The LOOPS Manual

(December, 1983)

by

Daniel G. Bobrow (Xerox PARC)
Mark Stefik (Xerox PARC)

Abstract. LOOPS adds data, object, and rule oriented programming to the procedure oriented programing of Interlisp. In object oriented programming, behavior is determined by responses of instances of classes to messages sent between these objects, with no direct access to the internal structure of an object. This approach makes it convenient to define program interfaces in terms of message protocols. Data oriented programming is a dual of object oriented programming, where behavior can occur as a side effect of direct access to (permanent) object state. This makes it easy to write programs which monitor the behavior of other programs. Rule oriented programming is an alternative to programming in LISP. Programs in this paradigm are organized around recursively composable sets of pattern-action rules for use in expert system design. Rules make it convenient for describing flexible responses to a wide range of events. LOOPS is integrated into Interlisp, and thus provides access to the standard procedure oriented programming of Lisp, and use of the extensive environmental support of the Interlisp-D system

Our experience suggests that programs are easier to build in a language when there is an available paradigm that matches the structure of the problem. The paradigms described here offer distinct ways of partitioning the organization of a program, as well as distinct ways of viewing the significance of side effects. LOOPS provides all these paradigms within a single environment. This manual is intended as the primary documentation for users of LOOPS. It describes the concepts and the programming facilities, and gives examples and scenarios for using LOOPS.

1 INTRODUCTION

Four distinct paradigms of programming available in the computer science community today are oriented around procedures, objects, data access and rules. Usually these paradigms are embedded in different languages. LOOPS is designed to incorporate all of them within the Interlisp programming environment, to allow users to choose the style of programming which best suits their application.

Procedure Oriented Programming: Lisp is a procedure oriented language; the procedure oriented paradigm is the dominant one provided in most programming languages today. Two separate kinds of entities are distinguished: procedures and data. Procedures are active and data are passive. The ability to compose procedures out of instructions and to invoke them is central to organizing programs using these languages. This is a major source of leverage in synthesizing programs. Side effects happen when separate procedures share a data structure and change parts of it independently.

Object Oriented Programming: This paradigm was pioneered by Smalltalk, and has its roots in SIMULA and in the concept of data abstraction. In contrast with the procedure-oriented paradigm, programs are not primarily partitioned into procedures and separate data. Rather, a program is organized around entities called objects that have aspects of both procedures and data. Objects have local procedures (methods) and local data (variables). All of the action in these languages comes from sending messages between objects. Objects provide local interpretation of the message form.

The object-oriented paradigm is well suited to applications where the description of entities is simplified by the use of uniform protocols. For example in a graphics application, windows, lines and composite structures could be represented as objects that respond to a uniform set of messages (i.e., `Display`, `Move`, and `Erase`). An important feature of these languages is an inheritance network, which makes it convenient to define objects which are *almost like* other objects. This works together with the use of uniform protocols because specialized objects usually share the protocols of their super classes.

Data Oriented Programming: In both of the previous paradigms, the invocation of procedures (either by direct procedure call or by message sending) is convenient for creating a description of a single process. In the data-oriented programming, action is potentially triggered when data are accessed. Data oriented programming makes use of long term storage of objects with implicit links from structures to actions.

Data oriented programming is appropriate for interfacing between nearly independent processes. A good example of this is the construction of a viewer for an independent traffic simulation process. The viewer provides a visual display of the changing traffic simulation process without affecting the code for the simulation. This independence means that the two processes can be written and understood separately. It means that the interactions between them can often be controlled without changing them.

Rule Oriented Programming: In rule oriented programming, the behavior of the system is determined by sets of condition-action pairs. These *RuleSets* play the same role as subroutines in the procedure oriented metaphor. Within a RuleSet, invocation of rules is guided largely by patterns in the data. In the typical case, rules correspond to nearly-independent patterns in the data. The rule-oriented approach is convenient for describing flexible responses to a wide range of events characterized by the structure of the data.

Our experience suggests that programs are easier to build in a language when there is an available paradigm that matches the structure of the problem. A variety of programming paradigms gives breadth to a programming language. The paradigms described here offer distinct ways of partitioning the organization of a program, as well as distinct ways of viewing the significance of side effects. LOOPS provides all these paradigms within the Interlisp environment [Xerox83]. In principle, the data-oriented programming

can be used with either the object-oriented or the procedure-oriented paradigms. In LOOPS, we have combined it only with variables in the object-oriented metaphor.

Summary: LOOPS adds data, object, and rule oriented programming to Interlisp. In object oriented programming, behavior is determined by responses of instances of classes to messages sent between these objects, with no direct access to the internal structure of an object. This approach makes it convenient to define program interfaces in terms of message protocols. LOOPS provides:

- inheritance of instance behavior and structure from multiple super classes
- user extendible property list descriptions of classes, their variables, and their methods
- composite objects - templates for related objects that are instantiated as a group.

Data oriented programming is a dual of object oriented programming, where behavior can occur as a side effect of direct access to (permanent) object state. This makes it easy to write programs which monitor the behavior of other programs. LOOPS provides:

- active values for object variables which can cause a procedure invocation on setting or fetching
- integration with facilities for long term storage of objects in shared knowledge bases
- support for incremental updates (layers), and the representation of multiple alternatives.

Rule oriented programming is an alternative to programming in LISP. Programs in this paradigm are organized around recursively composable sets of pattern-action rules for use in expert system design. Rules make it convenient for describing flexible responses to a wide range of events. LOOPS provides:

- a concise syntax for pattern matching and rule set construction
- use of objects as working memory for rule sets
- primitives for executing, stepping and suspending tasks based on ruleSets
- compilation of ruleSets into Lisp code for efficient execution

LOOPS is integrated into Interlisp. LOOPS provides:

- classes and instances as Interlisp file objects
- pseudoClasses to field messages to standard Interlisp datatypes

This manual is intended as the primary documentation for users of LOOPS. It describes the concepts and

the programming facilities, and gives examples and scenarios for using LOOPS.

1.1 Intellectual Precursors

LOOPS grew out of our research in a knowledge representation language (called Lore) for use in a project to create an *expert assistant* for designers of integrated digital systems. Along the way, we discovered that we needed to experiment with alternative versions of the representation language. A core of features was identified that we wanted to keep constant in our experiments. This core became a data and object-oriented programming system with many features not found in other available systems. Many of the features (e.g., active values, data bases, and composite objects) were motivated by the needs of our project, but we they would be useful for many other applications. LOOPS has been sufficiently useful and general that we decided to make it available outside of our group.

The design of LOOPS owes an intellectual debt to a number of other systems, including:

(1) Smalltalk ([Goldberg82], [Goldberg81], [Ingalls78]), which has pioneered many of the concepts of object-oriented programming.

(2) Flavors [Cannon82], which supports this style of programming in the MIT Lisp Machine environment and which confronted non-hierarchical inheritance.

(3) PIE [Goldstein80], which provided facilities for incremental, sharable data bases.

(4) KRL [Bobrow77], which explored many issues in the design of frame-based knowledge representation languages and which provoked much additional work in this area.

(5) UNITS [Stefik79], which provided a substantial testbed for experiments in problem solving that have guided our decisions about the importance of several language features.

(6) EMYCIN [VanMelle80] which showed the power of rule oriented programming for building expert systems.

While all of these languages provided ideas, none of them was quite right for our current needs. For example, Smalltalk supports only hierarchical inheritance and does not have a layered data base, active values, or property lists on variables. PIE and KRL are not easily supportable or extendable. Flavors does not run on the machines available to us. UNITS was the closest existing language to our needs, but we wanted to change many of its features. Since we have compared these languages and traced the intellectual history elsewhere [Bobrow82], we will not pursue that further in this document.

In designing LOOPS, we wanted a general inheritance mechanism, a way of attaching access-triggered procedures to variables, a way of instantiating composite objects recursively, and a way of creating permanent databases of objects that can be shared and updated incrementally.

In tension with the desire for extensive language features was a desire to keep LOOPS small so that it would be easy to understand and to implement. To this end we have tried to create a small repertoire of powerful features that work well together.

1.2 Acknowledgments

from the LOOPS Manual:

Thanks to Alan Bell, Harry Barrow, Harold Brown, Gordon Foyster, Phil Gerring and Gordon Novak, Chris Tong, Schlomo Weiss, Terry Winograd and the other members of the KBVLSI project (past and present) for bug reports and suggestions, and for enduring the wait for it to mature into existence while so many things have been pressing. Special thanks to Johan de Kleer for extensive discussions of design issues, and to Richard Fikes, Adele Goldberg, Danny Hillis, Dan Ingalls, and Gordon Novak for comments on earlier drafts of this manuscript. We are grateful to Larry Masinter and Bill Van Melle for help on the integration of LOOPS with Lisp, and to the Interlisp-D group for unfailing support and encouragement. Thanks also to Lynn Conway for encouraging this work and to the Xerox Corporation for providing the intellectual and computing environments in which it could be done.

from the Rules Manual:

Special thanks to Danny Berlin and Lynn Conway for many suggestions and for the patience it takes to be the first real users of something new. Sanjay Mittal and Terry Winograd offered helpful criticisms and advice on the documentation and concepts of the rule language. Larry Masinter and Bill van Melle have provided substantial support in the entire Loops enterprise with Interlisp-D. Thanks to the Xerox Corporation and George Pake of Xerox PARC for providing the stimulating environment and computational facilities that made this work possible.

1.3 References

[Aiello81] Aiello, N., Bock, C., Nii, H. P., White, W. C., *AGE Reference Manual.* Technical Report, Heuristic Programming Project, Computer Science Department, Stanford University, October 1981.

[Bobrow82] Bobrow, D. G., & Stefik, M. J. Introducing new programming metaphors to LISP. (submitted to *Communications of the Association for Computing Machinery*).

[Bobrow80] Bobrow, D. G., & Goldstein, I. P. Representing design alternatives. *Proceedings of the AISB Conference*, Amsterdam, 1980.

[Bobrow77a] Bobrow, D. G., & Winograd, T. An overview of KRL, a knowledge representation language, *Cognitive Science* 1:1, 1977, pp 3-46.

[Bobrow77b] Bobrow, D. G., & Winograd, T. Experience with KRL-0, one cycle of a knowledge representation language, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Mass. August, 1977, pp 213-222.

[Cannon82] Cannon, H. I. Flavors: a non-hierarchical approach to object-oriented programming, *personal communication*, 1982.

[Consumers80] Anon, Washing Machines. *Consumer Reports*, November 1980, pp. 679-684.

[Erman81] Erman, L. D., London, P. E., Fickas, S. F. The design and an example use of Hearsay-III. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, August 1981, pp. 409-415.

[Fain81] Fain, J., Gorlin, D., Hayes-Roth, F., Rosenschein, S., Sowizral, H., Waterman, D. *The ROSIE Reference Manual*, Rand Note N-1647-ARPA, Rand Corporation, December 1981.

[Feigenbaum78] Feigenbaum, E. A., The art of artificial intelligence: themes and case studies of knowledge engineering, *AFIPS Conference Proceedings* 47 National Computer Conference, 1978, pp. 227-240.

References

[Forgy81] Forgy, C. L. *OPS5 User's Manual.* Technical Report CMU-CS-81-135. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, July 1981.

[Goldberg82] Goldberg, A., Robson, D., Ingalls, D. *Smalltalk-80: The language and its implementation.* Reading, Massachusetts: Addison-Wesley (in press).

[Goldberg81] Goldberg, A. Introducing the Smalltalk-80 System, *Byte* 6:8, August 1981.

[Goldstein80] Goldstein, I. P., & Bobrow, D. G. Extending object oriented programming in Smalltalk. *Proceedings of the Lisp Conference*, Stanford University, 1980.

[Ingalls78] Ingalls, D. H. The Smalltalk-76 programming system: design and implementation. *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp 9-16.

[Maytag] Anon. *Operating Instructions for Model A510.* Printed by the Maytag Company, Newton Iowa 50208.

[Stefik82] Stefik, M., Aikins,5 J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., Sacerdoti, E. The organization of expert systems: a tutorial. *Artificial Intelligence*, 18:2, March 1982, pp. 135-173.

[Stefik79] Stefik, M. An examination of a frame-structured representation system. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Tokyo, Japan, August 1979, pp. 845-852.

[VanMelle80] Emycin ... To be filled in

[Weinreb81] Weinreb, D., & Moon, D. *Lisp Machine Manual*, Massachusetts Institute of Technology, 1981

[Xerox83] *Interlisp Reference Manual*, Xerox Palo Alto Research Center, October, 1983.

2 OVERVIEW

2.1 Structure of Classes and Instances

Classes: A class is a description of one or more similar objects. An *instance* is an object described by a particular class. Every object within LOOPS is an instance of exactly one class. Classes themselves are instances of a class, usually the one called `Class`. Classes whose instances are classes are called *metaclasses*.

Variables: LOOPS supports two kinds of variables - class variables and instance variables. Class variables are used to contain information shared by all instances of the class. A class variable is typically used for information about a class taken as a whole. Instance variables contain the information specific to an instance. Both kinds of variables have names, values, and other properties. A class describes the structure of its instances by specifying the names and default values of instance variables. For example, the class `Point` might specify two instance variables, `x` and `y` with default values of `0`, and a class variable, `lastSelectedPoint`, used by methods associated with all instances of class `Point`. LOOPS also allows "variable length" classes, which have some instance variables that are referenced by numerical index.

Methods: A class specifies the behavior of its instances in terms of their response to *messages*. The class associates *selectors* (LISP atoms) with *methods*, the Interlisp functions that respond to the messages. All instances of a class use the same selectors and methods. Any difference in response by two instances of the same class is determined by a difference in the values of their instance variables. For example, `PrintOn` is used as a selector for the message which knows how to print out a representation of an object on a file.

Properties: LOOPS provides user-extendible property lists for classes, their variables, and their methods. Property lists provide places for storing documentation and additional kinds of information. A property list on a variable is used to store additional information about both the variable and its value. For example, in a knowledge engineering application, a property list for an instance variable could be used to store such information as *support* (i.e., reasons for believing a value), *certainty factors* (i.e., numeric assessments of degree of belief), *constraints* on values, *dependencies* (i.e., relationships to other variables), and *histories* (i.e., previous values).

Metaclasses: Classes themselves are instances of some class. When we want to distinguish classes whose instances are classes, we call them metaclasses, after the Smalltalk usage. When a class is sent a message, its metaclass determines the response. For example, instances of a class are created by sending the class the message `New`. For most classes, this method is provided by the standard metaclass for classes: `Class`. The user can create other metaclasses to perform specialized initialization. The metaclass for `Class` itself (called `MetaClass`) contains the `New` method for making classes. Another useful metaclass provided in the system is `AbstractClass`. It is used for classes that are placeholders in the inheritance network that it would not make sense to instantiate. Its response to a `New` message is to cause an error.

```
[DEFCLASS AreaBudget
   (MetaClass Class    EditedBy (* dgb "15-Feb-82 14:32 ")
              doc
(* * This is a sample class chosen to illustrate the syntax
 of classes in LOOPS.  Commentary on the class is inserted
```

```
in a standard property in the class.  -- e.g. Budgets are ...))
  (Supers OwnedObject Budget)
  (ClassVariables (maxBase 25000))
  (InstanceVariables
     (owner #$VLSI doc (* organizational area that owns budget) )
     (base 1000 doc (* The initial amount of money))
     (overhead 2.25 doc (* Multiplied by base to get total.))
     (employees NIL doc (* list of employees in this area))
     (manager NIL doc (* manager of this area))
     (total #(SHARED getTotal UpdateNotAllowed)
        doc (* value of total is computed using active value.))
  (Methods
     (Report AreaBudget.Report doc (* Prints out a budget report))
     (StoreBase AreaBudget.StoreBase
        doc (* store base value checking maxBase))]
```

Figure 1. Example of a class definition in LOOPS. The class, called `AreaBudget`, inherits variables and methods from both of its super classes (`OwnedObject` and `Budget`). The form of the definition here does not show inherited information, only the changes and additions. In this example the new class variable `maxBase` is introduced, and six instance variables (`owner`, `base`, `overhead`, `employees`, `manager`, and `total`) are defined. The `Methods` declaration names the Interlisp functions that implement the methods. For example, `AreaBudget.Report` is the name of a function that implements the `Report` method for instances of `AreaBudget`.

2.2 Inheriting Variables and Methods

Inheritance is an important tool for organizing information in objects. It enables the easy creation of objects that are "almost like" other objects with a few incremental changes. Inheritance avoids the user have to specify redundant information and simplifies updating, since information that is common need be changed in only one place.

LOOPS objects exist in an *inheritance network* of classes. An object inherits its instance variable description and message responses. All descriptions in a class are inherited by a subclass unless overridden in the subclass. For methods and class variables, this is implemented by a runtime search for the information, looking first in the class, and then at the super classes specified by its *supers list*. For instance variables, no search is made at run time; default values are cached in the class, and are updated if any super is changed, thus maintaining the same semantics as the search. Each class can specify inheritance of structure and behavior from any number of super classes in its supers list.

Hierarchy: In the simplest case, each class specifies only one super class. If the class `A` has the supers list `(B)`, a one element list containing `B`, then all of the instance variables specified local to `A` are added to those specified for `B`, recursively. That is, `A` gets all those instance variables described in `B` and all of `B`'s supers. In this case one obtains strict inheritance hierarchy as in Smalltalk.

Any conflict of variable names is resolved by using the description closer to `A` in traversing up the hierarchy to its root at the class `Object`. Method lookup uses the same conflict resolution. The method to respond to a message is obtained by first searching in `B`, and then searching recursively in `B`'s supers list. An example of this is given in figure 2.

Class	Super	InstanceVariables	Methods
Object	NIL	none	(s4 M6)
C	Object	(w 7)	(s2 M4) (s3 M5)
B	C	(y 4) (z 3)	(s1 M2) (s2 M3)
A	B	(x 1) (y 0)	(s1 M1)

Figure 2. In the definitions given in the above chart, an instance of A would be given four instance variables, w, y, z, and x in that order. The default value for y would be 0, which overrides the default value of y inherited from B. The instance would also respond to the four messages with selector s1, s2, s3, and s4. The method used for responding to s1 is M1, which is said to override M2 as the implementation of the message s1. Similarly, M3 overrides M4 as the implementation of message s2. Notice that the root class in the system, Object, has no super class. All classes in the system are subclasses of Object, directly or indirectly.

Multiple Super Classes: Classes in LOOPS can have more than one class specified on their supers list. Multiple super classes admit a modular programming style where (i) methods and associated variables for implementing a particular feature are placed in a single class and (ii) objects requiring combinations of independent features inherit them from multiple supers. If D had the supers list (E A), first the description from E and its supers would be inherited, and then the description from A and its supers. In the simplest usage, the different features have unique variable names and selectors in each super. In case of a name conflict, LOOPS uses a depth-first left to right precedence. For example, if any super of E had a method for s1, then it would be used instead of the method M1 from A. In every case, inheritance from Object (or any other "common" super class) is only considered after all other classes on the recursively defined supers list.

2.3 Data Oriented Programming – Using Active Values

In data oriented programming, one needs a way of specifying for any variable of an object whether any special procedure is to be invoked on read or write access, and if so which. In LOOPS we check on every variable access whether the value is marked as an *active value*. If so, the active value specifies the procedures to be invoked when the value of a variable (or property) is read or set. This mechanism is dual to the notion of messages; messages are a way of telling objects to perform operations, which can change their variables as a side effect; active values are a way of accessing variables, which can send messages as a side effect. The following notation for active values illustrates its three parts:

#(*localState getFn putFn*)

This notation is converted by a read macro into an instance of the LISP data type activeValue. The *localState* is a place for storing data. The *getFn* and *putFn* are the names of functions that are applied with standard arguments when a program tries to get or put the value of a variable. Every active value need not specify both a *getFn* and a *putFn*. If the *getFn* is NIL, then a get operation returns the local state. If the *putFn* is NIL, then a put operation replaces the local state.

Active values enable one process to monitor another one. For example, we have developed a LOOPS debugging package that uses active values to trace and trap references to particular variables. Another

example is a graphics package that updates views of particular objects on a display when their variables are changed. In both cases, the monitoring process is invisible to and isolated from the monitored process. No changes to the code of the monitored object are necessary to enable monitoring.

Model/View Controller Example: figure 3 shows an application of this to a simulation model. Suppose that we want a program that simulates the flow of traffic in a city and displays selected parts of the simulation on a screen. Active values enable us to divide the programming of this example into two parts: the traffic model and the view controller. The traffic model consists of objects representing automobiles, traffic lights, emergency vehicles, and so on. These objects exchange messages to simulate traffic interactions (e.g., when a traffic light turns green, it would send `Move` messages to start cars moving). The view controller provides windows into different parts of the city. It contains information about how the objects are to be displayed. We want a user to be able to move these windows around to change the view.

```
(DEFINST Automobile-1 ...
   (InstanceVariables
      (position #(Pos1 NIL UpdateDisplay)
         displayObjects (DispObj1 DispObj2 DispObj3)
         doc (* position of car in traffic coordinate system))
      (speed 25))
   ...]
```

Figure 3. Instance of an automobile in a traffic simulation model. Other classes describe such things as traffic lights, city blocks, and emergency vehicles. Instances of these classes exchange messages while simulating the vehicles moving around in the model. The instance variable `position` is used to record the location of an automobile in the traffic coordinate system. In this example, an active value in `position` is used to update view objects that control pictures of the traffic patterns on an interactive display. Whenever a simulation method puts a new value into the `position` variable, the procedure `UpdateDisplay` sends update messages to each object in a list of view objects. These messages ultimately cause the graphics display to be updated.

In figure 3, there is an active value in the `position` variable of an instance of `Automobile`. This active value is the interface between the object in the simulation model and the view controller. Whenever a method in the simulation model changes the value of a `position` variable, the procedure `UpdateDisplay` in the *putFn* of the active value is invoked. `UpdateDisplay` updates the local value and sends a message to each of the view objects in the list stored as a property of `position`. These objects respond to a message by updating the view in the windows on the display screen. The important point of this example is that it shows how the view controller can be invoked as a side effect of running the simulation. The view can be changed without effecting any programs in the simulation model. To change the set of simulation objects being monitored, only the interface to the view controller needs to be changed by adding active values. The objects in the view controller may also be changed (e.g., to reflect changes to relative coordinates of the window and the traffic model).

2.4 Knowledge Bases

LOOPS was created to support a design environment in which there are community knowledge bases that people share, and to which they can add incremental updates. We have chosen the term *knowledge base* instead of *data base* to emphasize the intended application of LOOPS to expert systems. In expert systems, knowledge bases contain inference rules and heuristics for guiding problem solving. This is in

contrast to the tabular files of facts usually associated with data bases.

Knowledge Bases: Knowledge bases in LOOPS are files that are built up as a sequence of layers, where each layer contains changes to the information in previous layers. A user can choose to get the most recent version of a knowledge base (that is, all of the layers) or any subset of layers. The second option offers the flexibility of being able to share a community knowledge base without necessarily incorporating the most recent changes. It also provides the capability of referring to or restoring any earlier version. figure 4 illustrates this with an example.

```
------------------------- Layer 1 -------------------------
Obj1 (x 4) ...
Obj2 (y 5) (w 3) ...
------------------------- Layer 2 -------------------------
Obj2 (y 7) (w 2) ...
Obj3 (z 6) ...
------------------------- Layer 3 -------------------------
Obj1 (x 8) ...
Obj4 (z 9) ...
```

Figure 4. Knowledge bases in LOOPS are files that are built-up incrementally as a sequence of layers. Each layer contains updated descriptions of objects. When a knowledge base is opened, the information in the later layers overrides the information in the earlier layers. LOOPS makes it possible to select which layers will be used when a knowledge base is opened. In this example, if the knowledge base is opened and only the first 2 layers are used, then `Obj1` will have an `x` variable with value 4. If all three layers were connected, then the value would be 8.

Community Knowledge Bases: LOOPS partitions the process of updating a community knowledge base into two steps. Any user of a community knowledge base can make tentative changes to a community knowledge base in his own (isolated) environment. These changes can be saved in a layer of his personal knowledge base, and are marked as associated with the community knowledge base. In a separate step, a data base manager can later copy such layers into a community knowledge base. This separation of tasks is intended to encourage experimentation with proposed changes. It separates the responsibility for exploring possibilities from the responsibility of maintaining consistent and standardized knowledge bases for shared use by a community. The same mechanisms can be used by two individuals using personal knowledge bases to work on the same design. They can conveniently exchange and compare layers that update portions of a design.

Unique Identifiers: The ability to determine when different layers are referring to the same entity is critical to the ability to share data bases. To support this feature the LOOPS data base assigns unique identifiers (based on the computer's identification numbers, the date, and an unbounded count) to objects before they are written to a knowledge base. This facility provides a grounding for more sophisticated notions of equality that might be desired in knowledge representation languages built on LOOPS.

Environments: A user of LOOPS works in a personalized *environment.* An environment provides a lookup table that associates unique identifiers with objects in the connected knowledge bases. In an environment, user indicate dominance relationships between selected knowledge bases. When an object is referenced through its unique identifier, the dominance relationships determine the order in which knowledge bases are examined to resolve the reference. By making personal knowledge bases dominate over community knowledge bases, a user can override portions of community knowledge bases with his own knowledge bases.

Multiple Alternatives: An important use of environments is for providing speedy access to alternative versions (e.g., multiple alternatives in a design). A user can have any number of environments available at the same time. Each environment is fully isolated from the others. Operations that move information between environments are always done explicitly through knowledge bases.

3 CREATING AND USING OBJECTS

In the LOOPS implementation of object-oriented programming, there are three types of objects: Instances, Classes, and Metaclasses. Instances are used like data objects in Lisp; they are commonly created, passed around, and modified by procedures (although all objects can be). Classes and metaclasses are objects which "define" a group of objects that are "instances of" that class or metaclass. The difference between classes and metaclasses is that the instances of a class are instances, and the instances of a metaclass are classes—all comments about classes apply to metaclasses, except where otherwise stated.

Note that the word "instance" is used in two separate ways: the phrase "instance of" refers to the relation between any object and the class (or metaclass) that "defines" it. The noun "instance" is only used to refer to those objects which are instances of classes.

A class contains information about instance variables, class variables, and methods. Instance variables are local variables stored within each instance of the class. Class variables are variables stored within the class object, accessable from each instance of the class. Methods are procedures which are used to perform operations on instances of the class.

Each Class also contains a list of other classes called "super classes" or "supers". The super class list provides a mechanism for inheriting instance variables, class variables, and methods from other classes (see page 31).

This section first describes how to create and use objects. Next, "sending a message" (the standard way to invoke a method). Next, creating and using new instances. Next, defining and editing new classes. Finally, defining a new method for a class.

3.1 Sending a Message to an Object

Operations in LOOPS are invoked by sending messages. Sending a message to an object invokes a method (from the class that the object is an instance of) to execute the operation. Messages are sent using the function ← as follows:

(← *object Selector* $arg_1 \cdots arg_N$) [NLambda NoSpread Function]

Sends the message *Selector* to the object *object* with the arguments $arg_1 \cdots arg_N$. *Selector* is always implicitly quoted (i.e., not evaluated); the remaining arguments are evaluated.

object must be an "internal pointer" to the object. The internal pointer to the object with the LOOPS name `FOO` can be extracted by the form `($ FOO)`.

Note: `SEND` can be used instead of ←. The arrow notation, although less mnemonic, is usually used to make expressions shorter and hence easier to type and read.

If it is necessary to *compute* the selector, one can use the function ←!, which is just like ← except that it also evaluates its *Selector* argument.

Example:

```
(← ($ PayRoll) PrintOut file1)
```

This sends a `PrintOut` message to the class `PayRoll` (with a single argument: the value of the Intrerlisp variable `file1`).

3.2 Creating a New Instance

To create an instance of a particular class, one sends the message `New` to the class:

(← *class* `New`) [Message]

Returns a new instance of the class *class*.

In the usual case, initial values for instance variables are taken from the instance variable descriptions associated with the class. LOOPS provides some other ways to exercise control over the initialization of values in instances (see page 34).

3.3 Naming and Pointing to Objects

In order to manipulate a LOOPS object, it is necessary to have a pointer to it. One way to do this is to save a pointer to the object in an Interlisp variable, for example:

```
(SETQ myVariable (← ($ Transistor) New))
```

This creates a new instance of the `Transistor` class, and stores a pointer to this instance in the Interlisp variable `myVariable`. Pointers to instances can also be saved in instance variables.

LOOPS objects may be passed around and examined by Lisp functions. The following function is useful:

(`Object?` *X*) [Function]

Returns *X* if it is a LOOPS objects, otherwise `NIL`.

Another way to manipulate an object is by giving it a unique "LOOPS name". An object can be given a LOOPS name by sending it the message `SetName`

(← *object* `SetName` *name*) [Message]

Sets the LOOPS name *name* to refer to *object*. LOOPS names are unique in a LOOPS environment; the name is assigned in the environment specified by the global variable `CurrentEnvironment` (see page 41 for a complete description of environments).

If an attempt is made to assign a name already in use in the environment, and the global flag `ErrorOnNameConflict=T`, an error is generated. If `ErrorOnNameConflict=NIL`, and there is already an object *oldObject* with that name, the name is unset for *oldObject* and set for *object* without generating an error.

For example, if `I1` is an Interlisp variable whose value is a pointer to some instance, the object can be given the LOOPS name `Foo` as follows:

```
(← I1 SetName 'Foo)
```

After naming `I1` this way, the user can refer to this object as `($ Foo)`, which returns the object whose name is `Foo`.

The user can refer to an object with a *computed* LOOPS name using the form (`$!` *EXPR*). For example, if the value of the lisp variable `X` is the atom `Apple`, then (`$! X`) = (`$ Apple`).

Classes having `NamedObject` (see page 115) as a super class inherit an instance variable, `name`, that contains the name of the objects. Instances of these classes can be named, as before, with a `SetName` message, or alternatively as a side effect of setting the `name` instance variable.

Class objects are automatically given a LOOPS name when they are created, as described below.

3.4 Defining a New Class

The way one creates a new class is to send the message `New` to a metaclass. Usually, the metaclass named `Class` is used.

(← *metaClass* `New` *className supersList*) [Message]
Returns a new instance of the metaclass *metaClass*. *className* is the new class name and *supersList* is a list of the names of the super classes for this new class. If the list of super class names is omitted, *supersList* defaults to (`Object`).

Example:

```
(← ($ Class) New 'StudentEmployee '(Student Employee))
```

This defines a new class, `StudentEmployee` as a subclass of the known classes named `Student` and `Employee`.

An abbreviated way of defining a class is to use the function `DC`:

(`DC` *className supersList*) [Function]
("define class") Sends the class `Class` an appropriate `New` message:

(← (`$ Class`) *className supersList*)

Example:

```
(DC 'StudentEmployee '(Student Employee))
```

This specifies that the class `Student` is to be used recursively, inheriting both from `Student` and all its supers, and from `Employee` and all its supers.

After defining the class, one can modify its structure by editing the textual source for the class with `EC`:

(`EC` *className* —) [Function]
("edit class") `EC` envokes the Interlisp editor on the textual source for the class named *className*.

The editor can also be envoked by sending the `Edit` message: (← (`$` *className*) `Edit`).

For example, (`EC 'StudentEmployee`) might start the editor editing the expression:

```
[DEFCLASS StudentEmployee
```

```
    (MetaClass Class Edited: (* lc: "18-Oct-82 14:26"))
    (Supers Student Employee)
    (InstanceVariables)
    (Methods]
```

One can then change this to:

```
[DEFCLASS StudentEmployee
    (MetaClass Class Edited: (* lc: "18-Oct-82 14:26"))
    (Supers Student Employee)
    (InstanceVariables
        (sponsor NIL doc (* Name of sponsor))
        (stay     3 doc (* number of months here)))
    (Methods]
```

Leaving the editor successfully at this point would install the two instance vararible descriptions in the class `StudentEmployee`. Then, in addition to those instance variables `StudentEmployee` inherited from `Student` and `Employee`, each instance would also have two new ones, `sponsor` and `stay` with default values of `NIL` and 3 respectively. A more extensive description of editing and changing classes is found in section 13.4.

3.5 Defining a Method

In order to define a method for a class, one can use the Interlisp function `DM`:

(`DM` *className selector argsOrFnName form*) [Function]

Defines a method for the class named *className* that can be called using the selector *selector*. If *form* is non-`NIL`, then *argsOrFnName* is interpreted as the list of arguments for a function, and *form* as the body of that function. If the first element of the list *argsOrFnName* is not `self`, then `self` is added on the front. `DM` defines a function whose name is the concatenation of *className*, a period, and *selector*. For example, `Class.List` is the function name created for the `List` selector in the class `Class`. The function definition is created by substituting into (`LAMBDA` *argsOrFnName . form*).

If *argsOrFnName* and *form* are `NIL`, `DM` creates a skeleton definition for the function and puts the user into the Interlisp editor, editing the skeleton.

If only *form* is `NIL`, *argsOrFnName* is interpreted as the name of a function to be used for implementing the method.

Note: a method can also be defined by sending the `DefMethod` message to the class: (← *class* `DefMethod` *selector argsOrFnName form*).

Example:

```
(DM 'Number 'Increment '(self)
    '((* incr my IV) (←@ :myValue (ADD1 (@ :myValue)))))
```

This defines a method with selector `Increment` for the class `Number` which adds 1 to the instance variable `myValue` (the `@`-notation for accessing variables is described on page 18). This form results in

the definition of a function named `Number.Increment` as follows:

```
(DEFINEQ
   (Number.Increment
      (LAMBDA (self)       (* incr my IV)
         (←@ :myValue(ADD1 (@ :myValue)))]
```

(EM *className selector* —) [Function]

Calls the Interlisp editor to edit the method for the class named *className* associated with the selector *selector*.

Often it is more conveniently to use the LOOPS browser to edit the code for a method (see page 102).

Example:

To edit the method from the example above, one could type:

```
(EM 'Number 'Increment)
```

This will edit the method of class `Number` which responds to the selector `Increment`, whether or not it has a name of the standard form.

4 OBJECT VARIABLES AND PROPERTIES

There are two kinds of variables associated with an instance: its private *instance variables* and the *class variables* that it shares with all instances of the class. This section deals with the functions for getting and putting values, and with a compact programming notation for referring to these variables from inside functions that implement methods. In addition, there are properties which are associated with instance variables and class variables, with the methods of a class, and with classes themselves. Given an object or a class, one can fetch or set any of these properties. This section describes the functions for accessing all of these properties and values.

4.1 Access Expressions

As mentioned above, there are a number of different types of variables and properties that can be associated with each class. However, most of the accessing operations (getting and putting) in methods refer to the values or properties of instance variables or class variables of an instance. LOOPS provides general functions (described later) for accessing these values, allowing variable names and property names to be computed. However, most of the time the programmer knows the variable and property name to be used, and writing calls to these functions can be cumbersome.

Therefore, a simplified notation has been introduced for writing many common accessing operations, which is translated into calls to the appropriate functions:

(@ *object accessExpr*) [Macro]
(@ *accessExpr*) [Macro]

Returns the variable or property value of the object *object* as specified by *accessExpr*. Note that *accessExpr* is not evaluated; *object* is evaluated.

If only one argument is given to `@`, it is assumed that the object is bound to the variable `self`. This is very useful because by convention the first argument to any method is named `self`.

(←@ *object accessExpr newValue*) [Macro]
(←@ *accessExpr newValue*) [Macro]

Similar to `@`, sets the value of the variable or property specified by *accessExpr* (unevaluated) in the object *object* to *newValue*. Returns *newValue*. Note that *accessExpr* is not evaluated; the other arguments are evaluated.

Like `@`, if *object* is ommitted, it defaults to the value of the variable `self`.

Both `@` and `←@` take the argument *accessExpr*, which is an "access expression" which specifies exactly which variable or property value should be retrieved or set. *accessExpr* is an atom which specifies a variable name, an optional property name, and whether the variable is an instance variable or a class variable.

Some examples:

`(@ :FOO)` Retrieve the value of instance variable `FOO` (from the object that is the value of `self`).

`(@ XX ::FOO)` Retrieve the value of class variable `FOO` (from the object that is the value of `XX`).

(←@ ::FOO:,BAR 5)
Store 5 as the value of the BAR property of class variable FOO (of the object that is the value of self).

4.2 Getting Variable and Property Values

The functions GetValue and GetClassValue retrieve from an instance the values of variables or their properties. If the value bound to an instance variable or class variable is an *active value* with a *getFn*, then GetValue and GetClassValue of these functions trigger the *getFn* (see page 25).

(GetValue *object varName propName*) [Function]
Returns the value or property value of the instance variable *varName* in the object *object*. Each instance of a class has its own separate set of instance variables.

If *propName* is NIL, GetValue returns the value of the variable. In proper usage, *object* is an instance and the local value of the variable is returned. If no local value has been set, GetValue returns the default value from the class. Since this is a common case, default values inherited from super classes of the class are cached in the class itself, thus avoiding a runtime search.

If *propName* is not NIL, GetValue returns the value associated with the property named *propName* of the variable *varName*. If none is found in the instance, it returns the default property value found in the class or one of its super classes. If no property value is found in any of the super classes, the default value used is the value of the global variable NotSetValue (currently bound to ?). Note: this is different from Interlisp, where if no value of a property is found, then NIL is returned.

GetValue fetches a value from an *instance* of a class. It is an error to try to use GetValue to fetch an instance variable from a class. To fetch the default value of an instance variable from a class, use GetClassIV (see page 22).

(GetClassValue *object varName propName*) [Function]
Returns the value (if *propName*=NIL) or property value of the class variable *varName* for the class of the *object* (which may be either an instance or a class).

Class variables are inherited from the super classes. If *object* is an instance, lookup begins at the class of *object* since instances do not have class variables stored locally. If the class does not have a class variable *varName*, GetClassValue searches through the super classes of the class until it finds *varName*. Since this is thought to be an relatively rare in code, class variables are stored only in the class in which they are defined, and the runtime search is necessary.

Conceptually, one should think of a class variable of a class as being shared by all instances of that class, and by all instances of any of its subclasses. For example, suppose Transistor is a class with class variable, TransSeqNum, and DepletionTransistor is a subclass of Transistor. Then setting the class variable TransSeqNum from an instance of DepletionTransistor would be seen by all instances of Transistor.

4.3 Putting Variable Values and Property Values

`PutValue` and `PutClassValue` are functions used for storing variable or property values in an instance. They are analogous to `GetValue` and `GetClassValue`; as with these functions, if the value of the variable or property is an active value with a *putFn*, trying to store a value for that variable or property will invoke the *putFn* (see page 25).

(`PutValue` *object varName newValue propName*) [Function]

Stores *newValue* as the value or property value of the instance variable *varName* in the object *object*. Returns *newValue*.

If *propName* is `NIL`, `PutValue` stores *newValue* as the value of *varName* in *object*. If *propName* is non-`NIL`, then *newValue* is stored as the value of the property *propName* of the instance variable *varName*.

For example, (`PutValue` *pos* `'X 0`), stores `0` as the value of the instance variable `X` of the object *pos*.

`PutValue` works for storing values in an instance of a class. It is an error to try to store a default instance variable in a class with `PutValue`. To store the default value for an instance variable directly in the class, use `PutClassIV` (see page 22).

(`PutClassValue` *object varName newValue propName*) [Function]

Similar to `PutValue`, except it stores *newValue* as the value or property value of a class variable and property. *object* may either be an instance or a class. Returns *newValue*.

If *varName* is not local to the class, then the value will be put in the first class in the inheritance list that *varName* is found.

The following functions push a value on the front of a list already stored in a variable:

(`PushValue` *object varName newValue propName*) [Function]
(`PushClassValue` *object varName newValue propName*) [Function]

`PushValue` and `PushClassValue` add *newValue* on the front of the list that is the value of the indicated variable or property, and store the result back in the variable or property.

These functions are defined so that if the value accessed is an active value, the *getFn* will be triggered when the old value of the list is fetched, and the *putFn* when the new value is stored back (see page 25).

The following function adds a value on the end of an instance variable list:

(`AddValue` *object varName newValue propName*) [Function]

Similar to `PushValue`, except that *newValue* is added to the *end* of the variable list.

There is no function for adding values to the end of class variable lists.

4.4 Non-triggering Get and Put

Using active values (page 25), it is possible to associate functions with a variable (or property) that will be called whenever the variable (or property) is read or set. In some cases, it is useful to be able to access a value from an instance or class variable without triggering any active value which might be stored. This can be done using the following functions:

`(GetValueOnly` *object varName propName*`)` [Function]
`(GetClassValueOnly` *object varName propName*`)` [Function]
`GetValueOnly` and `GetClassValueOnly` retrieve the value of instance variables and class variables, respectively, without triggering any active values.

`GetValueOnly` retrieves the default value from the class if none exists in the instance.

To store a value without triggering any active values, the following functions are provided:

`(PutValueOnly` *object varName newValue propName*`)` [Function]
`(PutClassValueOnly` *object varName newValue propName*`)` [Function]
These functions store *newValue* in the instance variable or class variable, without triggering any active values, and return *newValue*.

Note that `GetClassValueOnly` and `PutClassValueOnly` can take either a class or an instance. `GetValueOnly` and `PutValueOnly` will only take instances.

4.5 Local Get Functions

Sometimes it is desirable to find out if a value or property is set in a particular class or instance, without inheriting any information which is not local, and not activating any active values. This can be done with the following functions:

`(GetIVHere` *object varName propName*`)` [Function]
object must be an instance. Returns the instance variable value that is found in the instance; if none is found, then returns the value of the global variable `NotSetValue` (initially `?`).

`(GetCVHere` *object varName propName*`)` [Function]
object must be a class. Returns the class variable value that is found in the class; if none is found, then returns the value of `NotSetValue`.

In both `GetIVHere` and `GetCVHere`, if the value is an active value, the actual active value is returned, without being triggered.

Note that there are no need to have special local put functions, since all put functions are local to the instance or class. For local nontriggering storage functions, use `PutValueOnly` and `PutClassValueOnly` (page 21).

4.6 Accessing Class and Method Properties

Most of the get and put functions described in the preceding sections work with instances, but not with classes. Some exceptions are `GetClassValue`, `PutClassValue`, `GetClassValueOnly`, and `PutClassValueOnly`, which can take either an instance or a class, and access class variables, and `GetCVHere` which takes a class.

The following functions access the *default* value or property value of an instance variable (which is stored in the class):

`(GetClassIV` *class varName propName*`)` [Function]
Returns the *default* value or property value of the instance variable *varName* in the class *class*.

`(PutClassIV` *class varName newValue propName*`)` [Function]
Stores *newValue* as the *default* value or property value of the instance variable *varName* in the class *class*. If *varName* is not local to the class, this will cause an error. Returns *newValue*.

Note: `GetClassIV` and `PutClassIV` do not trigger active values (page 21).

LOOPS provides property list storage for classes themselves and for methods of classes. A typical use of these properties is to document a class and its methods. Like the put and get functions for variables, these functions can trigger active values. The functions for class properties are:

`(GetClass` *class propName*`)` [Function]
Returns the value of the property *propName* of *class*. If *propName* is `NIL`, `GetClass` returns the metaclass of *class*.

Class properties are inherited like class variables, so `GetClass` will search through the super classes of *class* if *propName* is not found in *class* itself.

`(PutClass` *class newValue propName*`)` [Function]
Sets the value of the property *propName* of *class* to *newValue*. If *propName* is `NIL`, `GetClass` sets the metaclass of *class* to *newValue*.

`(GetClassOnly` *class propName* —`)` [Function]
`(PutClassOnly` *class newValue propName*`)` [Function]
These functions are analogous to `GetClass` and `PutClass`, except that they never trigger active values.

`(GetClassHere` *class propName*`)` [Function]
Returns the local value of the property *propName* of *class*. If *propName* is not found locally, `GetClassHere` returns the value of the global variable `NotSetValue` (initially ?).

The functions for accessing method properties are:

`(GetMethod` *class selector propName*`)` [Function]
If *propName* is `NIL`, `GetMethod` returns the method (Interlisp function name) which implements the message *selector* of the class *class*. If *propName* is non-`NIL`, it returns the value of the property *propName* of the method.

Method properties are inherited: the retrieval process involves searching through super classes of *class* if the property is not found in *class* itself.

(PutMethod *class selector newValue propName*) [Function]
If *propName* is NIL, PutMethod sets the method which implements the message *selector* of the class *class* to *newValue*. If *propName* is non-NIL, it sets the value of the property *propName* of the method to *newValue*. Returns *newValue*.

(GetMethodOnly *class selector propName*) [Function]
(PutMethodOnly *class selector newValue propName*) [Function]
Analogous to GetMethod and PutMethod except that they never trigger active values.

(GetMethodHere *class selector propName*) [Function]
Returns the local value of the property *propName* the the method which implements the message *selector* of *class*. If *propName* is not found locally, GetMethodHere returns the value of the global variable NotSetValue (initially ?).

All of the above functions only work directly on classes, not on instances of those classes. In addition, if a method or class variable is inherited, then the put functions change the property in the class in which the method or class variable is found in the supers list, not in the class which was the argument of the put function.

4.7 General Get and Put Functions

The following functions are generalized get and put functions which accept a type argument and invoke the more specialized functions:

(GetIt *object varOrSelector propName type*) [Function]
(PutIt *object varOrSelector newValue propName type*) [Function]
(GetItOnly *object varOrSelector propName type*) [Function]
(PutItOnly *object varOrSelector newValue propName type*) [Function]
(GetItHere *object varOrSelector propName type*) [Function]
For all of these functions, the value of the *type* argument can be one of IV, CV, CLASS, or METHOD for instance variable, class variable, class, or method, respectively. If *type* is NIL, IV is assumed. The argument *varOrSelector* is interpreted as a variable name if *type* is IV or CV, a selector name if *type* is METHOD, and is ignored if *type* is CLASS.

These functions are interpreted as follows:

```
(GetIt ... 'IV)  ==>  (GetValue ...)
(GetIt ... 'CV)  ==>  (GetClassValue ...)
(GetIt ... 'CLASS)  ==>  (GetClass ...)
(GetIt ... 'METHOD)  ==>  (GetMethod ...)
```

The other functions are similar.

Note: Actually, if *type*=IV, these functions will call different functions depending on whether the object is a class or instance.

4.8 Summary of Get and Put Functions

In the following table, * indicates that no function is available.

	Inherit/Trigger	Inherit/DontTrigger	Local/DontTrigger
from instances:			
Get/Put fns for instance variables	`GetValue / PutValue`	`GetValueOnly / PutValueOnly`	`GetIVHere`
Get/Put fns for class variables	`GetClassValue / PutClassValue`	`GetClassValueOnly / PutClassValueOnly`	*
from classes:			
Get/Put fns for instance variables	*	*	`GetClassIV / PutClassIV`
Get/Put fns for class variables	`GetClassValue / PutClassValue`	`GetClassValueOnly / PutClassValueOnly`	`GetCVHere`
Get/Put fns for class properties	`GetClass / PutClass`	`GetClassValueOnly / PutClassValueOnly`	`GetClassHere`
Get/Put fns for method properties	`GetMethod / PutMethod`	`GetMethodOnly / PutMethodOnly`	`GetMethodHere`

5 ACTIVE VALUES

Active values provide a way of invoking procedures when the value of a variable (or property) is read or set. This mechanism is dual to the notion of messages: messages are a way of telling objects to perform operations, which can change their variables as a side effect; active values are a way of accessing variables, which can send messages as a side effect. This section presents the notation for creating active values. Then, the concept of nested active values is introduced. The nesting property enables many of the important applications of active values by supporting composition of the access functions. Next is described how to use active values as the default values in a class, and how to share them. Finally, the standard arguments to active value access functions are described, along with LOOPS functions that can be used in user-defined access functions.

5.1 Active Values Notation

The notation for an active value illustrates its three parts:

#(*localState getFn putFn*)

This notation is converted by a read macro into an instance of the Interlisp data type `activeValue`. The *localState* field is used as a place for storing data. The *getFn* and *putFn* are the names of functions that are applied with standard arguments when a program tries to get or put the value of a variable whose value is an active value. Every active value need not specify both a *getFn* and a *putFn*. If the *getFn* is `NIL`, then a get operation returns the local state. If the *putFn* is `NIL`, then a put operation replaces the local state.

5.2 Nested Active Values

Often it is desirable to associate multiple access functions with a variable. For example, we may want more than one process to monitor the state of some objects (e.g., a debugging process and a display process). To preserve the isolation of these processes, it is important that they be able to work independently. LOOPS uses nested active values as a way of composing these functions.

Nested active values are arranged so that the innermost active value is stored in the *localState* of the penultimate *localState*, and the outermost active value is the immediate value of the variable. Put operations to a variable through such nested active values trigger the *putFn*s in sequence from the outermost to the innermost. For example, suppose the variable tracing facility were used to trace access of the `position` variable from the model/view controller example (page 10). The resulting active value would look like

```
#( #(Pos1 NIL UpdateDisplay) GettingTracedVar SettingTracedVar)
```

An attempt to set the position variable would cause the function `SettingTracedVar` to be called with the new value as one of its arguments. `SettingTracedVar` would operate and call the LOOPS function `PutLocalState` to set its own *localState*. This, in turn, would trigger the inner active value causing `UpdateDisplay` to be invoked.

Get operations work in the opposite order. If there are three nested active values, a request to get the value will cause the innermost *getFn* (if any) to run, followed by the middle *getFn* (if any), followed

by the outermost *getFn* (if any) whose value is returned by the get operation. Each *getFn* sees only the value returned by the next nested *getFn*, and the innermost *getFn* sees the value stored in its localState.

LOOPS provides functions for embedding and removing active values from variables. This idea of functional composition for nested active values is most appropriate when the order of composition does not matter. We have resisted the development of other combinators for the functions using the same parsimony arguments that we used earlier about specializing and combining methods. Just as inheritance from multiple super classes works most simply when the super classes describe independent features, active values work most simply when they interface between independent processes using simple functional composition. Any more sophisticated control is seen as overloading the active value mechanism. The escape for more complex cases is to combine the implicit access functions using Interlisp control structures to express the interactions.

5.3 Active Values as Default Values

Suppose that I is an instance of a class with an instance variable V, whose default value is the active value A. Further suppose that the value of V in the instance I has never been set. The first time (PutValue I V *exp*) is invoked, a copy of A is made. This copy is inserted in the instance itself as the the value of the instance variable, with pointers to the same contents as A. Then the *putFn* is invoked, with the copy as the *activeVal* argument; this copy of A provides a place where local state can be stored private to I.

In some cases, one knows that the *putFn* will not actually write into the active value, and therefore the active value which is the default could be shared instead of needing to be copied. To indicate this, the *localState* of A should be made the atom Shared. In the example below, the user knows that no change will be made in A itself and thus uses a shared active value.

Example: SUM is a class with three instance variables, top, bottom, and sum; top and bottom start with default values of 0, and sum is to be computed when asked for. One cannot update sum independently.

```
[DEFCLASS SUM
     (MetaClass Class)
     (Supers Object)
     (InstanceVariables
          (top 0)
          (bottom 0)
          (sum #(Shared ComputeSum NoUpdatePermitted))
     (ClassVariables)
     (Methods
          (printOn PrintColumn)]
```

The method for printOn used in this example, and the *getFn*, ComputeSum, and the *putFn*, NoUpdatePermitted, are Lisp functions whose definitions are not shown here. NoUpdatePermitted is available as part of the kernel.

5.4 Standard Access Functions

LOOPS provides a convenient set of functions for some common applications. For example, NoUpdatePermitted, described in the example above, is used to stop update of the *localState* of an active value. FirstFetch is a standard *getFn* that expects the *localState* of its active value to be

an Interlisp expression to be evaluated: on the first fetch, the instance variable is set to the result of evaluating the expression. This is illustrated in figure 5, which shows a class `TestDatum` that describes an instance variable `sampleX`, to be computed on the first time that it is fetched, and then cached for future references. At the time of activation of `FirstFetch`, `self` and `varName` are bound to the instance and instance variable name in which the active value was found.

```
(DEFCLASS TestDatum
     (MetaClass Class)
     (...
     (InstanceVariables (sampleX #((RAND 0. 100.) FirstFetch)))...)
```

Figure 5. Using an active value to compute and cache a value for a variable on the first fetch.

In some applications it is important to be able to access values indirectly from other instances. For example, Steele [Steele80] has recommended this as approach for implementing equality constraints. figure 6 shows a way of achieving this by using using the standard access functions `GetIndirect` and `PutIndirect`.

```
(DEFINST JoeAsFatherPerspective ...
     (InstanceVariables
       (age #((#$JoeAsManPerspective age) GetIndirect PutIndirect))
       ...
```

Figure 6. Active values can be used to provide indirect access to values. This is useful when it is desired for a variable in one instance to reflect the value of a variable stored elsewhere. In this example, the instance `#$JoeAsFatherPerspective` has an `age` variable which always has the same value as the `age` variable of the instance `JoeAsManPerspective`.

For some uses, the user may want to compute a default value if given, but replace the active value by the value given if the user sets the value of a variable. For this the user can employ the system provided *putFn* of `ReplaceMe`, as in:

```
#(NIL ComputeGoodValue ReplaceMe)
```

If this value is made the default in a class, then when a program tries to set this value, the instance will contain the value set. However, if the user tried to fetch the value form this variable before setting it, the *getFn* `ComputeGoodValue` would be invoked.

5.5 User-Defined Access Functions

The *getFn* and *putFn* of an active value are functions that are called with standard arguments:

(*self varName oldOrNewValue propName activeVal type*)

These arguments are interpreted as follows:

self — The object containing this active value.

varName — The name of the variable where this active value was stored. This is `NIL` if it is not stored in a variable.

oldOrNewValue — For a *getFn*, this is the *localState* of the active value. For a *putFn*, this is the new value to be stored in the active value.

propName — The name of a property. This is NIL if the active value is not associated with the value of a property (i.e., if it is associated with the value of the variable itself).

activeVal — The active value in which this *getFn* or *putFn* was found.

type — This specifies where the active value is stored: NIL means a instance variable, CV means a class variable, CLASS means a class property, or METHOD means a method property.

The value returned by the *getFn* is returned as the value of the get operation.

The *putFn* is expected to make any necessary changes to the *localState*. This can be done using function PutLocalState described below. In changing the *localState*, embedded active values may be triggered.

Given an active value, the following functions can be used to retrieve or store its *localState*:

(GetLocalState *activeValue self varName propName type*) [Function]
(PutLocalState *activeValue newValue self varName propName type*) [Function]
GetLocalState returns the *localState* of the active value *activeValue*. PutLocalState stores *newValue* as the *localState* of the active value *activeValue*, and returns *newValue*.

Note that it is necessary to pass these functions the values for *self*, *varName*, *propName*, and *type*, in case any imbedded active values are triggered.

If the *localState* of the active value is itself an active value, then it will be triggered to obtain the *localState* argument for the *getFn*. For a *putFn*, an embedded active value will be triggered when the *putFn* calls PutLocalState. The following functions can be used to access the *localState* of an active value without triggering any embedded active values:

(GetLocalStateOnly *activeValue*) [Function]
(PutLocalStateOnly *activeValue newValue*) [Function]
GetLocalStateOnly returns the value of the *localState* of the active value *activeValue*. PutLocalStateOnly stores *newValue* as the *localState* of the active value *activeValue*, and returns *newValue*. Both functions access the *localState* without triggering embedded active values.

In some cases, it is important to be able to replace the entire active value expression by some quantity, independent of the depth of nesting of active values, without destroying the outer levels of nesting:

(ReplaceActiveValue *activeVal newValue self varName propName type*) [Function]
ReplaceActiveValue overwrites *activeVal* whereever it is (either directly as the value or property of an instance variable, or as the local state of an embedded active value) with *newValue*

ReplaceActiveValue searches the value (property) determined by its arguments until it finds *activeVal* in the nesting. If *activeVal* is not found, an error is invoked.

Example: Suppose that we have a class RandomDatum which describes an instance variable sampleX, which we want to be computed as a random number on the first time that it is fetched, and then returned

as a constant on all future fetches. We could do this by defining the class as follows:

```
(DEFCLASS RandomDatum
     (MetaClass Class)
     (...
     (InstanceVariables (sampleX #(NIL SmashRandom ReplaceMe)))
     ...)
```

where the function SmashRandom is defined as follows:

```
(LAMBDA (self varName value propName activeValue)
     (ReplaceActiveValue activeValue (RAND 0. 100.) self varName]
```

On the first fetch of the value of sampleX in any instance of RandomDatum, the function SmashRandom over-writes the active value with a random number. This is a special case of the active value function FirstFetch described earlier.

The function MakeActiveValue is used to make the value of some variable or property be an active value:

(MakeActiveValue *self varOrSelector newGetFn newPutFn newLocalSt propName type*) [Function]

self is the object, *varName* is typically the name of a variable when the active value is being placed in an instance variable. If the active value is being placed in a method, then *varName* should be bound to the selector name. Active values can also be used for class variables, or properties of instance or class variables, or methods. The interpretation of where to create the active value is determined by the argument *type*, which must be one of IV (or NIL), CV, CLASS, or METHOD.

If *newLocalSt* = EMBED, then a new active value is always created, containing as its *localState* whatever was found by GetItOnly (page 23). For other values of *newLocalSt*, an active value is created only if the current value is not an active value; otherwise the old one is simply updated with *newLocalSt*, *newGetFn*, and *newPutFn*.

If an old active value is being updated, then if *newGetFn* or *newPutFn* is NIL, the old *getFn* or *putFn* is not overwritten. If *newGetFn* or *newPutFn* is T, the old *getFn* or *putFn* is reset to NIL.

The easiest way to define a function for use in active values is to use the function DefAVP:

(DefAVP *fnName putFlg*) [Function]

DefAVP creates a template for defining an active value function and leaves the user in the Interlisp editor. *fnName* will be the name of the function and *putFlg* is T if this is to be a *putFn* and NIL if it is to be a *getFn*.

For *getFn*s, the template is

```
[LAMBDA (self varName localSt propName activeVal type)
     (* This is a getFn for ...)
          localSt]
```

This template incorporates the standard arguments that a *getFn* receives, and the convention that they

often return the value that is in their local state.

For *putFns*, the template is

```
[LAMBDA (self varName newValue propName activeVal type)
     (* This is a putFn for ...)
          (PutLocalState activeVal newValue self varName propName type)]
```

This template incorporates the standard arguments that a *putFn* receives, and the convention that they often put their resulting *newValue* in the *localState*.

6 COMBINING INHERITED METHODS

In practice, most methods used to manipulate LOOPS objects are inherited. In the simplest examples of multiple inheritance, classes represent independent features and there is no conflict between inherited methods. However, when features inherited from classes interact, it is essential to be able to describe how to combine them. Howard Cannon recognized this "mixing issue" as central in the design of Flavors:

> "To restate the fundamental problem: there are several separate (orthogonal) *attributes* that an object wants to have; various *facets* of behavior (features) that want to be independently specified for an object. For example, a window has a certain behavior as a rectangular area on a bit-mapped display. It also has its behavior as a labeled thing, and as a bordered thing. Each of these three behaviors is different, wants to be specified independently for each object, and is *essentially* orthogonal to the others. It is this "essentially" that causes the trouble."
>
> "It is very easy to combine completely non-interacting behaviors. Each would have its own set of messages, its own instance variables, and would never need to know about other objects with which it would be combined. Either the multiple object or simple multiple superclass scheme could handle this perfectly. The problem arises when it is necessary to have *modular* interactions between the orthogonal issues. Though the label does not interact *strongly* with either the window or the border, it does have some minor interactions. For example it wants to get redrawn when the window gets refreshed. Handling these sorts of interactions is the Flavor system's main goal."
>
> ... from [Cannon82]

This section considers cases where the inherited features interact, and describes some LOOPS facilities for combining interacting methods. First, we describe a way of combining an inherited method with local method code. Next, we describe other ways of combining methods inherited from multiple super classes. Finally, we describe some special functions one can use to "escape" from the normal method inheritence conventions.

6.1 Augmenting an Inherited Method

The inheritance examples shown previously considered only cases where methods are inherited in toto. In these examples, subclasses inherit a method or value unchanged, or they override it completely. No mechanism was described that would enable a subclass to track changes in a method after it had been specialized in some way.

For combining an inherited method with local code, LOOPS provides the special method invocation ←Super.

(←Super *object selector* $arg_1 \cdots arg_N$) [NLambda NoSpread Function]

object is the object to which the method is applied (typically self), *selector* is the selector for the method and $arg_1 \cdots arg_N$ are the arguments for the method. As with ←, *selector* is not evaluated; the remaining arguments are evaluated.

←Super provides a form of relative addressing; it invokes the next more general method of the same name even when the specialized method invoking ←Super is inherited over a distance. An example of the use of ←Super is given in figure 7.

Note: SENDSUPER can be used instead of ←Super.

```
(BorderedWindow.Refresh
   [LAMBDA (self)          (* mjs: "11-JAN-82 19:28")

      (* * Method for refreshing a window that has a border)

                        (* First use the refresh method
                           inherited from Window.)
      (←Super self Refresh)
                        (* Then Re-display the border.)
      (← (@ :border) Display)
      self])
```

Figure 7. This Interlisp procedure implements the Refresh message for the class BorderedWindow. It uses ←Super to invoke the more general method in the class Window. The object for the "border" of the bordered window is in the instance variable border. The specialized method returns the bordered window as its value. In more complicated examples, calls to ←Super and ← can be combined using Interlisp iterative and conditional statements.

6.2 Combining Multiple Inherited Methods

Using ←Super, a method can invoke the *single* next general method. However, when a class has multiple super classes, sometimes it is necessary to invoke the general methods from *each* of the super classes. In this situation, one can call ←SuperFringe:

(←SuperFringe *object selector* $arg_1 \cdots arg_N$) [NLambda NoSpread Function]
This is similar to ←Super, except that ←SuperFringe invokes the next more general method of the same name for *each* of the super classes on the supers list of the class of the currently-executing method.

6.3 General Method Invocation

The functions ←Super and ←SuperFringe have proved to be sufficient for implementing most methods. However, sometimes it is necessary to manipulate multiple inherited methods, and invoke them in some other order. The following functions provide more general ways of invoking particular methods. It is important to note that while these functions are more powerful than ←Super or ←SuperFringe, they are also more "dangerous", in that they do not conform to the conventions of method inheritence. These functions should only be used as a last resort when a method cannot be implemented in any other way.

(DoMethod *object selectorExpr class* $arg_1 \cdots arg_N$) [NLambda NoSpread Function]
DoMethod allows computation of the name of the selector and the class from which that method should be found: it applies that method to *object*.

All the arguments to DoMethod are evaluated; *selectorExpr* should evaluate to a selector name in the class computed from *class*. If *class* is NIL, then the class of *object* is used. If no method for the computed selector is found in the computed class, an error is generated. The remaining arguments, $arg_1 \cdots arg_N$ are the arguments

for the method.

In the case where the arguments to the method have already been evaluated, then one can use `ApplyMethod` instead of `DoMethod`:

`(ApplyMethod` *object selector argList class*`)` [Function]
argList is a list of all the arguments to the method (except *object*) already evaluated. The function applied is the one found by searching from *class*. If *class* is `NIL`, the class of *object* is used.

`(DoFringeMethods` *object selectorExpr* $arg_1 \cdots arg_N$`)` [NLambda NoSpread Function]
Like `DoMethod`, all of the arguments are evaluated. `DoFringeMethods` calls the method for *selectorExpr* in the class of *object*, if that method is defined in that class. If the method is not defined in the class of *object*, the method of the same name for *each* of the super classes on the supers list of the class of *object* is envoked.

7 INSTANCE CREATION

The standard process of creating an instance of a class is to send a `New` message to the class. In the simplest case, this causes the information in the *instance variable descriptions* of the class to be used to establish default values for variables in the newly created instance. When that process is finished, the instance can be altered in various ways by sending it messages.

LOOPS provides a variety of facilities for controlling this by using active values, standard access functions, and metaclasses. This section summarizes some of the common cases. See page 38 for an illustratation of the use of these facilities to support the important example of composite objects.

7.1 Specifying Values at Instance Creation

The `NewWithValues` message simplifies the case where it is desired to specify values and properties in an instance when it is created. The form of this message is:

(← *class* `NewWithValues` *valDescriptionList*) [Message]

valDescriptionlist must evaluate to a list of value descriptions, each of which is a list of a variable name, variable value, and properties; e.g.

$$((varName_1\ \ value_1\ \ prop_1\ \ propVal_1\ \cdots)\ \ (varName_2\ \ value_2\ \cdots)\ \cdots)$$

The method for `NewWithValues` first creates the object with *no* other initialization (e.g. without computing values specified in the class, as described in sections below). It then directly installs the values and property lists specified in *valDescriptionList* and returns the created object. Variables which have no description in *valDescriptionList* will be given no value in the instance, and thus will inherit the default value from the class.

7.2 Sending a Message at Instance Creation

A simplification in form is available when one wants to send a message to an instance immediately after its creation. For example, consider:

```
(← (← ($ Transistor) New) Display windowCenter)
```

which creates an instance of the `Transistor` class, and then displays it at a point `windowCenter`. A more compact notation for doing this is provided:

```
(←New ($ Transistor) Display windowCenter)
```

where `←New` ("send New") means to create a new instance and send it a message. The value returned by `←New` is the new instance. Any value returned by the method is discarded.

In order to name an object, one can send the message `SetName` to that object. As a simplification, if one provides an argument to the `New` message, the default interpretation of that argument is to use it as a name, sending the newly created object the `SetName` message.

7.3 Computing a Value at First Fetch

As described earlier, one can use an active value to activate arbitrary procedures when values are fetched. The built-in function `FirstFetch` can be used as a *getFn* in an active value as the default value in the class. If no value has been assigned to the variable or property before the value is fetched for the first time, the `FirstFetch` active value is invoked.

The local state of this active value can be a list which is a form to be evaluated. During the evaluation, the variables `self`, `varName`, and `propName` are are appropriately bound. The local state of the `FirstFetch` active value can also be an atom: if so, it is treated as the name of a function to be applied to the object, `varName` and `propName`. The value of the form or function application is made the value in the instance as well as being returned as the value of the fetch.

For example, the random number example could have been done as follows:

```
(DEFCLASS TestDatum
     (MetaClass Class)
     (...
     (InstanceVariables (sampleX #((RAND 0. 100.) FirstFetch)))
     ...)
```

In this example `FirstFetch` evaluates the form `(RAND 0. 100.)` and replaces the value of the `sampleX` variable of the instance by the random number. In many cases the form may be a ← expression.

7.4 Computing a Value at Instance Creation

In the previous example, `FirstFetch` initializes the value of an instance variables at first access. Sometimes it is important to initialize an instance variable when the instance is created. For such cases LOOPS provides a distinguished *getFn*, `AtCreation`. If a default value of an instance variable or property contains an active value with `AtCreation` as its *getFn*, then at creation time, the *localState* of this active value will be used to determine a value to be inserted in the new instance.

As with `FirstFetch`, if the *localState* is an atom, then it will be treated as the name of a function to be applied to the object, variable name, and property name. If it is a list, then that list will be evaluated in a context in which `self`, `varName`, and `propName` are appropriately bound. Functions run at initialization time are run in the order in which they appear in the class. Default values of variables are available to these functions.

If an object is created by `NewWithValues` without a value being supplied for a variable which contains an `AtCreation` default value, then at the first fetch of that variable, the function or form will be evaluated.

Example:

Suppose we want to have an instance variable called `creationDate` which tells the date that an instance was created. This can be implemented in LOOPS as follows:

```
(DEFCLASS DatedObject
     (MetaClass Class)
     (...
```

```
(InstanceVariables (creationDate #((DATE) AtCreation)))
...)
```

The function DATE in Interlisp computes a string which is the current date and time. The value of this string at instance creation time is made the initial value of `creationDate`.

Another use of an `AtCreation` active value might be to make an index entry to a newly created object.

7.5 Special Actions at Instance Creation

For some special cases, the user may want to have more control over the creation of instances. For example, LOOPS itself uses different LISP data types to represent classes and instances. The **New** message for classes is fielded by their metaclass, usually the object `MetaClass`. This section shows how to create a new metaclass.

Any metaclass should have `Class` as one of its super classes and `MetaClass` as its metaclass. The easiest way to create a new metaclass is to send a **New** message to `MetaClass` as follows:

(← ($ MetaClass) New *metaClassName supers*)

This creates a new metaclass with the name *metaClassName* and with the super classes named in the list *supers*. The default supers for metaclasses is the list containing `Class`. The metaclass for the the new class is `MetaClass`.

One then installs the specialized method for **New** in the new metaclass. This method provides the mechanism for creations of instances of the class which have this as a metaclass. Sending this metaclass the message **New** will cause the creation of a class with the appropriate property.

As a simple example we will define a new metaclass `ListMetaClass` which will augment the instance creation process by keeping a list of all instances which have been created. This list will be kept on the class property `allInstances`. To create this class we go through the scenario in figure 8.

```
← (← ($ MetaClass) New 'ListMetaClass '(Class))
#$ListMetaClass          - We have now defined a new metaclass

                         - This defines the New method for that metaclass
← (DM 'ListMetaClass 'New '(self name)
    '((* Create an instance and add it to list in class)
      (PROG ((newObj (←Super self New name)))
                (* newObj created by super method from class)
         (PutClass
            self
            (CONS newObj
                  (LISTP (GetClassHere self 'AllInstances)))
            'AllInstances)
                  (* LISTP returns previous list or NIL if none)
         (RETURN newObj]
ListMetaClass.New

← (← ($ ListMetaClass) New 'Book)
```

```
#$Book                          - This creates a new class ($ Book)
                                whose metaclass is ($ ListMetaClass)
← (← ($ Book) New 'B1)
#$B1                            - Creating #$B1 using ListMetaClass.New
← (← ($ Book) New 'B2)
#$B2
← (GetClass ($ Book) 'AllInstances)
(#$B1 #$B2)                     - The list of instances created so far.
```

Figure 8. In this scenario, a new metaclass `ListMetaClass` is defined by the `New` method of `($ MetaClass)`. It has metaclass `($ MetaClass)`. We then define the specialized `New` method for `ListMetaClass`. This includes a call to its super `(Class)` to actually create the object; it puts the newly created object on its list of objects. We then create `($ Book)` which has `ListMetaClass` as its metaclass. When two instances of book are created, each is placed on the list `AllInstances` which is a class property.

8 COMPOSITE OBJECTS

LOOPS extends the notion of objects to make it recursive under composition, so that one can instantiate a group of related objects as an entity. This is especially useful when relative relationships between members of the group must be isomorphic (but not equal) for distinct instances of the group. The implementation of composite objects combines many of the programming features described above. In particular, it is an application of the notion of metaclass.

8.1 Basic Concepts for Composite Objects

Parameters and Constants: LOOPS supports the use of structural templates to describe composite objects having a fixed set of parts. Composite objects are normal LOOPS objects, created by an instantiation process and describable in the class inheritance network. This contrasts with the idea of using for templates data structures that are merely *copied* to yield composite objects. A primary benefit of making composite objects be classes is the ability to create slightly modified versions of a template by making a new subclass which inherits most of the structure of its super.

Creating a Template: To describe a composite object, one creates a class whose metaclass is `Template`. One can also use a metaclass one of whose supers is `Template`. Any class whose metaclass is `Template` or one of its subclasses is called a template. In a template, the default values for instance variables can point to other templates; these will be treated as *parameters* and will be recursively instantiated when the parent template is instantiated. All non-template classes and any other default values are treated as *constants* that are simply inherited by instances.

Instantiation: Instances of a template are created by sending it a `New` message. The instantiation process is recursive through all of the parameters of a template. Every parameter is instantiated when it is first encountered. Multiple references to the same parameter are always replaced by references to the same instantiated instance. The instantiated composite object that is created is isomorphic to the original template structure with constants inherited and with distinct instances substituted for distinct templates (parameters). Parameters in lists or active values are found and the containing structure is copied with appropriate substitutions. If a composite object needs multiple distinct instances of the same type (e.g., two inverters), then multiple templates are needed in the description.

Example: figure 9 shows an example from digital design - a composite object for `BitAmplifier` that is composed of two series-connected inverters. The input of the first inverter is the input of the amplifier, the output of the first inverter is connected to the input of the second inverter, and the output of the second inverter is the output of the amplifier. Different instantiations of `BitAmplifier` contain distinct inverters connected in the same relative way. This example also shows a possible use of active values in templates. The containing composite object is set up so that its *output* instance variable uses an active value to track the value of the output variable of the second inverter.

```
[DEFCLASS BitAmplifier
   (MetaClass Template doc
      (* * Composite object template for an amplifer
           made of two series connected inverters.))
   (Supers Amplifier)
   (ClassVariables)
   (InstanceVariables
```

```
      (inputTerminal ($ Inverter1))
      (output #( (($ Inverter2) output) GetIndirect PutIndirect)
         doc (* Data is stored and fetched from the variable
                output in the instance of Inverter2))
   (Methods)]

[DEFCLASS Inverter1
   (MetaClass Template partOf ($ BitAmplifier)
         doc (* Instance variable Input is inherited from Inverter))

   (Supers Inverter)
   (ClassVariables)
   (InstanceVariables
      (output ($ Inverter2)
         doc (* Output connected to second inverter)))
   (Methods)]

(DEFCLASS Inverter2
   (MetaClass Template partOf ($ BitAmplifier) )
   (Supers Inverter)
   (ClassVariables)
   (InstanceVariables
      (input ($ Inverter1)
         doc (* Input connected to first inverter)))
   (Methods)]
```

Figure 9. Composite object templates for a `BitAmplifier`. When instances are made, they will have distinct instances of the two inverters, with their input and output interconnected. The instantiation process must be able to reach (possibly indirectly) all of the parts starting from the class to which the `New` message is sent. In this case, `Inverter1` and `Inverter2` are both mentioned in `BitAmplifier`. The example also illustrates the use of active values to provide indirect variable access in LOOPS. In this example, the active value enables the output variable of an instance of `BitAmplifier` to track the corresponding output variable of an instance of `Inverter2` in the same composite object.

8.2 Specializing Composite Objects

Because the templates are classes, all of the power of the inheritance network is automatically available for describing and specializing composite objects. To make this convenient, one can send the message `Specialize` to any template form. For example:

```
(← ($ BitAmplifier) Specialize)
```

This creates a new set of templates such that each template in the new set is a specialization of a template in the old set. One can then selectively edit the templates describing the new composite object. In particular, one may want to change the names of the generated classes by sending them the message `SetName`. Unchanged portions of the template structure will continue to inherit values from the parent composite object. A user can specialize a template by overriding instance variables. To add parameters, one creates references to new templates. Conversely, one can make a parameter into a constant by overriding an inherited variable value with a non-template in a subclass.

8.3 Conditional and Iterative Templates

Because the templates are fixed, they are not a sufficient mechanism for describing the instantiation of composite objects having conditional or repetitive parts. Consistent with our stand on control mechanisms, we have not added *conditional* or *iterative structural descriptions* to LOOPS, but use available Interlisp control structures in methods. For these cases, a user defines a new metaclass for the composite object. (Recall that metaclasses are classes whose instances are classes.) The metaclasses for templates should be subclasses of the distinguished metaclass `Template`. The specialized metaclass should have a `New` method that performs the conditional and iterative steps in the instantiation. This approach works well in conjunction with the LOOPS mechanisms for specializing classes and methods. For example, the specialized `New` method can use `←Super` to access the standard code for the template-directed portion of the instantiation process. figure 10 shows an example of a LOOPS template for a ring oscillator. This composite object is made of a loop of serially connected inverters.

```
(MetaRingOscillator.New
   [LAMBDA (self assocList numStages)    (* mjs: "11-JAN-82 19:28")
              (* * Procedure for creating a ring oscillator.)

   (PROG (ringOscillator firstInverter lastInverter inv1)
                              (* Create the inverter chain.)
      (SETQ inv1 (SETQ firstInverter (← ($ Inverter) New)))
      [for i to (SUB1 numStages)
        do (SETQ lastInverter (← ($ Inverter) New))
           (← inv1 Connect lastInverter)
           (SETQ inv1 lastInverter]
                              (* Close the loop)
      (← lastInverter Connect firstInverter)
                              (* Make the ringOscillator object.)
      (SETQ ringOscillator (←Super self New assocList))
              (* * the assocList here is the pairing
                   of Template classes found in the
                   instantiation of a template so far)
      (@← (ringOscillator input) firstInverter)
      (@← (ringOscillator output) lastInverter)
      (RETURN ringOscillator) ])
```

Figure 10. Example of an iteratively specified composite object, a ring oscillator. The ring oscillator is composed of a series of inverters serially-connected to form a loop. To specify the iteration and interconnection of the inverters, a `New` method is defined for the metaclass `MetaRingOscillator`. The Interlisp function for this method (`MetaRingOscillator.New`) uses `←Super` to perform the template-driven part of the instantiation, that is, instantiating the ring oscillator object itself. In this case, the template-driven portion of the instantiation is trivial, but the example shows how it can be combined generally with the procedural description. `MetaRingOscillator.New` uses iterative statements to make an instance of `Inverter` for each stage of the oscillator. After connecting the components together, it returns the ring oscillator object.

9 LOOPS KNOWLEDGE BASES

Loops was created to support a design environment in which there are community knowledge bases that people share, and to which they can add incremental updates. This section describes our goals for this facility, the concepts that we have employed, and scenarios for using knowledge bases in Loops.

We have chosen the term knowledge base instead of data base to emphasize two things: the kind of information being stored and constraints on the amount of information. Loops will be used mainly for expert system applications where relatively modest amounts of information are used for guiding reasoning. This information (i.e., knowledge) consists of inference rules and heuristics for guiding problem solving. This is in contrast to potentially enormous files of facts, for example, social security records for California. Reflecting this difference of scale, we have optimized the implementation to support fast access and updating to a smaller amount of information which is expected to fit in main memory for any one session. For example, we maintain an index to the object information in computer memory.

9.1 Review of Knowledge Base Concepts

Knowledge Bases: Knowledge bases in LOOPS are files that are built up as a sequence of layers, where each layer contains changes to the information in previous layers. A user can choose to get the most recent version of a knowledge base (that is, all of the layers) or any subset of layers. The second option offers the flexibility of being able to share a community knowledge base without necessarily incorporating the most recent changes. It also provides the capability of referring to or restoring any earlier version. figure 11 illustrates this with an example.

```
------------------------- Layer 1 -------------------------
Obj1 (x 4) ...
Obj2 (y 5) (w 3) ...
------------------------- Layer 2 -------------------------
Obj2 (y 7) (w 2) ...
Obj3 (z 6) ...
------------------------- Layer 3 -------------------------
Obj1 (x 8) ...
Obj4 (z 9) ...
```

Figure 11. Knowledge bases in LOOPS are files that are built-up incrementally as a sequence of layers. Each layer contains updated descriptions of objects. When a knowledge base is opened, the information in the later layers overrides the information in the earlier layers. LOOPS makes it possible to select which layers will be used when a knowledge base is opened. In this example, if the knowledge base is opened and only the first 2 layers are used, then `Obj1` will have an `x` variable with value 4. If all three layers were connected, then the value would be 8.

Community Knowledge Bases: LOOPS partitions the process of updating a community knowledge base into two steps. Any user of a community knowledge base can make tentative changes to a community knowledge base in his own (isolated) environment. These changes can be saved in a layer of his personal knowledge base, and are marked as associated with the community knowledge base. In a separate step, a data base manager can later copy such layers into a community knowledge base. This separation of tasks is intended to encourage experimentation with proposed changes. It separates the responsibility for

exploring possibilities from the responsibility of maintaining consistent and standardized knowledge bases for shared use by a community. The same mechanisms can be used by two individuals using personal knowledge bases to work on the same design. They can conveniently exchange and compare layers that update portions of a design.

Unique Identifiers: The ability to determine when different layers are referring to the same entity is critical to the ability to share data bases. To support this feature the LOOPS data base assigns unique identifiers (based on the computer's identification numbers, the date, and an unbounded count) to objects before they are written to a knowledge base. This facility provides a grounding for more sophisticated notions of equality that might be desired in knowledge representation languages built on LOOPS.

Environments: A user of LOOPS works in a personalized *environment.* An environment provides a lookup table that associates unique identifiers with objects in the connected knowledge bases. In an environment, user indicate dominance relationships between selected knowledge bases. When an object is referenced through its unique identifier, the dominance relationships determine the order in which knowledge bases are examined to resolve the reference. By making personal knowledge bases dominate over community knowledge bases, a user can override portions of community knowledge bases with his own knowledge bases.

Multiple Alternatives: An important use of environments is for providing speedy access to alternative versions (e.g., multiple alternatives in a design). A user can have any number of environments available at the same time. Each environment is fully isolated from the others. Operations that move information between environments are always done explicitly through knowledge bases.

9.2 Environmental Objects and Boot Layers

Knowledge bases, environments, and layers are represented in Loops by special objects called *environmental objects.* All knowledge base and environment operations are performed by sending messages to these objects. Environmental objects are accessible from any environment in Loops.

In this section, we will need to distinguish between environmental objects and the things that they represent. figure 12 summarizes some of the terminology that we will use.

Loops Object	*Represents*	*Description*
Layer	file layer	Portion of a file which contains descriptions of objects.
KB	knowledge base	A file and sequence of file layers. A knowledge is known by the name field of its file name.
KBState	State of a knowledge base	A sequence of file layers. Used to access a fixed explicit set of file layers (e.g., a version of a knowledge base that is older than the most recent version).
Environment	environment	An environment associates names and unique identifiers with objects in working memory.

Figure 12. Summary of terminology for environmental Loops objects and the entities that they represent.

Environments: An Environment provides a name space in working memory. Each Environment associates names and unique identifiers with objects. In general, Environments are designed to be independent. For convenience, Environments are usually named. An Environment is always associated with a particular knowledge base. The specifications for creating an Environment come from some knowledge base, and changes to the Environment are stored on that knowledge base.

Layers: A file layer is a portion of a file which contains descriptions of objects. An object description consists of a unique identifier and an expression that can be read by Interlisp to create the Loops object. A different unique identifier is associated with each expression. In addition, a file layer contains a mapping from names (Interlisp atoms) to unique identifiers. A file layer is represented in Loops by a Layer object. A Layer indicates the file on which it is written, the starting address of the file layer, and the name of the knowledge base with which it is conceptually associated. A Layer also contains various bookkeeping information such as the name of its creator and the date of its creation.

KBs and KBStates: A knowledge base is a set of file layers. Typically, most of the layers of a knowledge base are located on a single file. A knowledge base is known by its file name. By convention, such files have the extension "KB". A KB is a Loops object that represents a knowledge base. A KB has a name equal to the name field of the file name of the knowledge base that it represents. For example, the KB with name `Test` would be associated with a version of the file `Test.KB`.

A KBState is a generalization of a KB. It refers to an explicit set of file layers. KBs and KBStates indicate their Layers using a list on an instance variable named `contents`. An element of this list must be either a Layer or a KBState. When a KBState appears in the list, it is as if the Layers listed in the KBState's contents variable appeared explicitly in the list. This provides a mechanism for indirect fetching of layers from other knowledge bases.

To indicate all of the layers of the most recent version of a knowledge base, the contents of the KBState can be the special value "CURRENT". When such a KBState appears in the list, it is as if the Layers of the most recent version of the knowledge base were inserted in the list. These Layers are retrieved by retrieving the KB from the referenced knowledge base.

Boot Layers: Environmental objects are distinguished from other objects when they are accessed and when they are written out to a knowledge base. They are accessed differently in that they are kept in a global name table accessible in all environments. This means that an Environment can be described in terms of the environmental objects before the Environment is made current.

Environmental objects are also special in that the file layer that describes them is a special file layer at the end of a knowledge base called the boot layer. In order to access the contents of a knowledge base, it is necessary to read the boot layer first because it contains the environmental objects that describe the knowledge base. A boot layer for a knowledge base contains a single KB describing itself, a Layer describing each of its file layers, and the KBStates mentioned (directly or indirectly) in the KB.

The Global Name Table: Loops keeps environmental objects in a global name table that is accessible from any environment. This name table also includes the basic classes that are part of the Loops kernel. If Loops is used without exercising the Environments feature, then all created objects are also placed in the global table.

When another environment is opened, objects not in core are first looked for by UID or name in the open environment. If no object is found there, then the UID or name is looked up in the Global Environment. Thus, object descriptions in a new environments override those in Global Envrionment, but old objects which have no counterparts are still available.

9.3 Starting With No Preexisting Knowledge Bases

The knowledge base facility in Loops has been designed to cover a number of situations. Because of this generality, it is not always easy for a newcomer to discover the simplest way of using the features. The following sections describe all the features of the Knowledge Base system; however each feature is introduced within a particular scenario that shows how to do some of the most common operations for which Loops was designed.

In the first scenario, a user wants to start from scratch using no preexisting knowledge bases. The results of this Loops session are saved in a personal knowledge base.

When a user invokes Loops, the Loops name space will contain some objects from the Loops kernel. Before creating any new objects, the user should type an expression of the form:

(← $KB New '*KBName* '*environmentName newVersionFlg*)

where *KBName* is an atom (e.g., use FOO to create a knowledge base named FOO.KB) and *environmentName* will be the name of the Environment. This will create both a new KB corresponding to the *KBName* and a new Environment with the name *environmentName*.

Loops checks that a knowledge base with *KBName* does not already exist. If it does exist and *newVersionFlg* is NIL, Loops will report an error. If *newVersionFlg* is T, then Loops will create a new version of the file. Because of the way the file system works, the name of a KB must be all in upper case. If the user attempts to use a *KBName* which contains lowercase letters, Loops will correct the name to all upper case and print a warning message.

Warning: Objects created before creating and opening an Environment are placed in the global name table. Hence, any objects so created will be shared by all Environments. However, Loops will not save such objects in a knowledge base later in the session unless they are explicitly moved to some environment. Alternatively, such objects can be saved using the Interlisp file package.

The next step is to open the Environment:

```
(← $environmentName Open)
```

This makes the new Environment be the current environment. New objects that are created will be associated with the KB.

Having created an Environment, the user can then proceed to create whatever new objects he desires in the session. To dump the current state of the environment and continue afterwards, the user can type:

```
(← $environmentName Cleanup)
```

This does not close any files, and leaves the environment as it was, except that all changed objects have been dumped to the knowledge base, and then marked as unchanged. `Cleanup` can be done any number of times in a session.

At the end of a session the user should do a `Close`:

```
(← $environmentName Close)
```

This writes out all of the objects to a file layer, updates the environmental objects accordingly, and writes them out to a boot layer, deletes these objects from memory, and closes all files associated with the environement. The user can then exit from Interlisp. After a `Close` is done, the user must go through the following scenario to start up again.

9.4 Continuing from a Previous Session

The case where a user wants to create a new knowledge base is less common than the case where he wants to modify or add objects to a knowledge base that he has previously created. In this scenario a user wants to resume from where he was at the end of his previous session.

The first step is to obtain the user's knowledge base, and link it to an environment. This is done by a message to the class `KB` as follows:

```
(← $KB Old 'KBName 'environmentName)
```

This reads the boot layer of the knowledge base named *KBName* and creates an Environment named *environmentName* that is then connected to the KB. At this point the user must open the environment to make the contents of the KB available in this environment:

```
(← $environmentName Open)
```

This causes Loops to read in each Layer contained (possibly implicitly) in the contents of the associated KB (named *KBName*). It also makes the new Environment be the current environment. Having opened an Environment, the user can then proceed to define whatever new objects he desires in the session. New objects that are created will be associated with the KB. When he is done, he should type as in the previous scenario:

```
(← $environmentName Cleanup)
```

or

```
(← $environmentName Close)
```

9.5 Starting from a Community Knowledge Base

Users will not usually start from scratch. Rather, they will often begin by using previously created community knowledge bases. This scenario starts with obtaining a single community knowledge base. The user does not own the community knowledge base, so the results of the session will have to be saved in a personal knowledge base. The personal knowledge base will contain any new objects that created as well as any objects from the community knowledge base that have changed.

As in the first or second scenario, the first step is to create a personal knowledge base.

```
(← $KB New 'KBName 'environmentName newVersionFlg)
```

or if the user has a personal knowledge base already, by doing a:

```
(← $KB Old 'KBName 'environmentName)
```

This obtains both the KB and an Environment. The next step is to add the community knowledge base to the KB as follows:

```
(← $KBName AddToContents 'communityKBName)
```

where *communityKBName* is an atom that is the name of the community knowledge base.

This step should be repeated for each knowledge base to be added to the KB named *KBName*. The message creates a KBState describing the "current" state of the community knowledge base and adds that KBState to the contents of the KB for the personal knowledge base. The effect of this action is that Loops will remember to associate the community knowledge base with the user's knowledge base in the future. (This step need not be repeated in any future session which uses the knowledge base *KBName*.)

At this point, the user can open the Environment as before:

```
(← $environmentName Open)
```

This causes Loops to read in each Layer contained (possibly implicitly) in the contents of the KB named *KBName*. The `Open` message also makes the Environment named *environmentName* be the current environment.

Since the KB associated with the environment contains a KBState for *communityKBName*, those Layers will also be read. They are found by reading the boot layer of the community knowledge base. The message `AddToContents` on *KBName* will work properly even after the environment is `Open`, in the sense that when it is done on a KB connected to an `Open` environment, it causes all the layers of the newly added KB to be read in.

All creation and modification operations will take place in this Current Environment. The user can create new objects and modify objects in the community knowledge base. When done, the results of the session can be saved using `Cleanup` (or `Close`). This will cause two file layers to be written out to the personal knowledge base (and none to the community knowledge base). First a file layer is written out to *KBName* for changes made to the community knowledge base (if any). The Layer for this file layer is marked as associated with the community knowledge base. Second, a file layer is written out for the