needed. Thus 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 to 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))
   (call-next-method))
```

```
(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 the following: 1) the **:around** method, 2) the **:before** method, 3) the primary method, and 4) the **:after** method.

11. Meta-Object Protocol

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*. Because meta-objects underlie the rest of the object system, they may be used to define other objects, other ways of manipulating objects, and hence other object-oriented systems. The use of meta-objects is specified by in the Common Lisp Object System meta-object protocol.

The meta-object protocol is designed for use by implementors who need to tailor the Object System for particular applications and by researchers who wish to use the Object System as a prototyping tool and delivery environment for other object-oriented paradigms. It is also designed for the implementation of the Object System itself.

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

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

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

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

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

The classes **class**, **standard-class**, **slot-description**, **standard-slot-description**, **method**, **standard-method**, **generic-function**, and **standard-generic-function** are all *instances* of the class **standard-class**.

These classes represent a minimal structure and functionality on which the implementation of all other classes in the system is based. Implementations may choose to add additional slots for these classes.

12. Evaluation of the Design Goals for the Common Lisp Object System

Even when designers start with a clean slate, it is difficult to achieve the design goals of a language, but the task of satisfying those goals is very much more difficult when two different languages with existing user communities are being merged. The final design of a language—its shape and clarity—depends on the ancestor languages. If the ancestor languages do not have a clean design, the designers must weigh the desirability of a clean design against the impact of changes on their existing user communities.

In this case there were two such ancestor languages, each with a young but growing user community. CommonLoops has a relatively clean design. Its experimental implementation, "Portable CommonLoops," is freely distributed to a small user community. New Flavors is the second generation of a basic system called "Flavors." Flavors is an older, commercially supported message-passing system. New Flavors is a generic-function-based descendant of Flavors, but a descendant that supports compatibility with its ancestor. The design of New Flavors is not as clean as that of CommonLoops, but its user community is larger and is accustomed to commercially motivated stability.

With these facts in mind, let us examine the design goals for the Common Lisp Object System.

1. Use a set of levels to separate programming language concerns from each other.

This goal was achieved quite well. The layering is well defined and has been shown to be useful both in terms of implementing the Object System as well as in terms of implementing another object-oriented system (Hewlett-Packard's CommonObjects). CommonLoops already has a layered design.

2. Make as many things as possible within the Object System first-class.

This goal was also achieved quite well. Both classes and generic functions are first-class objects. Generic functions were not first-class objects in either CommonLoops or New Flavors, but fortunately the user-community disruption is minimal for these changes.

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.

There are two additional languages in the Object System: the language of method combination keywords and a pattern language for selecting methods, which is used when defining new method combination types. Method combination was considered important enough to tolerate these additional languages.

CommonLoops has no additional languages. New Flavors has a rich method combination facility with more complex versions of the two additional languages just mentioned; these two languages were simplified for inclusion in the Object System.

4. Be willing to trade off complex behavior for conceptual and expository simplicity.

This was largely achieved, although it required a considerable amount of additional technical work. For example, the class precedence list algorithm was altered from its original form largely because the original was not easily explainable.

5. Make the specification of the language as precise as possible.

The specification of Common Lisp is fairly ambiguous because it represents a technical compromise between several existing Lisp dialects. The ground rules of the compromise were explicitly established so that with a small effort each of the existing Lisp dialects could be made into a Common Lisp.

The Object System design group was in a similar position to the Common Lisp design group, but on a smaller scale. The language of the specification is considerably more precise than the language of the Common Lisp specification.

13. Conclusion

The Common Lisp Object System is an object-oriented paradigm with several novel features; these features allow it to be smoothly integrated into Common Lisp.

14. 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 Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

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

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