CLOS in Context:
The Shape of the Design Space

Daniel G. Bobrow
Xerox PARC

Richard P. Gabriel
Lucid, Inc.

Jon L White
Lucid, Inc.

May 3, 2004

Programming language design combines the art of invention with judicious adaptation and rejection of ideas previously tried. This chapter presents aspects of the design of the Common Lisp Object System (CLOS) in the context of related ideas from many languages, providing a view of the CLOS within a broader space of designs.

CLOS is the latest in a long history of additions of object-oriented extensions to Lisp. But CLOS is the first such extension that integrates the type and class systems, and provides a uniform client interface for calling ordinary functions and functions implemented in an object-oriented style.

CLOS is also designed with three constraints not found in other object-oriented extensions. The first is to provide essential compatibility with other previously used object-oriented LISP facilities, such as the Symbolics Flavors system. Essential compatibility implies an easy path for transforming programs, supporting the most frequently used capabilities in such systems. The second constraint is that CLOS is to facilitate experimentation with new language features in a way that allows them to integrate but not interfere with the kernel of the system. The third is that CLOS kernel facilities are to be efficiently implementable on a wide range of hardware, from special purpose Lisp machine to RISC processors. The CLOS metaobject protocol[18] supports these three constraints by making available to the user and system developer a specification for the interpreter for the language, and an object-oriented protocol that supports extension. (Actually the metaobject protocol provides only a partial specification—for only some steps—of the processing of CLOS. If certain customizations of those steps are made by the user, the effect is as if the interpreter were customized.)

The four sections of this chapter provide four projections of the design space for object-oriented systems. In each, we characterize the range of variability found in current programming practice, locate CLOS in this context, and then explore extensions that might be coherently added to CLOS. The extensions are important for two reasons. First, a language must be judged not only on what it contains, but on what it leaves out. The extensions are facilities that could have been included in CLOS, but only with the attendant risk of complicating standard practice and reducing understandability. Secondly, these extensions show how the CLOS metaobject protocol provides a smooth continuum of capabilities from system developer to user.

The first section explores four traditions in which incremental definition of operations has appeared. These include the object-oriented programming (OOP) languages, data-driven interpreters, pattern-directed languages, and languages with polymorphism. Incremental definition is important because it supports (i) conceptual separation of the interface and implementation(s) of an operation, (ii) extension and specialization of the domain of operations with new implementations.

The second section focuses on the concepts of class and type. It distinguishes five notions of type and shows different ways the introduction of classes into a language can interact with the type system. The CLOS choice of making classes support the entire type system is important in moving all of Common Lisp towards being object-oriented.

The third section on factoring descriptions focuses on use of mixin classes, and on combining and enhancing methods. Factoring descriptions in a programming language is important

because it makes possible more and finer grained reuse of programs, and hence supports greater programming productivity.

The final section, on reflection, looks at the issue of embedding self-description within the system with first class objects that represent program elements. Reflection facilitates programmatic introspection, interpreter extension, and incremental program development. Reflection is important because it allows the use of a programming language to support the programming process itself.

2.1 Incremental extension of operations

Some operations make sense only on a particular type of data. For example, `string-length` requires its argument to be a `string`. But often there is a generic operation with a core of common meaning, like `length`, that makes sense for many different types of data. For such a *generic function* the code to be run must depend on the data type of the argument provided. Such *polymorphic functions*, as they are sometimes called, exist in a number of different programming languages. In languages like Lisp, where lambda abstraction is the primary extension mechanism, a function is usually defined by a single, monolithic piece of code; any implementation conditionality that depends on the types of passed parameters is expressed as code explicitly programmed in by the user.

In contrast, a *partitioned operation* is one in which the implementation for each type can be separated textually. This allows incremental extension of an operation, without requiring modification or even access to the original source. The CLOS notion of generic functions supports automatic dispatch to separately defined, type-specific implementational parts. CLOS implementational parts are called *methods*. CLOS automatically combines them in a single generic function, with system-generated code embedded in the generic function to select the appropriate implementation at run-time. In the CLOS system, the definition of a generic function can be extended incrementally at any time by defining a new method.

As a simple example, suppose one wanted to define a symbolic differentiation program in CLOS. One might start with class definitions such as the following:

```
(defclass algebraic-combination (standard-object)
  ((s1 :initarg :first :accessor first-part)
   (s2 :initarg :second :accessor second-part)))

(defclass symbolic-sum (algebraic-combination)
  ())

(defclass symbolic-product (algebraic-combination)
  ())
```

These definitions specify that `symbolic-sum` and `symbolic-product`, both subclasses of `arithmetic-combination`, each have two slots, accessed with functions `first-part` and

`second-part`. These classes describe instances used to represent algebraic combinations. For each class of algebraic combination we define an independent method for implementing `deriv`. Here are two such method definitions:

```
(defmethod deriv ((expression symbol) variable)
  (if (eq expression variable) 1 0))

(defmethod deriv ((expression symbolic-sum) variable)
  (make-instance 'symbolic-sum
    :first  (deriv (first-part expression) variable)
    :second (deriv (second-part expression) variable)))
```

The `deriv` operation is invoked by standard function calling syntax, because it is implemented as a standard procedural interface. Suppose `exp` is bound to a structure that represents an algebraic expression; then the following computes its derivative with respect to the variable `X`:

```
(deriv exp 'X)
```

The `deriv` generic function will use the type of the expression bound to `exp` to select a method to be called. The method definitions shown support only two of the cases required for complete differentiation. As classes to support other algebraic combinations are introduced, corresponding methods can be defined, thus incrementally extending the `deriv` operation. A new method can be defined without having to change the sources of previously defined methods and classes. These new definitions can be added even after previous definitions have been compiled, and instances of the structures have been created. Furthermore, any client of the `deriv` function need not worry about whether `deriv` is implemented as a function or generic function. The procedural abstraction barrier is still in force.

Although the concept of generic functions is familiar, its role as the foundation of object-oriented programming in Lisp is relatively new. Earlier inclusions of object-oriented programming in Lisp incorporated the message-passing style typified by Smalltalk, which is only one of a number of programming traditions supporting incrementally defined implementations with automatic choice of code. The data-driven tradition uses an element of an argument as a key to choose the implementation; in pattern-directed invocation, the structure of the arguments are used to select an invocation; in the polymorphism tradition, the types of the arguments provide the basis for implementation selection (usually at compile-time). We examine each of these traditions in turn, arriving at a set of dimensions that allow us to understand the space of language design tradeoffs.

2.1.1 Object-Based Programming Tradition

The object-based programming tradition is usually identified with message-passing. A message containing the name of an operation and arguments is sent to a *receiver*. The receiver and the name of the operation are jointly used to select a *method* to invoke.

For example, in Smalltalk [15] the solution to the symbolic derivative problem above might define a class for SymbolicSum:

```
class:                  AlgebraicCombination
instance variables      s1, s2
class methods:
instance creation
  of: first and: second
    ↑super new setargs: first and: second.
instance methods:
private
  setargs: first and: second
    s1←first. s2←second.
```

```
class:                  SymbolicSum
superclass:             AlgebraicCombination
instance methods:
  deriv: variable
    ↑SymbolicSum of: (s1 deriv: variable)
                and: (s2 deriv: variable).
```

As in CLOS, an expression is represented as a nested composite of objects (like a parse tree), and each class has a method that defines how it responds to a message whose selector is `deriv:`. Thus, the expression

```
(s1 deriv: variable)
```

is understood as sending a message to `s1` with selector `deriv:` and argument `variable`.

Smalltalk occupies one particular point in a spectrum of object-based programming styles. In Smalltalk the structure of instances, as in CLOS, is determined by class definitions. In Smalltalk methods are much more strongly associated with classes than in CLOS, and are thought of as belonging to the classes. Sharing is done through class-based inheritance of methods and structural descriptions. The Self system [27] is very similar to Smalltalk. However, in Self, methods are thought of as belonging to individual objects. Sharing behavior is done by delegating an operation to another object. Delegation to other objects can be done in both CLOS and Smalltalk in ad hoc ways, but Self provides language support for delegation.

Lisp is a language with a syntax based on function application, while Smalltalk and Self are message-based languages. There is a difference in syntax for invocation of operations between CLOS on one hand, and Smalltalk and Self on the other, which reflects this distinction. In functional application languages, the operation appears leftmost in a function

call form, which indicates primacy in Western tradition. In object-oriented languages, the receiver object appears leftmost. In early mergers of object-oriented programming and Lisp, a `send` operation was introduced to perform message-passing; its leftmost argument was the receiver object. Both definitions operator invocations looked much like those in Smalltalk.

2.1.2 Data-driven Dispatch

The data-driven tradition is based on the technique of explicitly dispatching to a first-class function determined by some aspect of relevant data such as the arguments to an operation. First-class functions are needed because in most cases a function must be fetched from a data structure and invoked based on the relevant aspect of data. (It is sufficient for the language to have a means to store pointers to functions and be able to invoke a function given a pointer to it. The language C satisfies this condition.) However, generally the dispatch is done in an ad hoc manner, meaning there is no language-supported mechanism. For example, suppose that algebraic expressions are represented in Lisp as lists with prefix arithmetic operators:

```
(+ (expt X 2) (* 2 X) 1)
```

The following could be the driver for a data-driven symbolic differentiation program where each differentiation rule is implemented by one function in a table:

```
(defun deriv (expression variable)
  (if (atom expression)
      (if (eq expression variable) 1 0)
      (funcall (get-from-table (first expression)
                               *deriv-table*)
               (rest expression)
               variable))))
```

The way to differentiate a sum is to sum the derivatives, and this behavior can be added quite easily:

```
(add-to-table *deriv-table*  '+
      #'(lambda (expressions variable)
          `(+ ,@(mapcar #'(lambda (expression)
                            (deriv expression variable))
                        expressions))))
```

The data-driven tradition has been used extensively in symbolic algebra and artificial intelligence.

2.1.3 Pattern-directed Invocation

Pattern-directed invocation provides a means for the programmer to describe the arguments on which a particular clause of a definition is to operate. More specifically, the choice of what to do is dependent on the morphology of the argument expression. Pattern-directed invocation can be used with a number of different control structures. It is a cornerstone of production system (rule-based) languages like OPS-5 [4], and characteristic of backward chaining rule languages, such as Prolog [9]. A Prolog program to differentiate an expression might look like this:

```
deriv([+ X Y],V,[+ DX DY]) <= deriv(X,V,DX), deriv(Y,V,DY).
deriv([* X Y],V,[+ [* X DY] [* Y DX]]) <=  deriv(X,V,DX),
                                           deriv(Y,V,DY).
deriv(X,X,1).
deriv(X,Y,0).
```

And it would be invoked like this:

```
deriv(expression,variable,result)
```

Each clause is simply a relation that states that the left-hand side of the clause holds true if the right-hand side does. A clause with an empty right-hand side indicates that the left-hand side is always true. The control structure for backwards-chaining pattern-directed languages is that of a *goal* statement whose unknowns are eliminated by matching the left-hand sides of clauses against the goal statement or a part of it, and replacing the goal by zero or more subsidiary goals obtained from the right-hand side of the matching pattern, along with possible execution of attached code.

Sometimes several left-hand sides match, or a single left-hand side can match more than one way. In this case, one of the matches is chosen. If this choice is successful through *backward chaining*, a solution has been found. If the backward chaining meets with failure, then by *backtracking*, other solutions can be found. To find all solutions, the program is forced to backtrack after each solution. Sometimes, to shorten the running time of a program, it is necessary to prune backtracking. There is an operator, called *cut*, that prevents backtracking due to failure from retreating beyond a certain point.

Matching a data description to a set of supplied arguments is a succinct, powerful way to describe the code selection process. Backtracking control structure makes sense in a problem-solving setting where the issue is search. In production system programming, the conflict that arises when more than one left-hand side matches is resolved differently. In some systems, only the most specific matching rule is fired. In some systems all applicable rules are fired once. There is no concept of failure; each rule firing brings about changes in the workspace, and these changes make new rules applicable. Thus the concept of using pattern-driven selection of code is independent of the control structure of the higher level program in which it is embedded.

2.1.4 Polymorphism Tradition

Some languages have *polymorphic operators* [8]. A polymorphic operator is one that can accept arguments of a variety of types or classes. It is typical of languages in the polymorphic tradition that multiple arguments can be used for selection of the implementation and that the selection is usually done at compile-time on the basis of a description of the arguments. Burstall [5] has called this *ad hoc* polymorphism. He contrasts this with *universal* polymorphism, where because the representation of arguments is uniform, the same code can be executed by the operator for any types of the arguments. As an example of universal polymorphism, consider the Lisp function to reverse a list. It depends only on the representation of a list cell as a pair consisting of a pointer to a data element and a link to the next list element. The pointer to the data element is of uniform shape. Hence `reverse` can operate on lists of numbers, lists of symbols, and lists of numbers or symbols—lists of anything, really. Therefore, the types of the objects stored in the lists are irrelevant.

As an example of ad hoc polymorphism, FORTRAN has polymorphic arithmetic operators where the implementation chosen is based on the types of the arguments. Types of variables are determined by the first letter of variable names. The rule is that any variable whose name begins with the letters `I`,...,`N` refer to fixed point numbers while all others refer to floating point numbers. Types of expressions are based on the known result types of polymorphic operations with known argument types. For example, the code fragment `I+J` is a fixed point addition while `X+Y` is a floating point addition, and `X+I` returns a floating point result with an implied coercion.

At compile-time, the appropriate operation is chosen based on the apparent type of the arguments. FORTRAN does not allow users to define such polymorphic operations. C++ supports ad hoc polymorphic operations; operations of the same name which take different types of arguments are simply treated as independent. Users can add extensions to most polymorphic operations. For example, users can define a polymorphic function `size` that takes an argument of type `string`, and another of the same name for an argument of the user-defined structure type `shoe`. The C++ compiler will choose at compile-time the appropriate implementation to use based on the statically declared or inferred type of the argument in the calling code. The declared types of multiple arguments can be used to make this choice.

Common Lisp even before CLOS had some polymorphic operators. All the built-in Lisp arithmetic operators perform type-specific operations depending on the types of the actually supplied arguments. For example, evaluating the expression `(+ x y)` does floating point addition if the value of either `x` or `y` is a floating point number; if one of them is not floating point, it is first coerced to floating point.

The difference between the FORTRAN and Lisp notions of arithmetic polymorphism is that FORTRAN determines the operator implementation at compile-time, while Lisp usually delays the decision to run-time. More about this flexibility will be discussed below.

In CLOS, any of the argument parameters of a method definition can likewise be constrained by a class specification. This allows all the arguments to a generic function—not just the first one—to participate in method selection. This is unusual in class-based systems.

2.1.5 Categorizing Language Families

There are four styles for object-oriented systems: object-centric, class-centric, operation-centric, and message-centric.

In object-centric systems, objects have state and associated operations [28]—structure and behavior—but there is no special provision for classes. Object-centric systems provide a minimal amount of data abstraction. Sharing may be achieved through delegation.

Class-centric systems give primacy to classes: Classes describe the structure of objects, contain the methods defining behavior, provide inheritance topologies, and specify data sharing between objects.

Operation-centric systems give primacy to operations: The operations themselves contain all the behavior for objects, but they may use classes or objects to describe inheritance and sharing.

Message-centric systems give primacy to messages: Messages are first-class, and carry operations and data from object to object; but they also may use classes or objects to describe inheritance and sharing.

Wegner [28] uses the term *object-oriented programming* for object-based systems with classes that support inheritance.

In class-centric and object-centric languages, the class or object is highlighted rather than operations; the object receives the operation name (or message), which it examines to determine what to do. But in functional languages, the operation is in control—arguments are passed to the code implementing the operation, and that code might examine the arguments to determine what to do. Only operation-centric systems support discrimination for method selection on multiple arguments because there is no accepted notion of a sequence of objects receiving a message. CLOS is a combination of class-centric and operation-centric.

These four styles can be divided into two independent axes which can be used to partition the space of languages and programming traditions, as summarized in the following table.

	message-centric	operation-centric
class/type-centric	Smalltalk C++ virtuals	CLOS C++ overloads FORTRAN
object-centric	Self Actors	Prolog Data-driven programs

2.1.6 Characterizing Partitioned Operations

A partioned polymorphic operation is an operation that is incrementally defined, potentially at different times in the development of a program, and usually at several different textual points. The meaning of the operation is determined in part by bringing together the textually separated parts of the definition into one piece, and possibly by considering the compile-time and/or run-time context of the operation. The context is determined by the nature

of the arguments (usually their type) and possibly by a wider context. Thus the design space of polymorphic languages can be categorized by a number of different independent characteristics.

The first is whether the selection of the implementation is at compile-time or at run-time. Smalltalk method selection is done only at run-time based on the class of the object. At the other extreme, FORTRAN only has compile-time selection of operations based on the (implicit) type declarations of its variables. Common Lisp is an example of a language that has chosen to let the programmer decide where along this compile-time/run-time spectrum to lie. This decision is driven by the number and nature of optional declarations the programmer supplies. C++ also lets the programmer decide, but with less flexibility than CLOS: C++ has overloads, whose implementations are selected at compile-time based on the declarations provided by the programmer; C++ also has virtual functions, whose implementations are selected at run-time.

A language is not object-oriented according to Wegner[28] unless there is some form of run-time polymorphism.

The second characteristic is whether polymorphic operators are defined only for system-defined operators or whether the programmer can define them. For example, C++ provides mechanisms for user-defined compile-time and run-time polymorphic functions; but FORTRAN does not provide any means for users to define polymorphic operators.

The third characteristic is whether operators are polymorphic in one or more than one argument. CLOS provides polymorphism for all required arguments. C++ overloads provide compile-time polymorphism for several arguments; but, asymmetrically, virtual member functions are polymorphic only on their first arguments even though overloaded functions and virtual member functions have the same client syntax. FORTRAN provides compile-time polymorphism based on all the arguments to an arithmetic operation. However Smalltalk, as with most message-centric languages, only provides language support for selection based on the class of the first argument of the operation (the receiver of the message).

Multi-argument polymorphism can be emulated in message-passing systems, but at a cost. Selection based on several arguments can be simulated by a sort of currying. Typically, a series of messages is sent, each further refining the choice of the real method. Each class in the chain calls a new message that captures in its name (selector) the classes of arguments seen so far; the last object in the series selects the appropriate method based on the resultant compound selector[16]. For example, in Smalltalk, sending the message

```
aCircle display: aScreen
```

is converted by a method on the class `Circle` to the message

```
aScreen displayCircle: aCircle
```

which captures in the selector name the type of the graphic object to be displayed on the screen.

A consequence of multi-argument polymorphism is that methods are now associated with more than one class; it is no longer possible to think of a single class as the owner of the

method, since each method parameter may refer to a different class. In CLOS, the generic function owns the methods, and its overall type signature involves the set of classes on which the applicable methods are defined.

The fourth characteristic is whether the descriptions of the arguments are restricted to types or whether other descriptions are possible. In CLOS, object identity—in addition to object type—can be used as a description. For example, a method can be defined that is applicable only when the actual argument is `EQL` to the object in the parameter specification (these are called *eql specializations*.) Neither Smalltalk nor C++ supports selection on anything other than the classes of the arguments.

A fifth characteristic is whether the syntax of the call to a generic operation must be different than the syntax for an ordinary operation; for example, does it force the client to focus on some particular argument—and hence use a message-passing style—or does it allow the client to focus on the semantics of the operation regardless of whether it is implemented in a generic way or not. For Lisp, the extension of ordinary functions to become generic, rather than the addition of a new message-passing syntax, is suggested for a number of reasons. First, generic functions are a natural extension of the pre-existing Common Lisp type-generic functions such as the numeric functions. Second, their use helps achieve consistency for client code—all operations are invoked using function call syntax. This insight was behind the switch in New Flavors [20] to use generic functions, even though methods were thought of as belonging to the class of the object which was the first argument to the generic function.

2.1.7 Extensions

CLOS supports run-time polymorphism based on the types of one or more required arguments (actually, on the subset of data types that correspond to classes). The ability to do something like pattern-directed programming is minimally supported by the ability to define a method whose selection is dependent on the identity of one or more of its arguments. For example, if we replaced the call to `make-instance` in the `deriv` method for `arithmetic-sum` with a call to `make-sum` we could include some automatic expression simplification in the generic function dispatch mechanism by using the following methods:

```
(defmethod deriv ((expression symbolic-sum) variable)
  (make-sum  (deriv (first-part expression) variable)
             (deriv (second-part expression) variable)))

(defmethod make-sum (first second)
  (make-instance 'symbolic-sum
    :first first
    :second second))

(defmethod make-sum ((first (eql 0)) second)
  second)
```

```
(defmethod make-sum (first (second (eql 0)))
  first)
```

A useful extension of this capability not currently supported by CLOS would be to support a matching process like that used in rule-based languages, where method selection can be based on the structure of arguments. By structure, we mean specifying a method that is applicable when an argument is an object with specified values in its slots.

Let us consider an example. Suppose we have a general method for finding how far a *movable-object* is from the origin. On a slow arithmetic processor, it might be worth while to have a specialized method for the case where the object was at `<0,0>`. The following is a suggestive syntax that might be used to specify this special case for an instance of *movable-object* whose slots `x` and `y` are both *eql* to 0:

```
(defmethod distance-from-origin
    ((x (:class movable-object
         :slots (x (eql 0))
                (y (eql 0)))))
  0)
```

Such a structure-based specification would make it easier to customize behavior to particular, well-described situations, similar to those used in pattern-directed languages.

Another possibly useful extension to CLOS would add parametric types as a way of specifying the scope of a method. For example, it might be useful to define a method like the following:

```
(defmethod add
    ((x (:class list :elements number))
     (y (:class list :elements number)))
  (mapcar #'+ x y))
```

The new payoff here is the ability to state concisely what would otherwise be expressed as explicit conditional code in the body of a function. In general, one can consider as a candidate extension the abstraction of any initial test of the arguments with branches to separate code depending on the outcome. The branching can be made part of the generic function dispatch, and hence allow new branches to be defined incrementally.

For all these extensions, the expectation is that a metaobject protocol for CLOS [18] can simplify the implementational effort. The idea is that for each extension, all that would be needed are new classes of generic functions and specializers that provide the appropriate behavior through specialized methods on generic functions defined in the protocol. Given the design of a metaobject protocol, it should be possible to add these extensions without interfering with the kernel language of CLOS. Of course, the detailed issues of the syntactic portions of the extensions require careful attention. This dimension of extensibility through a metaobject protocol is unique to CLOS, and was not included in the characterizing dimensions described above.

As so far described, selection of a method has only been dependent on the arguments to an operation and not on any characteristics of the operator or the environment in which the operation is being done—except insofar as the class of the generic function dictates the entire method dispatch and effective method construction mechanism. In some situations, it might be useful to use exogenous features for method selection. As illustrative examples, one might include the process in which the operation is being done or the types of returned results. Artificial intelligence applications might make use of the latter extension to select an operation based on a description of the desired effect. An early example of a system constructed on this principle can be found in [13].

2.2 Classes and Types

The Common Lisp type system existed before the incorporation of CLOS. There were a number of built-in structures. The `defstruct` extension mechanism provided for defining new types of composite objects. Types are distinguishable at run-time by a set of built-in predicates; `defstruct` extends the set of available predicates.

In addition to this structural notion of type, Common Lisp extended the notion of type to include general run-time predications, which can select out subsets of pre-existing data types. To add a new predication type, the `deftype` construct would associate a type symbol to the specified Lisp predicate. Finally, Lisp supports the notion of declarations of types within programs. Such declarations specify expected types for particular inputs and outputs of functions. These declarations are used by some compilers for optimization of code.

CLOS introduced classes as the primary means of specifying both new composite data structures and the applicable domains of methods. It was a goal of the CLOS design to allow methods to be defined on pre-existing Common Lisp types. To achieve this goal, CLOS supports the notion of classes separate from, but integrated with, Common Lisp types. In doing this, it became apparent that there were at least five different meanings of "type": *declaration* types, *representational* types, *signature* types, *predication types* and *methodical* types.

Declaration types are a feature of the program text. They specify an invariant for the program such as what values can be stored in a particular variable or structure. Such declarations allow humans to reason about the behavior of their programs. They allow linkers to check whether programs obey rules of program composition. Declarations allow compilers to partially validate program correctness and to make code optimizations.

A *representational type* is one that defines the storage layout of objects at run-time. Often a special run-time representation is used to provide an optimization in space or time for frequent operations. For example, a Lisp `fixnum` is a special representational type for small integers allowing them to be efficiently stored in a single word of memory. Since small integers are used quite frequently, this can be a big savings. When compiling an invocation of a generic arithmetic function, a Lisp compiler may emit code specialized to a more efficient type representation such as a fixnum based on static type inferencing.

A hierarchical representational type system can be defined where the leaves of the type hierarchy are basic representational types and the composite type nodes (interior node) are boolean combinations of subnodes. This is how the Common Lisp type system is defined, where the basic set of datatypes are the representational types. For example, the type `list` is defined to be the same as `(or null cons)`.

A *signature type* is one defined by the operations that can be performed on objects. Such a type is sometimes called an abstract data type. In this view, the primary distinction between a `cons` and a `number` is their signatures—given a `cons`, one can read and alter its `car` and `cdr` components; given a number, one can perform arithmetic calculations with it. Both integers and reals support a signature that includes the four basic operations (`+`, `-`, `*`, `/`).

A *methodical type* is one which can be used to specify the domain of applicability for a method. Methodical types are a feature of program texts. Classes in most object-oriented languages are the primary example of methodical types. In C++ the equivalent of methods (virtual functions) cannot be defined on base datatypes. Virtual functions can only be written for user-defined classes. Thus the C++ base datatypes are nonmethodical.

A *predication type* is one that can be distinguished at run-time by a type test. Compile-time only systems such as C++ provide no predication types, since there is no run-time type system. Common Lisp has extended the notion of type to allow general predications. Smalltalk does support finding out the class of an object at run-time, but provides no language support for doing type testing.

We can characterize these notions of type across two axes, as shown in the table below:

	program text feature	run-time feature
structure	declaration type defclass/defstruct	representation type predication type
operation	methodical type	signature type

These different meanings of type are not mutually exclusive. A type system based purely on representational types can also be a consistent signature type system, just as a methodical type system can be a valid signature type system. In fact, representational types are often designed to support specific operations. A methodical type system can also be based on storage layout (representational) decisions.

Some aspects of representational types will always be necessary, at least implicitly. For example, the mechanism by which the class of an object is determined almost always involves a selection based on representational information. Furthermore, objects usually can store data, and the mechanism for such storage and retrieval is representational.

There are three design dimensions regarding issues of classes and types: whether methodical types include system-defined types or only user-defined classes; whether new bit-pattern-level representations for objects can be defined along with methods on those new

representations; and whether declarations about methodical types can be used to optimize code.

With respect to the first dimension—the inclusion of system-defined types—Smalltalk, C++, and CLOS run nearly the entire gamut. In Smalltalk types and classes are identified; there is no semantic difference between those supplied by the system and those defined by the user. In C++, the only methodical value types—those for which the user can declare methods—are user-defined classes. In CLOS, types and classes are separate concepts; every class corresponds to a unique type, but not every type has a corresponding class. For example, many compound types (i.e., `(or cons integer)`) are not classes, but correspond (at most) to the union of several classes.) Most such compound types cannot be classes because there is no unique representation for such an object, and there is no intuitive for notion making an instance of one. Also, predication types in general do not correspond to classes, but merely to a filtering of some pre-existing type (or, set).

However, in order to cover all system-defined data types, CLOS defines a set of classes which *span* the pre-existing Common Lisp types; one set of types spans a second set of types if every object in the second set is a member of at least one type in the first spanning set. The CLOS spanning set is large enough to encompass most representational distinctions within Common Lisp implementations, but small enough that each system-defined data type is directly covered by a single spanning type. This allows implementors to retain their historic, low-level optimizations based on representational types. For example, in the spanning set of classes there is a class named `float` that corresponds to the type `float`. The type `float` has 4 subtypes, `short-float`, `single-float`, `double-float`, and `long-float`. Implementations of Common Lisp are not required to support distinct types corresponding to these subtypes of `float`, and therefore CLOS does not requires classes for them either.

But most importantly, the use of a spanning set of classes respects abstraction barriers by using the same syntax for polymorphic functions written on system-defined data types as for those written on user-defined classes.

With respect to the second goal—extensibility for new representational types—none of Smalltalk, C++, or Lisp allow the user to define representational types. For Lisp, there is no standard way to recognize a representational type. However, any particular implementation can add a new representational type provided it extends the system implementation for `class-of`. For example, in Symbolics CLOS a separate `flavors-class` is maintained as a metaclass, and `class-of` is able to distinguish Flavor instances from all other types and classes.

With respect to the third goal—the use of declarative types—C++ uses strong, static typing as much as possible, and Smalltalk uses only run-time typing. Common Lisp has a rich, extensible, but optional declaration syntax; although implementations are not required to do any static type-checking based on a program's use of type declarations, a number of compilers take full advantage of this feature, especially for optimization. For example, some Common Lisp compilers are able to achieve nearly the same performance as C on certain benchmarks [14].

2.2.1 Extensions

The possible type-oriented extensions to CLOS correspond to the distinct meanings of type: representational, signature, methodical, predication and declarative extensions.

A representational extension would provide support for users to define new representational types. This has two requirements. The first is specifying how the system can recognize this type. This might include patterns of the data itself, or using part of the data address to index into auxiliary tables, or some combination. The second requirement is to extend `class-of` (including the version of `class-of` used in method dispatch) so that for all representational types a class can be quickly determined. It would be nice if this extension of `class-of` could use a method-definition-like syntax, and specialize on representational types. The implementation of selection for such "representation" methods would of course not use the standard `class-of` itself.

What makes adding this extension difficult is that Lisp systems come with a large body of compiled code. That compiled code builds in certain assumptions about how the class can be found for any instance. Because `class-of` is crucial for the performance of the system, optimizations are used. Some of these optimizations would have to be undone to make an extended `class-of` work. This may be impossible because sources are unavailable for the average user, or at least inadvisable because the wrong kind of extension could slow the whole system even when the feature is not used. This is the same issue that prevents the functions that exist in Common Lisp from being made all generic functions— they have been defined and compiled with certain fixed assumptions that are difficult, if not impossible, to undo.

A *signature extension* would make signatures be first class objects, in some way similar to classes. No structural information would be associated with signature objects. They would carry the contract for the operations they support. For example, one could define a signature type for *stack*, one that supported at least the operations `push`, `pop`, `top`, and `empty`. A slightly more sophisticated notion of a signature would include constraints on the operations, such as `(pop (push x <stack>))` = `x`.

A signature type could be made a methodical type, for example, in defining a push-down automata, any object that can act like a stack would do.

```
(defmethod interpret-automata
    ((input string) (m state-machine) (s stack))
  ...)
```

Making signatures be first-class types is a useful step in separating implementation inheritance from notions of abstract data type inheritance[23].

Given this definition of signature type, there is a natural extension to a notion of inheritance for signature types. A derived subsignature is built by adding new operations to an existing signature, or generalizing existing operations; that is, a subsignature type would have additional operations or operations with domains that were supertypes of the domains of the supersignature type and ranges that were subtypes of the ranges of the supersignature type.

Making signatures be first-class is a step towards defining a formal notion of a protocol. Currently protocols are only defined informally (as in the CLOS MOP). One useful constraint on protocols might be that of limiting the signatures that classes supporting the protocol must support. However, signatures do not provide a sufficiently rich language to specify all that is needed for defining a protocol. In the MOP[18], we find other kinds of statements; for example, sometimes redefining one method of a protocol requires redefining other methods. Some methods can be only enhanced, not overridden. Some methods must call a specified generic function. No good language yet exists for specifying protocols whose constraints are at least as broad as in the CLOS MOP.

Although signature types are clearly conceptually useful, can they be added to CLOS without an unacceptable loss of performance? It may be an expensive operation to check if an object's class supports a set of operations. However, after first use of an object of a particular class, caching techniques like those described for PCL[17], can significantly reduce the cost. A simple check of class identity is sufficient to ensure applicability and specificity on repeated calls to these generic functions, even though the methods have a domain specified by signature.

Another natural extension that has been suggested for CLOS is to allow any set-theoretic combination of methodical types—a *methodical extension*. So for example, one could specify finite ranges of integers as a disjunction of EQL specifications. The problem in CLOS is that the built-in protocol for method combination (see next section) requires applicable methods to be sorted into a total order based on specificity of domain specialization of the methods. In fact, the constraints on allowed subclass and superclass relationships and the class precedence list algorithm are designed to facilitate natural, totally ordered class inheritance chains for any given class.

Because overlapping set combinations cannot be naturally totally ordered, some means of determining how to order the invocation of applicable methods must be found. One approach is to specify a means to totally order the classes including the set theoretic combinations—perhaps by user intervention. Another approach is to use method combination types that do not require a total order. A third is to specify that if there is no natural order, the behavior of the combined method must be independent of any arbitrary order chosen.

A *predication extension* would allow an arbitrary predicate to determine the applicability of a method. Although possible, this extension has the same issues of ordering as described above for the set-theoretic combinations. Even worse, caching methods as described above will not work, because an arbitrary predication (program) may have side effects, and even for the same instance, a method may be applicable one time and not another. However, it might be possible to partition the domain of the predicate such that the selected set is covered by only a few classes, and to implement a caching and/or dispatch scheme wherein only members of those classes fail to take advantage of the caching; members of the other classes would operate at a speed unaffected by the existence of such a predicate test.

Declarative extensions to CLOS would take compile-time advantage of method declarations for optimization. If there is a call to a generic function from within a particular method, and if there is sufficient declarative information about the arguments to the generic

function, the call to the generic function can be replaced by a direct call to the applicable effective method. This is like Self method unrolling [7]. Another kind of declarative extension is similar to that done by Bennett [6] where a global declaration that no more methods or classes will be defined can simplify global analysis. Such a global analysis can be used to produce an optimized run-time support system for delivery of applications.

2.3 Factoring Descriptions

Reuse is a major theme in current programming system circles. We used the term *factored description* for any program piece that can be used in more than one context. The simplest example of a factored description is a subroutine or function. A subroutine is an abstraction based on a reusable code fragment. This fragment is reused by being called or invoked dynamically from several different places. Abstraction is the process of identifying a common pattern with systematic variations; an abstraction represents the common pattern and provides a means for specifying which variation to use. For a subroutine, the means for variation are the supplied argument(s).

Classes and generic operations are the focus of factored descriptions in object-oriented languages. New classes can be defined in terms of previously defined classes, and the new descriptions can modify and augment these definitions. CLOS supports simultaneous inheritance from a number of previously defined classes.

The behavior of a class is determined by the set of applicable methods for that class. Behavior can either be inherited, overridden, or extended, and these qualities are independently determined for each message or generic function. When a particular behavior for a class is supplied by the superclasses of that class, we say that the behavior is inherited. When a particular behavior for a class is defined on the class and no behavior from superclasses is used in the definition of that particular behavior, we say that the existing behavior is overridden. When a particular behavior for a class is a combination of behavior defined specifically for the class and behavior inherited from superclasses, we say that the behavior is extended. CLOS provides mechanisms for all three, allowing both run-time and compile-time combinations.

2.3.1 Multiple Superclasses

Ideally, a description is factored into independent components, each of which is useful by itself. If a new class definition must directly depend on at most one other class definition, we say that the system uses *single inheritance*; if a new class can depend on more than one class we say that the language supports *multiple inheritance*. In single inheritance, direct combination of several classes is not possible: extension is possible only by subclassing. To combine behaviors from several classes, compound objects are typically created by storing instances of various classes into the components of a single object. This technique requires a visible layer of structure to be interposed between the client program and the components.

Multiple inheritance is a more direct solution to this problem of combining structural and behavioral features of previously defined classes. For example, suppose a class C is a compound derived from several other classes. If these other classes are in independent domains—share no names for commensurate attributes, either accidentally or deliberately—the resulting compound class has the union of all the components of each of the base classes. But if these classes are not independent, a means of handling conflicts is required. There are four approaches for addressing name conflicts: disallow them, select among them, form a disjoint union, or form a composite union.

The approach of disallowing conflicts is to signal an error if a name conflict occurs between commensurate attributes.

The approach of selecting among conflicts can be handled two ways. One is to require the inheriting class to specify an explicit choice. The second is to use encapsulation: The classes that are inherited from can include declarations that specify that certain attributes remain hidden even from subclasses. These two along with signaling an error are often used in concert, as in C++.

The approach of forming a disjoint union is to create a separate attribute for each same-named attribute. Attributes in the composite class will have to be disambiguated by some additional means beyond the normal names, perhaps by using an extended name that specifies the class contributing the attribute, either by fully naming that class with the extended name or by indicating the inheritance chain from which the desired attribute is inherited.

The approach of forming a composite union is to create a single attribute for each same-named attribute, by algorithmically resolving name clashes. The resolution algorithm must be designed to reflect the essential characteristics of the inheritance graph topology, and the features of the ancestor classes.

Smalltalk, which supports only single inheritance, signals an error if the same named instance variable is specified in a subclass.

With disjoint unions, there are as many copies of the conflicting item as there are superclasses with an item by that name. C++ uses this mechanism for member data elements. Only some of the named elements are visible in the C++ derived class, because C++ supports name encapsulation between subclasses and superclasses.

CLOS creates a single composite description for a slot from all the inherited slots of the same name. For some facets of slots, the resolution algorithm simply chooses the one definition directly found in the most specific superclass bearing such a slot; for other facets, it involves a union of all the inherited definitions; and ultimately, the resolution step is open to end-user tailoring. The inheritance of methods and the ordering of methods in a combined method also depend on the specificity of the classes involved. For this reason, CLOS defines an algorithm which constructs sets of linearizations on the inheritance graph, where each class has its own total precedence order—a unique linearization for each node in the inheritance hierarchy [10].

2.3.2 Inheriting, Overriding, and Enhancing Behavior

In CLOS, *a method is applicable to a set of arguments* if the actual arguments passed to the associated generic function are instances, direct or otherwise, of the classes specified for the parameters of that method. We use the term "direct instance" to refer to an instance of a class that is not an instance of any subclass of that class. We use the term "indirect instance" to refer to an instance of a class that is also an instance of some subclass of that class. We use this terminology because there appears to be no agreement about whether the term "instance" means "direct instance" or "instance, direct or indirect."

The meaning of being applicable to arguments can be seen in this example: the following method is applicable to the arguments `<3, ABC>` because `3` is an integer and `ABC` is a symbol:

```
(defmethod f ((i integer) (name symbol)) ...)
```

But the following method is not applicable because `3` is not a float:

```
(defmethod f ((i float) (name symbol)) ...)
```

The *behavior of an object* signifies the set of methods that are applicable to the object, for all generic functions, even when that object is only one of several arguments that participate in method selection. *Method inheritance* is also defined in terms of method applicability. For CLOS and Smalltalk, all methods defined in a class are applicable to subclasses. For C++, functions specific to a class are only applicable to a subclass if they are declared virtual. Also, even virtuals are not seen if those methods of a base class are declared private. C++ thus provides much more control on the inheritance of behavior to derived (sub)classes.

There are two problems with enhancing behavior—contextual reference to behavior and guaranteeing behavioral congruence. When behavior is extended, there must be a way to refer to the existing behavior, to invoke it at a reasonable point in the new behavior, and to guarantee that the overall semantics of the operation are preserved. For example, when + is defined on a class, it is reasonable to expect that the method implements something recognizable as, and congruent to, addition. Note that behavioral congruence is a problem with overriding as well.

There is no way to guarantee semantic congruence, but contextual reference to behavior can be handled. Reference may be made to the same operation as defined on subclasses (such as by invoking `inner` in Beta [19]), or to the same operation as defined on superclasses (such as by sending the message to the pseudo-variable `super` in Smalltalk). In C++, an inherited virtual can be invoked by calling it with a name qualified by the name of a base class through which it is inherited. In CLOS, invoking an operation inherited from one of the superclasses is normally done by using the special operator `call-next-method`, although method combination techniques extend this capability.

2.3.3 Method Combination by Roles

In CLOS, a method can be composed from subpieces that are designated as playing different roles in the operation through a technique known as *declarative method combination*. None of C++, Smalltalk, or Beta provide such declarative methods for combining method fragments. The enhancement techniques of Beta, C++, and Smalltalk are simple cases of *procedural method combination* as described earlier in the section on enhancing method behavior.

To understand the power of declarative combination, consider the following program that supports a fine-grained stream-opening protocol. It does this by calling a set of explicitly named generic functions whose appropriate methods will be provided by more specialized subclasses.

```
(defclass base-stream ...)

(defmethod open ((s base-stream))
  (pre-open s)
  (basic-open s)
  (post-open s))

(defmethod pre-open ((s base-stream)) nil)

(defmethod basic-open ((s base-stream)) (os-open ...))

(defmethod post-open ((s base-stream)) nil)

(defclass abstract-buffer ...)

(defmethod pre-open ((x abstract-buffer))
  (unless (empty-buffer-p x) (clear-buffer x)))

(defmethod post-open ((x abstract-buffer))
  (fill-buffer (buffer x) x))

(defclass buffered-stream (base-stream abstract-buffer) ...)
```

Notice that the method for `open` defined on `base-stream` provides a template for the operations on streams. The auxiliary methods `pre-open` and `post-open` defined on definitions on `base-stream` do nothing. They are only defined here to avoid calls to undefined methods.

Declarative method combination is an abstraction that makes it possible to supports patterns of method calls like this without the user having to build in these calls to auxiliary operations. In this example code there are four methods—`open`, `pre-open`, `basic-open`, and `post-open`. The main sub-operation, `basic-open`, cannot be named `open`, since that name refers to the whole combined operation. The other two names—`pre-open` and `post-open`—are placeholders for actions before and after the main one, i.e., those preparatory steps taken

before the main part can be executed, and those subsequent clean-up actions performed afterwards. There really is just one action—opening—and all other actions are auxiliary to it: they play particular roles. This constellation of actions should have just one name, and the auxiliary names need only distinguish their roles.

In declarative method combination, there are role markers that act like an orthogonal naming dimension. When a generic function is invoked, all applicable methods are gathered into roles; and then within roles, the methods are sorted according to class specificity. An effective method is then automatically constructed and executed, wherein each method plays its role.

Because some method combinations are so common, they are given names and definitions in CLOS. The most commonly used one is called *standard method combination*, which defines four method roles: primary methods for the main action, `:before` methods that are executed before the main action, `:after` methods that act after the main action, and `:around` methods that precede all other actions, and which optionally can invoke (via `call-next-method`) the sorted cluster of `:before`, `:after`, and primary methods.

The above example could be coded in CLOS as follows:

```
(defclass base-stream ...)

(defmethod open ((s base-stream)) (os-open ...))

(defclass abstract-buffer ...)

(defmethod open :before ((x abstract-buffer))
  (unless (empty-buffer-p x) (clear-buffer x)))

(defmethod open :after ((x abstract-buffer))
  (fill-buffer (buffer x) x))

(defclass buffered-stream (base-stream abstract-buffer) ...)
```

An important point to notice about this example is that the several methods of differing roles did not all come from the same superclass chains. The primary method is defined on `base-stream` and the auxiliary methods are defined on `abstract-buffer`. Buffers and streams are independent, and when they are brought together to form the compound object `buffered-stream`, the independent methods are brought together to define a compound method. Although this example does not show it, the `:before` and `:after` methods are often inherited from different classes at different levels in the class hierarchy. It is also important to note that use of declarative method combination also entails that `:before` and `:after` methods are inherited without the client having to be aware of inherited methods that should be used.

A *mixin* is a class designed to be used additively, not independently. Mixins provide auxiliary structure and behavior. The auxiliary behavior can be in the form of auxiliary

methods that are combined with the primary methods of the dominant class, or it can be in the form of new primary methods. The class named `abstract-class` above is a mixin.

The essential use of multiple inheritance is to provide reusable components; such components are combined to define new classes. The best use of multiple inheritance is to define pieces of structure and associated behavior that can be mixed in with other mainline classes to form subclasses that differ from their superclasses by the addition of auxiliary structure and behavior.

The stereotypical way that mixins are used extends beyond the composition of structure to the composition of auxiliary pieces of behavior. Patterns of behavior can be captured in code fragments such as shown in the definition of `open` above. Whenever we see patterns of use we define an abstraction to capture it (built-in method combination); and when we see the possibility that the user will also find patterns he wishes to capture, we define a user-accessible abstraction mechanism (`define-method-combination`). With the metaobject protocol, we extend this abstraction ability to programs, which can define method combination types and use them.

2.3.4 Extensions

CLOS mechanisms for factoring are a codification and simplification of a number of features that have been in long term use in Lisp-based object-oriented programming systems. For example, both Flavors and Loops supported multiple inheritance. Loops used the equivalent of `call-next-method`, and Flavors supported declarative method combination.

CLOS has no linguistic support for putting constraints on uses of classes defined as mixins, as was available in Flavors[26]. For example, it provided language support to specify that certain mixins were used with descendants of specified base classes, or with classes that support specified operations. The metaobject protocol can be used to simplify adding such a capability to a CLOS implementation.

CLOS has no mechanism for supporting encapsulation of names used in classes, except for the package system (package encapsulation does not affect inheritance, only visibility). For example, all slots named in a superclass will be found in a subclass, and if a subclass has a slot specified with the same name as one found in a superclass (even accidentally), only one such slot will exist in the subclass. For example, suppose one had the following classes defined in two different applications:

```
(defclass location ()
  ((location-x :accessor location-x)
   (location-y :accessor location-y)))

(defclass cad-element (location)     ; location on chip
  ...)

(defclass display-element (location) ; location on screen
  ...)
```

If a user now wanted to build a `displayable-cad-element` by inheriting from `cad-element` and `display-element`, an unintended conflict would occur. Such a conflict would not occur in C++ where two different copies of `location-x` and `location-y` would be created in such a subclass. Suppose each of `cad-element` and `display-element` had defined a slot `label` with the intended meaning that it was the string that labeled that element. These two should be merged. In C++, in the subclass of each of these classes, two of these slots would also appear.

No single default always works. Sometimes it is more appropriate to use composite objects rather than inheritance to achieve mixed behaviors; Stefik and Bobrow [25] discuss the tradeoffs among various techniques.

Common Lisp provides a mechanism that can hide the visibility of names—this mechanism is the package system. Some have suggested this as a mechanism for controlling these name conflicts. In the above example, the classes `cad-element` and `display-element` could be placed in separate packages, and the only symbols to be exported correspond to the public interfaces to those classes. The slot names would remain internal to the packages and therefore there would be no conflict between the two slots with apparently similar names located in different packages. In this case it does not have the desired effect because the name `location-x` is inherited twice from the same place, and hence the symbol is in the same package along each inheritance path.

The slot named `label` illustrates the opposite problem. If one separated these classes into different packages, two different slots each appearing to have the name `label` are created, with these symbols from different packages. A special declaration would have to be used to cause these two symbols to be identified. In general, packages are not a good solution to fine-grained visibility-hiding. In the worst case this might require one package per class; since the package system was designed as a module mechanism, not as a fine-grained encapsulation mechanism, use of a great many packages might lead to performance problems. In [18], code is shown for an extension to CLOS that would support appropriate encapsulated inheritance without use of packages.

A more radical extension would be a mechanism for disjoint multiple inheritance. One such mechanism, called *hybrid inheritance*, defines the notion of *domain* [12]. In this extension classes in separate domains are guaranteed to be independent, and rather than creating a class that represents combined classes, instances are created from classes in several domains: Such instances can be viewed as an instance of each class of which it is composed. For this extension the function `class-of` must be extended to take an optional second argument, the domain, and it returns the class of which it is an instance in that domain. Reference to inherited components is within domains. Method combinations within each domain operate independently. In fact, inheritance rules are local to each domain, so that each domain could have its own notion of object orientation.

The use of domains can be viewed by programmers as either providing a different model of multiple inheritance or providing multiple views of the same object. In the first case, there is a way to combine behavior provided by the different domains, which is a form of cross-domain method combination. In the second case the behavior is relatively independent,

and so there is a way to select the behavior provided by different domains.

In the above example of buffered streams, there would be no class representing buffered streams, but there would be instances that were instances both of `buffer` and of `stream`. The advantage of this approach is that it captures the essence of the disjoint union approach to combination while retaining the composition approach when constructing a combined class.

Another category of extension is enhancement of mechanisms for method combination. For example, a dynamic interpreter could be defined to interpret the roles of methods. Such an interpreter could be used to provide a backtracking mechanism that would search for methods that achieved certain goals. That is, the method combination mechanism is generalized beyond merely capturing patterns of use in the form of static roles to capturing more dynamically determined roles.

2.4 Reflection

Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: *introspection* and *intercession*. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called *reification*. The parts of the language that support these capabilities are referred to as the introspective and intercessory protocols.

There are four dimensions on which to explore reflection: structure, program, process, and development. *Structure* refers to the way that programs are put together. In an object-oriented setting, structure is reflected in the class hierarchy and the methods associated with each class. *Program* refers to the way the structure of the code effects behavior. This includes notions of qualified methods and their interaction with declarative method combination. *Process* refers to the way structure and program are interpreted to produce running systems. *Development* refers to how such reification supports incremental change and development of programs.

Within each dimension there are three points: implicit, introspective, and intercessory. When a quality is implicit, it is present in source code but is not represented as data in the language. When a quality is introspective, it is a first-class object that can be analyzed by a program. When a quality is intercessory, its representation can be altered, causing its behavior to change.

In some languages, such as C++, structure is only implicit in the source code. In these languages, source code is available for compilers and environments to examine, but typically the compiler and the environment are separate, external. In other languages, such as Smalltalk, CLOS, and LOOPS [1], the class hierarchy can be altered and dynamically extended by the running program.

With respect to the program dimension, language systems can be implicit, introspective, or intercessory. Implicit languages simply provide syntax and semantics for programs, and the compiler (or interpreter) and linker effect the semantics. Introspective languages provide a first-class representation for programs, and frequently they provide an operation that can convert data to program, such as `COERCE` in Common Lisp that converts a lambda expression to a Lisp function. Languages with intercessory capabilities have a *procedurally reflective interpreter*. When this interpreter is written in the same language it interprets, it is often called a *metacircular interpreter*. Altering or customizing parts of that interpreter can be done by writing programs in that language; this can change the semantics of some programs.

Lisp has traditionally supplied a means for programs to introspect, and to effect their own states, albeit sometimes in a limited sense. Lisp programs are encoded as *symbolic expressions*—the original base of data types for Lisp—so a program can construct other programs and execute them. A program can observe its own representation, and (heavens!) modify it on the fly.

Object-oriented programming offers an opportunity to migrate these notions from ad hoc mechanisms (e.g., "hooks" added to EVAL) to more principled ones. The first step in this direction is to require that classes be first-class—that each class be represented by a data object that can be passed as an argument to functions, held in program variables, and incorporated into data structures such as lists and the like. This is true for CLOS and Smalltalk, but not for C++. The second step is to require that each Lisp object be a direct instance of some unique class. Thus, each class itself is an instance of a class. Classes whose direct instances are themselves classes are sometimes called *metaclasses*.

In CLOS, generic functions and methods are also first-class—and hence are also instances of classes. The classes of generic functions, methods, and method combinations describe, at least to some extent, their structure and behavior. For example, there is the class `standard-generic-function`, whose instances are the CLOS generic functions; `#'print-object` is an instance of this class. These classes and objects are referred to as *metaobjects*. These metaobjects form a network that support introspection into the workings of CLOS. Note that the metaobject `standard-generic-function` and `#'print-object` are at two different metalevels. The latter is an object that reifies a specific function object, and the former is a class object that describes many metaobjects.

Object-oriented languages support incrementally extendible or specializable systems. The CLOS interpreter is an object-oriented program. Hence it supports extendibility. These extension capabilities are known collectively as the CLOS *metaobject protocol*. CLOS also supports dynamic changes of CLOS objects, such as dynamic redefinition of classes (without recompilation), and programmatic change of the class of existing objects while preserving their identity. The latter facilities are part of a *software development protocol*.

2.4.1 Metaobject Protocol

In CLOS, the details of facilities such as inheritance, method combination, generic function application, slot access, instance creation, and instance initialization are implemented as

if by a CLOS program whose structure and behavior can be observed (introspection) and affected (intercession).

By creating subclasses of specified metaobject classes and adding methods to generic functions specified in the metaobject protocol, standard behavior can be customized to a great degree. A particularly common customization of classes is to change how some local slots are to be stored—say, persistently in a database server rather than in computer memory. To implement persistent slots, a method applicable to instances of a user-defined subclass of `standard-class` overrides the standard method for `slot-value-using-class`, which is the metaobject generic function that determines how slot storage is stored into and retrieved from in memory, or wherever[22].

The exact mechanisms for metaobject programming are not part of the proposals for ANSI CLOS; but an informal, de facto standard is emerging out of the standardization process[18], supported in large part by the various commercial CLOS implementations.

A key aspect of intercession is that reflective capability not impose an excessive performance burden simply to provide for the possibility of intercession. What is not used should not affect the cost of what is used; and common cases should retain the possibility of being optimized. Even while supporting the development features described in the next section, several commercial CLOS implementations have very good performance.

2.4.2 Software Development

Different languages provide a variety of tools situated along a spectrum for aiding software development. At one end of the spectrum are languages like C++ in which the only tools are external development environments. In the middle are languages like Smalltalk that provide residential development environments that have access to every part of the language and its implementation, including, in some cases, its source code. At the other end of the spectrum are languages like CLOS that provide linguistic mechanisms to support development. The underlying nature of this spectrum is the degree to which the language provides mechanisms for introspection and intercession by the environment.

External development environments are not necessarily limited in power, but can provide incremental development and debugging capabilities similar in power to residential environments [11]. Residential development environments add to this the ability to use as libraries parts of the language and environment implementation.

Linguistic support for development can help with both incremental development and with delivery of applications. For example, CLOS supports a class redefinition protocol, a change-class protocol, and a mechanism to dump instances to a file and retrieve them. In addition, the metaobject protocol can be used to direct the analysis of application code, to provide flexibility during development, and to help achieve efficiency for delivery when that flexibility is no longer necessary. [6]

Software development environments can help solve linguistic problems. In CLOS a method may have several specialized arguments, and hence mention several different classes. Clearly, the definition of the method cannot be defined textually near each of those classes.

Furthermore, the methods of a generic function may be distributed over several different files, based on modularity issues arising out of the end application; hence one could not expect them all to be defined in a textually compact region. Environmental tools can be used to present related classes, methods, and generic functions in textually meaningful and compact ways, and made accessible from multiple points of view (e.g. methods as part of a generic function, or associated with a class).

2.4.3 Extensions

Lisp environments have traditionally been far in advance of environments for other languages. Editors, debuggers, compilers, trace facilities, and dynamic loaders are frequently part of a Lisp environment. Providing reflective capabilities for these components is a possible extension.

The flip side of reflection is encapsulation. As important as the ability to openly examine and modify a portion of a system is the ability to close the system once modified. Again, encapsulation is a useful extension in this context.

Some Lisp implementations provide a multitasking facility; others provide genuine parallelism. A possible extension to standard CLOS is to add a reflective layer on processes. For example, processes could be categorized into classes. Sets of processes could be treated as an object that can be manipulated. Subsets of these processes could be distinguished by their classes, and control code can be written whose behavior depends on the state of the set of processes or the states of any of the processes. When certain important states are achieved or events occur within the environment, specific generic functions could be automatically invoked by the system, and these generic functions may be specialized. For example, when a process terminates normally, the generic function `terminate`, which can be specialized, could be invoked by the system. [12] Facilities like this have been added to some Lisps, but not in a reflective style where the important parts of the state are reflected in metaobject classes, and where behavior can be specialized in an object oriented style.

There are two natural types of reflection, depending on the model of object orientation: operator-based reflection and object-based reflection. Operator-based reflection provides a mechanism to trap—perform user-specified reflection—whenever a particular operation is invoked, regardless of the objects on which it is invoked. Object-based reflection provides a mechanism to trap whenever any message is sent to a particular object, regardless of the operation. This duality seems to be inherent in the focus of the two systems.

Currently, reflection in CLOS is via generic functions, which can only be defined for methodical types. This is operator-based reflection. The mechanism for such reflection is the method selection and effective method construction machinery, which is defined on classes of generic functions and which are subject to end-user enhancement by using the metaobject protocol. Each generic function in such a class enables the reflective step at function invocation time (possibly doing nothing more than the normal, unreflective operations), because this is where the match up of arguments to the effective method occurs. In contrast, classical message-passing systems could easily be extended to provide object-based reflection

by having the message passing mechanism itself trap out on any object that is particularly marked for reflection; but they cannot as easily enable operation-invoked reflection as with generic functions in CLOS.

Another extension is garbage-collector-based finalization, which is a means to specify cleanup action that will be performed on an object when the last user reference to that object disappears. Finalization is often used to release storage and other resources that are tied to an object through means other than normal Lisp pointers. For example, finalization can be used to release an operating-system lock on a file when user code can no longer access the file handle; or finalization can be used to free up remote bitmap storage when a program drops all references to a network-based window. Most implementations of Lisp already support some internal, ad hoc notions of finalization for just these purposes—file-system and X-windows interfaces—but a few are beginning to support an end-user version of finalization by providing a protocol to register individual objects for final cleanup. The CLOS initialization and metaobject protocols make it possible to create classes whose instances would be subject to a particular finalization method, thus forming a dual to the `initialize-instance` protocol. By adding it at the reflective level, this would support object-oriented extensions of system-provided facilities.

2.5 Conclusions

We have seen how the confluence of several programming language themes have been realized in CLOS. We have attempted to place these ideas in the context of other similar ideas in other programming languages. We have also explored how the metaobject protocol might be used to adapt some additional features not in the kernel. Finally, we have looked at how a closer tie can be made between static program structure and the program development process in CLOS using the reflective capabilities of the language.

2.6 Acknowledgements

This chapter has benefited from careful reading and comments from Peter Wegner, Gregor Kiczales and Andreas Paepcke. The ideas here have been developed in concert with a large community of CLOS users, and the designers of the CLOS system. Residual mistakes and pomposity are solely the responsibility of the authors.

2.7 Biographies

Daniel G. Bobrow is a Research Fellow in the Systems and Practices Laboratory of the Xerox Palo Alto Research Center, and Manager of the Scientific and Engineering Reasoning Area. His current research interests include reasoning about the physical world, knowledge representation, and object-oriented programming.

At Xerox PARC, Bobrow has been involved in the production of such systems as KRL (one of the first knowledge representation languages), GUS (a "general" language understander), PIE (a personal information environment), Colab (an environment to support collaborative work), and Loops, Commonloops and CLOS (integrations of object-oriented programming with Lisp). He was a charter member of the ANSI Common Lisp Standard Committee (X3J13), and chair of the X3J13 Object Standard Subcommittee that designed CLOS.

Before Xerox , Dr. Bobrow taught for a year at MIT, and then joined Bolt, Beranek and Newman, Cambridge, Mass. where he started an Artificial Intelligence Department. He later became vice president of the BBN Computer Science Division. At BBN, he was involved in the development of Interlisp (then BBN Lisp), the TENEX operating system, and a number of natural language understanding systems

He is the Editor-in-Chief of the international journal, Artificial Intelligence, has been an editor of the Communications of the ACM, and on the editorial boards of a number of other professional publications. He has been President of the American Association for Artificial Intelligence, and Chair of the Cognitive Science Society. Bobrow received his bachelors in Physics from Rennselaer Polytechnic Institute, his masters in Applied Mathematics from Harvard University and his doctorate in mathematics from the Massachussetts Institute of Technology.

Dr. Richard P. Gabriel is Chief Technical Officer and principal founder of Lucid, Inc, a Unix software company specializing in object technology. He is a regular columnist for AI Expert. He is also Consulting Professor of Computer Science at Stanford University.

His interests include programming languages, programming environments, programming systems, Lisp systems, performance evaluation, and object-oriented programming.

Gabriel holds a B.A. in mathematics from Northeastern University an M.S. in mathematics from the University of Illinois, and a Ph. D. in computer science from Stanford University.

Jon L White, Principal Scientist and Lisp Technical Director at Lucid, Inc., holds a B.S. in Mathematics from Carnegie, and an A.M. in Applied Mathematics from Harvard University, where he did further graduate work in Computer Science. While working at the M.I.T. Artificial Intelligence Laboratory in the early 1970's, he developed and maintained MacLisp, a dialect of Lisp used widely on the PDP-10 computer by the AI community then. In the late 1970's, he began the development of NIL, a new implementation of Lisp, which became one of the several antecedents of Common Lisp. In the early 1980's, he participated in the development of Interlisp-D at Xerox's Palo Alto Research Center, and since 1985 has been working for Lucid in the development of its primary product—Lucid Common Lisp. Mr. White is the Editor of Lisp Pointers, an ACM SIGPLAN publication devoted to Lisp.

Bibliography

[1] Daniel G. Bobrow and Mark Stefik, *The LOOPS Manual*, Xerox Palo Alto Research Center, 1983.

[2] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, and David A. Moon, *The Common Lisp Object System Specification*, Technical Document 88-002R of X3J13, June 1988; also in *Special Issue of SIGPLAN Notices 23* September 1988 and *Lisp and Symbolic Computation*, January 1989.

[3] Daniel G. Bobrow, *Object-invoked procedural reflection*, Proceedings of the OOPSLA-90 Workshop on Computational Reflection, ACM 1990

[4] L. Brownston, R. Farrel, E. Kant, N. Martin. *Programming Expert Systems in OPS-5*, Addison Wesley, Reading MA 1985

[5] Rod Burstall and Butler Lampson, *A Kernel Language for Modules and Abstract Data Types*, Digital SRC Report Number 1, September 1, 1984. The authors attribute without reference the distinction between ad hoc and universal polymorphism to C. Strachey.

[6] James Bennett, John Dawes, Reed Hastings, *Cleaning CLOS Applications with the MOP*, presented at the Second CLOS Users and Implementors Workshop, OOPSLA 1989, October, 1989.

[7] Craig Chambers, David Ungar, E. Lee, *An Efficient Implementation of SELF, a Dynamically-typed Object-oriented Language Based on Prototypes*, Proceedings of OOPSLA 1989, October 1989.

[8] Luca Cardelli, and Peter Wegner, *On Understanding Types, Data Abstraction and Polymorphism*, Computing Surveys 17(4), pp. 471–522, 1985.

[9] William F. Clocksin and Christopher S. Mellish, *Programming in Prolog, Third Edition*, Springer-Verlag, 1987.

[10] R. Ducournau and M. Habib. *On Some Algorithms for Multiple Inheritance in Object-Oriented Programming*, Proceedings of ECOOP 1987, the European Conference on Object-Oriented Programming, Lecture Notes In Computer Science 276, Springer-Verlag, 1987.

[11] Richard P. Gabriel, Nickieben Bourbaki, Matthieu Devin, Patrick Dussud, David Gray, and Harlan Sexton, *Foundation for a C++ Programming Environment*, Proceedings of C++ at Work, September 1990.

[12] Richard P. Gabriel, *Ten Ideas for Programming Language Design*, Proceedings of the High Performance and Parallel Computing in Lisp Conference, November 1990.

[13] Richard P. Gabriel, *An Organization for Programs in Fluid Domains*, PhD Dissertation, 1981.

[14] Richard P. Gabriel, *Lisp: Good News; Bad News; How to Win Big*, AI Expert, Vol 6, no. 6, June 1991, pp. 30-39.

[15] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading MA 1983.

[16] Daniel H. H. Ingalls, *A Simple Technique for Handling Multiple Polymorphism*, the Proceedings of OOPSLA 1986; also a special issue of SIGPLAN Notices 21(11), November, 1986.

[17] Gregor Kiczales and Luis Rodriguez, *Efficient Method Dispatch in PCL*, this volume.

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

[19] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard, *The BETA Programming Language*, Research Directions in Object Oriented Programming, edited by Bruce Shriver and Peter Wegner, MIT Press, 1987.

[20] David A. Moon, *Object-Oriented Programming with Flavors*, Proceedings of OOPSLA 1986, also a special issue of SIGPLAN Notices 21(11), November 1986.

[21] David A. Moon, personal communication on another topic.

[22] Andreas Paepcke, *User-Level Language Crafting - Introducing the CLOS Metaobject Protocol*, this volume.

[23] Alan Snyder. *Encapsulation and inheritance in object-oriented programming languages* Proceedings of OOPSLA '86, also ACM Sigplan Notices 21(11) November 1986 pp 38-45

[24] Guy L. Steele Jr *Common Lisp the Language, Second Edition*, Digital Press, 1990.

[25] Mark Stefik and Daniel G. Bobrow, *Object-oriented programming: themes and variations* AI Magazine 6,4 p40-62 (Winter 1986)

[26] Symbolics, *Symbolics Common Lisp—Language Concepts*, Chapter 19 "Flavors", Symbolics, Inc., Burlington MA 1988.

[27] David Ungar and Randall Smith, *SELF: The power of simplicity*, Proceedings of OOPSLA 1987; also a special issue of SIGPLAN Notices 22(12), December 1987.

[28] Peter Wegner, *Dimensions of Object-Based Language Design*, the Proceedings of OOPSLA 1987; also a special issue of SIGPLAN Notices 22(12), December 1987.