CLOS: Integrating Object-Oriented and Functional Programming

Richard P. Gabriel
Lucid, Inc.

Jon L White
Lucid, Inc.

Daniel G. Bobrow
Xerox PARC

May 3, 2004

CR Categories: D.3.0, D.3.2, D.3.3

Lisp has a long history as a functional language, where action is invoked by calling a procedure, and where procedural abstraction and encapsulation provide convenient modularity boundaries. A number of attempts have been made to graft object-oriented programming into this framework without losing the essential character of Lisp—to include the benefits of data abstraction, extensible type classification, incremental operator definition, and code reuse through an inheritance hierarchy.

CLOS—the Common Lisp Object System[2], which is an outcome of the ANSI standardization process for Common Lisp—represents a marriage of these two traditions. We explore the landscape in which the major object-oriented facilities exist, showing how the CLOS solution is effective within the two contexts.

1 Incremental Definition of Operations

A *generic function*, or polymorphic function as it is sometimes called, is one whose implementation depends on the types of its arguments. That is, the code that is executed for a particular set of arguments is mechanically determined by the types of the arguments.

In strictly functional languages, an operation is defined by a single, monolithic piece of code; any argument-type conditionality is expressed as code explicitly programmed in by the user. In contrast, the CLOS notion of generic functions supports automatic selection from among separately defined, type-specific implementational parts. Yet, from the client's point of view, generic functions and traditional ordinary functions are called in the same way: The procedural abstraction barrier is still in force.

Although the concept of generic functions is familiar, its role as the foundation of an object-oriented language is relatively new. The use of generic functions rather than message-passing is suggested for a number of reasons. First, they 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. Third, they allow the programmer a means for defining operations for new types of objects, providing a desirable object-oriented extensibility. Fourth, they provide the framework for the class-based inheritance of behavior.

Separated, incrementally defined implementations with automatic choice of code has been used in a number of different programming styles: the polymorphism tradition, the data-driven tradition, the pattern-directed invocation tradition, and the object-oriented tradition. CLOS is closest to the object-oriented tradition, but shares some features of each of these. We will now examine each of these traditions, leading to an assessment of CLOS generic functions within this framework.

1.1 Polymorphism Tradition

Some languages have *polymorphic operators.*[4] A polymorphic operator is one whose behavior or implementation depends on a *description* of its arguments.

For example, FORTRAN has polymorphic arithmetic operators where the description is the types of its arguments. The types of the arguments are declared by type inferencing based on 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. For example, the code fragment `I+J` is a fixed point addition while `X+Y` is a floating point addition.

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 themselves. However, C++ is a language that supports separated definitions of polymorphic operations; operations of the same name which operate on arguments of different types are simply treated as independent. For example, one might have a function `size` that is defined to take an argument of type `string`, and another of the same name for an argument of the user-defined structure type `shoe`. The C++ compiler chooses the appropriate run-time implementation based on the declared or inferred types of the arguments in the calling code.

Even before CLOS, Common Lisp had some polymorphic operators. All arithmetic operators perform type-specific operations depending on the types of the actually supplied arguments, sometimes first performing type coercions. For example, evaluating the expression `(+ x y)` will perform floating point arithmetic if the value of either `x` or `y` is a floating point number.

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. However, Lisp supports a wide spectrum of choices between static and run-time typing. C++ supports a mixture of compile-time and run-time polymorphism—the operator implementation of an overload is selected at compile-time while the operator implementation of a virtual function is selected at run-time.

1.2 Data-driven Tradition

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. In some situations, it might be useful to use exogenous features for method selection. Some illustrative examples would include the process in which the operation is being done, the calling environment, or the types of returned results. Artificial intelligence applications could make use of extensions like this to be able to select an operation based on a description of the desired effect.

As an example of the data-driven technique, consider symbolic differentiation. Suppose that algebraic expressions are represented in Lisp as lists with prefix arithmetic operators:

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

The function `deriv` in Figure 1 could be the driver for a data-driven symbolic differentiation program (in Lisp syntax [12]) where each differentiation rule is implemented by one function in a table. The way to differentiate a sum is to sum the derivatives, and this behavior can be added quite easily as shown by the call to `add-to-table` in Figure 1. The data-driven tradition has been used extensively in symbolic algebra and artificial intelligence.

1.3 Pattern-directed Invocation Tradition

The pattern-directed invocation tradition is characteristic of backward chaining rule languages, such as Prolog [5]. 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. A program to differentiate an expression like `deriv(expression, variable,result)` might be written as shown in Figure 2.

Each clause is simply a relation that states that the left-hand side of the clause holds true if the right-hand side does. The control structure for 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

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

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

Figure 1: Table-driven Symbolic Differentiation

```
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).
```

Figure 2: Pattern-directed Symbolic Differentiation

attached code. Sometimes several left-hand sides match, or a single left-hand side can match more than one way, and by *backward chaining* or *backtracking* the first solution is found. Sometimes, to shorten the running time of a program, it is necessary to prune backtracking. There is an operator, called *cut*, that prevents 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, but few programs are run in such a setting. In many cases the ability to select a piece of code to run based on a matching process is powerful enough without requiring an automatic backtracking mechanism. If backtracking is necessary, it might be preferable to provide an explicit backtracking mechanism. Therefore it makes sense to separate the concept of selection from that of backtracking.

1.4 Object-Based Programming Tradition

The object-based programming tradition is often identified with message-passing, in which a message containing the name of an operation and argu-

```
class name                SymbolicSum
superclass                Object
instance variable names   s1
                          s2
class methods
instance creation
  of: first and: second
    ↑super new setargs: first and: second.
instance methods
  deriv: variable
    ↑SymbolicSum of: (s1 deriv: variable)
                 and: (s2 deriv: variable).
private
  setargs: first and: second
    s1←first. s2←second.
```

Figure 3: Object-oriented Symbolic Differentiation

ments is sent to a *receiver*. As with the data-driven tradition, the receiver and the name of the operation are jointly used to select a *method* to invoke.

For example, to do the symbolic derivative problem above in Smalltalk[8] one might have a class for SymbolicSum as shown in Figure 3.

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 only one point in a spectrum of object-based programming styles. There are four focuses for object-oriented systems: object-centric, class-centric, operation-centric, and message-centric. Object-centric systems provide the minimal amount of data abstraction, that is "state" and "operations"[15]—structure and behavior—with no special provision for classes, although 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

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

Table 1: Dimensions of Object-oriented Languages

classes or objects to describe inheritance and sharing. Wegner [15] uses the term object-oriented programming for object-based systems with classes that support inheritance.

Of these, only operation-centric systems are not message-passing based. As we will see, CLOS is a combination of class-centric and operation-centric.

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.

These four focuses can be divided into two independent axes which can be used to partition the space of languages and programming traditions, as summarized in Table 1.

The syntax of operation invocation in the two traditions reflects the difference in focus. In functional languages, the operation appears leftmost in a function call form. In object-oriented languages, the receiver object appears leftmost. In early mergers of object-oriented programming with Lisp, a `send` operation was introduced to perform message-passing; its first argument was the receiver object, and definitions looked much like those in Smalltalk. This contrasts starkly with how the differentiation example would be done in CLOS. The recursive call here is of the form `(deriv (first expression) variable)` which has the operation first (see Figure 4).

In the notation shown in Figure 4, the argument list for `deriv` contains the pair `(expression symbolic-sum)` which is a declaration that the first parameter for `deriv` is named `expression`, and this method is only applicable to instances, direct or otherwise, of the class `symbolic-sum`. 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.

```
(defclass symbolic-sum (standard-object)
  ((s1 :initarg :first :accessor first-part)
   (s2 :initarg :second :accessor second-part)))

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

Figure 4: Object-oriented Symbolic Differentiation in CLOS

1.5 Dimensions of Partitioned Operations

In the above descriptions of four traditions, several different dimensions characterize the design space of languages that support incremental definition of operations. We shall review each characteristic in turn.

The first is whether the selection of the implementation is at compile-time or at run-time. Common Lisp is an example of a language that has chosen to let the programmer decide where along this dimension to lie. C++ has overloads, whose implementations are selected at compile-time, but it also has virtual functions, whose implementations are selected at run-time.

A language is not object-oriented unless there is some form of run-time polymorphism.

The second dimension is whether polymorphic operators are defined only for system 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 user-defined polymorphic operators.

A language is not object-oriented unless there is a mechanism to define polymorphic operators.

The third dimension 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 on the entire argument spectrum (including the number of arguments); but, asymmetrically, virtual member functions provide (run-time) polymorphism only on their first arguments despite overloaded functions and virtual member functions having a uniform client syntax.

Multi-argument polymorphism can be programmed in message-passing systems, but at a cost. In such systems, one object must be the receiver of the message, and it is this object that determines the method selected. Selection based on several arguments can proceed by a sort of currying similar to the data-driven tradition outlined above. Typically, a series of messages is sent,

each further refining the choice of the real method. In one scheme, each object in the chain of argument objects provides a new "message" that captures the classes of arguments seen so far; the last object in the series selects the appropriate method based on the resultant compound message. [9]

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 dimension 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 operates only when the actual argument is `EQL` to the object in the parameter specification (these are called *eql specializations*.) Generalized selection is the heart of pattern-directed languages. Neither Smalltalk nor C++ supports selection on anything other than the types of arguments.

2 Classes and Types

CLOS supports the notion of classes separate from, but integrated with, Common Lisp types. There are at least four different meanings of "type": *declaration* types, *representational* types, *signature* types, and *methodical* types.

Declaration types are used to specify an invariant for the program about what values can be stored in a particular variable or structure. Such declarations allow compilers to reason about the correctness of programs and to make decisions about optimizations. Declaration types can be contrasted with the others, the *value* types, because they refer to the program text—a floating-point *variable* for example—while the value types refer to data objects.

A representational type is one that defines the storage layout of objects. Indeed, a design goal of such systems is to permit optimizations. For example, a Lisp `fixnum` is a representational type because it is defined to be efficiently stored in computer memory—typically in a single word of memory; and when compiling an invocation of a generic arithmetic function, a Lisp compiler may emit code specialized to a more efficient type representation based on static type inferencing. A hierarchical type system can be defined where the type leaves are purely representational and the composite type nodes are boolean and other simple combinations of subnodes. This is how the Common Lisp type system is defined; 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 cell, one can read and alter its `car` and `cdr` components; given a number, one can perform arithmetic calculations with it.

A methodical type is one for which methods can be written. Classes in most object-oriented languages are examples of methodical types.

These different meanings 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 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 new user-defined classes; whether new bit-pattern-level representations for objects can be defined along with methods on those new representations; and whether declarations can be used to optimize code.

With respect to the first goal—the inclusion of system-defined types—Smalltalk, C++, and CLOS nearly run 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 class behind it.

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 spanning set. This 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, there is a class named `float` in the spanning set of classes that corresponds to the built-in 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—neither Smalltalk, C++, nor Lisp allows the user to define represen-

tational types. For Lisp, there is no defined way to recognize a representational type.

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.

3 Factoring Descriptions

A *factored description* is one that can be used in more than one context. 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. The simplest example of a factored description is a subroutine or function. A subroutine is an abstraction based on a reusable code fragment, which is re-used by being called or invoked dynamically from several different places.

Classes 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 definitions from the used classes. Behavior, in terms of applicable functions, can either be inherited, shadowed, or enhanced. Inheriting behavior means that the behavior defined on superclasses is used as is. Shadowing behavior means that a new definition is directly supplied on the class being defined. Enhancing behavior means that new behavior is added in conjunction to inherited behavior.

3.1 Multiple Superclasses

Ideally, a description is factored into independent components, each of which is useful by itself. If each 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. Combined, compound objects can only be created by storing instances of various classes into the components of a single object—a technique that requires a visible layer of structure to be interposed between the client program and the components.

Multiple inheritance is a solution to this problem. If the classes are in independent domains—share no names for commensurate attributes, either accidentally or deliberately—the resulting compound object has the union of all the components of each superclass. If the 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 user to specify choices for every case. The second is to use encapsulation: The specification of a class includes those attributes that must remain hidden and thus automatically deselected.

The approach of forming a disjoint union is to create a separate attribute for each same-named attribute. Attributes in the composite will have to be disambiguated by some additional means beyond the normal names, perhaps by using a name that specifies the class contributing the attribute.

The approach of forming a composite union is to create a single attribute for each same-named attribute by algorithmically resolving them. 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 is to choose the one definition 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 depends 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. See the section "Determining the Class Precedence List" in [2], and refer to [6] for a discussion of linearization algorithms.

3.2 Inheriting, Shadowing, and Enhancing Behavior

The phrase *applicable to arguments*, said of a method, is used in CLOS to refer to the situation in which actual arguments are instances, direct or otherwise, of the classes specified for the parameters of the method. For 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))
  ...)
```

We speak of "the behavior of an object" to signify 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.

There are two problems with enhancing behavior—contextual reference to behavior and guaranteeing behavioral congruence. When behavior is enhanced, there must be a way to refer to the pre-existing behavior, to invoke it in a reasonable place in the new behavior, and to guarantee that the overall semantics of the operation are preserved. As an example of behavioral congruence, 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 shadowing 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 defined on subclasses (such as by invoking `inner` in Beta[11]), or to the same operation defined on superclasses (such as by sending the message to the pseudo-variable `super` in Smalltalk). In CLOS, invoking the shadowed operation in the superclasses is done by the special function `call-next-method`.

3.3 Method Combination by Roles

In CLOS, a method can be composed from sub-pieces that are designated as playing different roles in the operation through a technique known as *declarative method combination*. The enhancement techniques of Beta and Smalltalk are simple cases of *procedural method combination*: Explicit calls are made to related behavior. Another technique for procedural method combination is for a basic class to have a method that calls other explicitly named methods, which will be defined in derived classes. For example, consider a simple stream-opening protocol, shown in Figure 5. 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` have default definitions on `base-stream`, the base class, which do nothing.

Declarative method combination is an abstraction based on this sort of procedural method combination. In this example code there are four methods—`open`, `pre-open`, `basic-open`, and `post-open`. The main sub-operation, `basic-open`,

```
(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 (has-buffer-p x) (install-new-buffer x)))

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

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

Figure 5: Procedural Method Combination

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

Because some method combinations are so common, they are defaultly defined and given names 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

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

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

(defclass abstract-buffer ...)

(defmethod open :before ((x abstract-buffer))
  (unless (has-buffer-p x) (install-new-buffer x)))

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

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

Figure 6: Declarative Method Combination

other actions, and which optionally can invoke (via `call-next-method`) the sorted cluster of `:before`, `:after`, and primary methods.

The example in Figure 5 could be coded using method combination in CLOS as shown in Figure 6.

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 class `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.

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.

However, CLOS has no linguistic support for mixins, as Flavors does [13]. Such support would constrain the use of classes defined as mixins, or provide a means to hide names in a class so that, for example, slot names in mixin classes would not conflict with the slot names in the classes with which they are mixed.

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 *effectuation*. Introspection is the ability for a program to observe and therefore reason about its own state. Effectuation 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 state as data; providing such an encoding is called *reification*.

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 the class hierarchy. Program refers to the construction of code that effects behavior. Process refers to how structure and program are interpreted to produce running systems. Development refers to the support that such reification gives to the incremental change and development of programs.

Within each dimension there are three points: implicit, introspective, and effectual. 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 effectual, its representation can be altered, causing its behavior to change.

In some languages, like 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, like Smalltalk, CLOS and LOOPS[1], the class hierarchy can be altered and dynamically extended by the running program.

Language systems can be implicit, introspective, or effectual. 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 such as `FUNCTION` in Lisp to convert from data to program. Effectual languages provide something that resembles a *metacircular interpreter*, which is an interpreter (or executor) for the system written in that very system itself. Altering or customizing parts of that interpreter changes the semantics of 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, and 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 passed as an argument to functions, held in program variables, and incorporated into data structures such as lists and the like. The second step is to require that each Lisp object be a direct instance of some unique class. Thus, each class is (usually) an instance of some other class (but there will be a fixed-point class that is an instance, at least indirectly, of itself). Classes whose direct instances are themselves classes

are called *metaclasses*.

Generic functions and methods are also first-class, and are thus instances of classes; in fact, these classes form a network that provides introspection into the workings of CLOS. The objects that describe the class of generic functions, methods, and method combinations describe, at least to some extent, their structure and behavior. These classes and objects are referred to as *metaobjects*. Not all of these objects are metaclasses; for example, one metaobject is a class named `standard-generic-function`, whose instances are generic functions, not classes. Another metaobject is the function `#'print-object`, which is a generic function and not a class.

Object-oriented languages support incrementally extendible or specializable systems. CLOS moves further in this direction first by presenting the implementation of CLOS as a CLOS program, and second by supporting dynamic redefinition of classes and objects while still preserving the identity of existing instances.

The first step is a *metaobject protocol*, and the second is a software development protocol.

4.1 Metaobject Protocol

In CLOS, the details of inheritance, method combination, generic function application, slot access, instance creation, instance initialization, class redefinition, and altering an instance to belong to a different class, are implemented as if by a CLOS program whose behavior can be observed (introspection) and affected (effectuation), at least in principle.

By creating subclasses of metaobject classes and adding methods to metaobject generic functions, standard behavior can be customized to a great degree. A particularly common customization of classes that differs from the standard ones 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 user-defined metaclass would shadow the standard methods for `slot-value-using-class`, which is the metaobject function that determines how slot storage is laid out in memory, or wherever.

The metaobject protocol encompasses such things as the structure, representation, and allocation of classes, instances, generic functions, methods, and method combinations; and the definition and mechanisms behind inheritance, generic function invocation, method dispatch, class precedence, method applicability, method combination, and slots including slot access and slot inheritance.

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[10], and each CLOS implementation has at least some part of these mechanisms.

A key aspect of effectuation is that reflective capability not impose an excessive performance burden simply to provide for the possibility of effectuation.

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, some commercial CLOS implementations have very good performance. By using the reflective capabilities of CLOS, one commercial group developed technology to eliminate the overhead costs for CLOS's flexibility. [3]

4.2 Software Development

Different languages provide a variety of tools positioned 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. At the other end of the spectrum are language like CLOS that provide linguistic mechanisms to support development.

External development environments are not necessarily limited in power, but can provide incremental development and debugging capabilities similar in power to residential environments. [7]

Residential development environments add 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.

Software development environments can help solve linguistic problems. In CLOS a method may operate on several different classes, and the definition of the method cannot be defined textually near each of those classes on which it specializes. 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 such as editors, browsers, source locators, and graphers can be used to present related classes, methods, and generic functions in textually meaningful ways.

5 Conclusions

CLOS and its merger with functional programming has provided many lessons for the language designer and software developer. Its landscape can prove fruitful if examined carefully. During our brief tour of this landscape, we have seen

the various aspects of CLOS, and have attempted to draw connections between scattered parts of language design by connecting them through the ideas in CLOS.

References

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

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

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

[6] 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.

[7] 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.

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

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

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

[11] 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.

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

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

[14] David Ungar and Randall Smith, *SELF: The Power of Simplicity*, Proceedings of OOPSLA 1987, December 1987.

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

Biographies

Richard P. Gabriel is Chief Technical Officer of Lucid Inc, Consulting Professor of Computer Science at Stanford University, and lead guitarist of Bay Area rock band, The Wizards. His interests include programming languages, programming environments, programming systems, writing, and the blues.

Jon L White is Principal Scientist at Lucid, Inc., Technical Articles Editor of the Lisp Pointers newsletter, and general chair of the 1992 ACM Conference on Lisp and Functional Programming. His interests include programming languages, object-oriented systems, automated memory management (garbage collection), and computer arithmetic.

Daniel G. Bobrow is a Research Fellow in the System Sciences Laboratory at Xerox Palo Alto Research Center, and president of the American Association for Artificial Intelligence. His current research interests include programming language design, knowledge representation, and automating scientific and engineering reasoning.