Object-Oriented Programming: Themes and Variations

Mark Stefik & Daniel G. Bobrow

Intelligent Systems Laboratory, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304

Many of the ideas behind object-oriented programming have roots going back to SIMULA (Dahl & Nygaard, 1966). The first substantial interactive, display-based implementation was the SMALLTALK language (Goldberg & Robson, 1983). The object-oriented style has often been advocated for simulation programs, systems programming, graphics, and AI programming. The history of ideas has some additional threads including work on message passing as in ACTORS (Lieberman, 1981), and multiple inheritance as in FLAVORS (Weinreb & Moon, 1981). It is also related to a line of work in AI on the theory of frames (Minsky, 1975) and their implementation in knowledge representation languages such as KRL (Bobrow & Winograd, 1977), KEE (Fikes & Kehler, 1985), FRL (Goldstein & Roberts, 1977) and UNITS (Stefik, 1979).

One might expect from this long history that by now there would be agreement on the fundamental principles of object-oriented programming. As it turns out, the programming language community is still actively experimenting. Extreme languages can be found which share the description "object-oriented" but very little else. For example, there are object-oriented operating systems that use a much more general notion of message sending than in most of the languages described here.

This article is an introduction to the basic ideas of programming with objects. A map of the field is naturally drawn from where one stands. Most of the examples will be from the authors' own system, Loops (Bobrow & Stefik, 1981), and we will describe other object languages from that vantage point. We have not tried to be complete in our survey; there are probably fifty or more object-oriented programming languages now in use, mostly with very limited distribution. We have selected ones we know that are widely used for applications in artificial intelligence or have a particularly interesting variation of an issue under discussion. For pedagogical purposes we begin with a white lie. We introduce message sending and specialization as the most fundamental concepts of object-oriented programming. Then, we will return to fundamentals and see why some object languages don't have message sending, and others don't have specialization.

Thanks to Ken Kahn and Mark Miller who were especially generous with their time and ideas as we prepared this article for publication Thanks also to Sanjay Mittal and Stanley Lanning who read earlier drafts and who contributed to the design and implementation of Loops Ken Kahn, Gregor Kiczales, Larry Masinter, and Frank Zdybel helped broaden our understanding of the variations in object-oriented languages as we worked together on the design of CommonLoops. Much of the discussion in this paper was inspired by the electronic dialog of the members of the Object-oriented Subcommittee for Common Lisp.

Special thanks to John Seely Brown and Lynn Conway, who encouraged our work on Loops, and helped us to develop larger visions while we slogged through the bits Thanks also to Bill Spencer and George Pake for maintaining the kind of intellectual environment at PARC that has allowed many different projects in language design to flourish.

Abstract

Over the past few years object-oriented programming languages have become popular in the artificial intelligence community, often as add-ons to Lisp. This is an introduction to the concepts of object-oriented programming based on our experience of them in Loops, and secondarily a survey of some of the important variations and open issues that are being explored and debated among users of different dialects.

Basic Concepts of Object-Oriented Programming

The term object-oriented programming has been used to mean different things, but one thing these languages have in common is *objects*. Objects are entities that combine the properties of procedures and data since they perform computations and save local state. Uniform use of objects contrasts with the use of separate procedures and data in conventional programming.

All of the action in object-oriented programming comes from sending messages between objects. Message sending is a form of indirect procedure call. Instead of naming a procedure to perform an operation on an object, one sends the object a message. A selector in the message specifies the kind of operation. Objects respond to messages using their own procedures (called "methods") for performing operations.

Message sending supports an important principle in programming: *data abstraction*. The principle is that calling programs should not make assumptions about the implementation and internal representations of *data types* that they use. Its purpose is to make it possible to change underlying implementations without changing the calling programs. A data type is implemented by choosing a representation for values and writing a procedure for each operation. A language supports *data abstraction* when it has a mechanism for bundling together all of the procedures for a data type. In object-oriented programming the class represents the data type and the values are its instance variables; the operations are methods the class responds to.

Messages are usually designed in sets to define a uniform interface to objects that provide a facility. Such a set of related messages is called a *protocol*. For example, a protocol for manipulating icons on a display screen could include messages for creating images of icons, moving them, expanding them, shrinking them, and deleting them. When a message protocol is designed for a class, it should be made general enough to allow alternative implementations.

There is additional leverage for building systems when the protocols are *standardized*. This leverage comes from *polymorphism*. In general the term polymorphism means "having or assuming different forms," but in the context of object-oriented programming, it refers to the capability for different classes of objects to respond to exactly the same protocols. Protocols enable a program to treat uniformly objects that arise from different classes. Protocols extend the notion of *modularity* (reusable and modifiable pieces as enabled by data-abstracted subroutines) to *polymorphism* (interchangeable pieces as enabled by message sending).

After message sending, the second major idea in object-oriented programming is specialization. Specialization is a technique that uses class inheritance to elide information. Inheritance enables the easy creation of objects that are *almost like* other objects with a few incremental changes. Inheritance reduces the need to specify redundant information and simplifies updating and modification, since information can be entered and changed in one place.

We have observed in our applications of Loops that changes to the inheritance network are very common in program reorganization. Programmers often create new classes and reorganize their classes as they understand the opportunities for factoring parts of their programs. The Loops programming environment facilitates such changes with an interactive graphics *browser* for adding and deleting classes, renaming classes, splitting classes, and rerouting inheritance paths in the lattice.

Specialization and message sending *synergize* to support program extensions that preserve important invariants. Polymorphism extends downwards in the inheritance network because subclasses inherit protocols. Instances of a new subclass follow exactly the same protocols as the parent class, until local specialized methods are defined. Splitting a class, renaming a class, or adding a new class along an inheritance path *does not affect simple message sending* unless a new method is introduced. Similarly, deleting a class does not affect message sending if the deleted class does not have a local method involved in the protocol. Together, messsage sending and specialization provide a robust framework for extending and modifying programs.

Fundamentals Revisited

Object languages differ, even in the fundamentals. We next consider object languages that do not have message sending, and one that does not have specialization.

Variations on Message Sending. When object languages are embedded in Lisp, the simplest approach to providing message sending is to define a form for message sending such as:

```
(send object selector arg1 arg2 ...)
```

However, some language designers find the use of two distinct forms of procedure call to be unaesthetic and a violation of data abstraction: a programmer is forced to be aware of whether the subsystem is implemented in terms of objects, that is, whether one should invoke methods or functions.

An alternative is to unify procedure call with message sending. Various approaches to this have been proposed. The T (Rees & Meehan, 1984) programming language unifies message sending and procedure calling by using the standard Lisp syntax for invoking either methods or functions. For example, `(display obj x y)` could be used either to invoke the display method associated with obj, or to invoke the display lisp function. A name conflict resulting in ambiguity is an error.

CommonLoops (Bobrow, Kan, Kiezales, Masinter, Stefik, & Zdybel, 1985) takes this unification another step.

Lisp function call syntax is the only procedure calling mechanism, but ordinary Lisp functions can be extended by methods to be applied when the arguments satisfy certain restrictions. In Lisp, functions are applied to arguments. The code that is run is determined by the name of the function. The Lisp form (foo a b) can be viewed as:

```
(funcall (function-specified-by 'foo) a b)
```

Sending a message (send a foo b) in object-oriented programing can be viewed as equivalent to the invocation of:

```
(funcall (method-specified-by 'foo (type-of a) ) a b)
```

The code that is run is determined by both the name of the message, foo, and the type of the object, a A method is invoked only if its arguments match the specifications. In this scheme a method with no type specifications in its arguments is applied if no other method matches. These methods are equivalent to ordinary functions, when there are no other methods for that selector. From the point of view of the caller, there is no difference between calling a function and invoking a method.

CommonLoops extends the notion of method by introducing the notion of "multi-methods" to Common Lisp. It interprets the form (foo a b ...) as:

```
(funcall (method-specified-by 'foo (type-of a)
                                   (type-of b)...) a b ...)
```

The familiar methods of "classical" object-oriented programming are a special case where the type (class) of only the first argument is used. Thus there is a continuum of definition from simple functions to those whose arguments are fully specified, and the user need not be aware of whether there are multiple implementations that depend on the types of the arguments. For any set of arguments to a selector, there can be several methods whose type specifications match. The most specific applicable method is invoked.

A variation among object oriented languages is whether the method lookup procedure is "built-in." In the languages we have described here, there is a standard mechanism for interpreting messages—with the selector always used as a key to the method. In Actors, the message is itself an object that the receiver processes however it wishes. This allows other possibilities such as pattern matching on the message form. It also allows "message plumbing" where the receiver forwards the entire message to one or more other objects. Splitting streams to allow one output to go to two sources is a simple example of the use of this feature.

Variations on Specialization. Specialization as we have introduced it so far is a way to arrange classes so that they can inherit methods and protocols from other classes. This is a special case of a more general concept: the concept is that objects need to handle some messages themselves, and to pass along to other objects those messages that they don't handle.

In actor languages (Lieberman, 1981) this notion is called "delegation" and it is used for those programming situations where inheritance would be used in most other object languages. Delegation is more general than specialization, because an actor can delegate a message to an arbitrary other object rather than being confined to the paths of a hierarchy or class lattice.

If delegation was used in its full generality for most situations in actor programming, the specifications of delegation could become quite verbose and the advantages of abstraction hierarchies would be lost. Actor programs would be quite difficult to debug. In practice, there are programming cliches in these languages that emulate the usual forms of inheritance from more conventional object languages, and macros for language support. However, since there is no standardization on the type of inheritance, it makes it more difficult for a reader of the code to understand what will happen.

Classes and Instances

In most object languages objects are divided into two major categories: classes and instances. A class is a description of one or more similar objects. In comparison with procedural programming languages, classes correspond to types. For example, if "number" is a type (class), then "4" is an instance. In an object language, *Apple* would be a class, and *apple-1* and *apple-2* would be instances of that class. Classes participate in the inheritance lattice directly; instances participate indirectly through their classes. Classes and instances have a declarative structure that is defined in terms of object variables for storing state, and methods for responding to messages.

Even in these fundamentals, object languages differ. Some object languages do not distinguish between classes and instances. At least one object language does not provide a declarative structure for objects at all. Some languages do not distinguish methods from variable structure. Languages also differ in the extent to which variables can be annotated.

Themes

We begin be describing classes and instances as they are conceived in Loops. We will then consider some variations.

What's in a Class? A class in Loops is a description of one or more similar objects. For example, the class *Apple* provides a description for making instances, such as *apple-1* and *apple-2*. Although we usually reserve the term instance to refer to objects that are not classes, even classes are instances of a class (usually the one named *Class)*. Every object in Loops is an instance of exactly one class.

GLOSSARY FOR OBJECT-ORIENTED PROGRAMMING

class. A class is a description of one or more similar objects. For example, the class *Apple*, is a description of the structure and behavior of instances, such as *apple-1* and *apple-2*. Loops and Smalltalk classes describe the instance variables, class variables, and methods of their instances as well as the position of the class in the inheritance lattice.

class inheritance. When a class is placed in the class lattice, it inherits variables and methods from its superclasses. This means that any variable that is defined higher in the class lattice will also appear in instances of this class. If a variable is defined in more than one place, the overriding value is determined by the inheritance order. The inheritance order is depth-first up to joins, and left-to-right in the list of superclasses.

class variable. A class variable is a variable stored in the class whose value is shared by all instances of the class Compare with *instance variable*

composite object. A group of interconnected objects that are instantiated together, a recursive extension of the notion of object. A composite is defined by a template that describes the subobjects and their connections.

data abstraction. The principle that programs should not make assumptions about implementations and internal representations. A *data type* is characterized by operations on its values. In object-oriented programming the operations are methods of a class. The class represents the data type and the values are its instances.

default value. A value for an instance variable that has not been set explicitly in the instance. The default value is found in the class, and tracks that value until it is changed in the instance. This contrasts with initial values.

delegation. A technique for forwarding a message off to be handled by another object.

initial value. A value for an instance variable that is computed and installed in the instance at object creation. Different systems provide initial values and/or default values.

instance. The term "instance" is used in two ways. The phrase "instance of" describes the relation between an object and and its class. The methods and structure of an instance are determined by its class. All objects in Loops (including classes) are instances of some class. The noun "instance" refers to objects that are not classes

instantiate. To make a new instance of a class.

instance variable. Instance variables (sometimes called slots) are variables for which local storage is available in instances. This contrasts with class variables, which have storage only in the class In some languages instance variables can have optional properties.

lattice. In this document we are using "lattice" as a directed graph without cycles. In Loops, the inheritance network is arranged in a lattice. A lattice is more general than a tree because it admits more than one parent. Like a tree, a lattice rules out the possibility that a class can (even indirectly) have itself as a superclass.

metaclass. This term is used in two ways: as a relationship applied to an instance, it refers to the class of the instance's class; as a noun it refers to a class all of whose instances are classes

message. The specification of an operation to be performed on an object. Similar to a procedure call, except that the operation to be performed is named indirectly through a *selector* whose interpretation is determined by the class of the object, rather than a procedure name with a single interpretation.

method. The function that implements the response when a message is sent to an object. In Loops, a class associates *selectors* with methods.

mixin. A class designed to augment the description of its subclasses in a multiple inheritance lattice. For example, the mixin *NamedObject* allocates an instance variable for holding an object's name, and connects the value of that variable to the object symbol table

object. The primitive element of object-oriented programming. Objects combine the attributes of procedures and data. Objects store data in variables, and respond to messages by carrying out procedures (methods).

perspective. A form of composite object interpreted as different views on the same conceptual entity. For example, one might represent the concept for "Joe" in terms of *JoeAsAMan*, *JoeAsAGolfer*, *JoeAsAWelder*, *JoeAsAFather*. One can access any of these by view name from each of the others.

polymorphism. The capability for different classes of objects to respond to exactly the same protocols. Protocols enable a program to treat uniformly objects that arise from different classes A critical feature is that even when the same message is sent from the same place in code, it can invoke different methods.

protocol. A standardized set of messages for implementing something. Two classes which implement the same set of messages are said to follow the same protocol.

slot. See instance variable

specialization. The process of modifying a generic thing for a specific use

subclass. A class that is lower in the inheritance lattice than a given class.

super class. A class that is higher in the inheritance lattice than a given class.

Figure 1 shows an example of a class in Loops. A class definition is organized in several parts—a class name, a metaclass, super classes, variables, and methods. The metaclass part names the metaclass (*Class* in this case) and uses a property list for storing documention. A *metaclass* describes operations on this class viewed as a Loops object. The super class part (supers) locates a class in the inheritance network. The other parts of a class definition describe the places for specifying data storage and procedures.

Truck

MetaClass Class

EditedBy *(*dgb "29-Feb-85 4:32)*

doc*(**This sample class illustrates the syntax of classes in Loops.*

Commentary is inserted in a standard property in the class. e.g. Trucks are ...)

Supers (Vehicle CargoCarrier)

ClassVariables

tankCapacity 79 doc(*gallons of diesel)

InstanceVariables

owner PIE *doc(*owner of truck)*

highway 66 *doc(*Route number of the highway.)*

milePost 0 *doc(*location on the highway)*

direction East *doc(*One of North, East, South, or West.)*

cargoList NL *doc(*List of cargo descriptions.)*

totalWeight 0 *doc(*Current weight of cargo in tons.)*

Methods

Drive Truck.Drive *doc(*Moves the vehicle in the simulation)*

Park Truck.Park *doc(*Parks the truck in a double space.)*

Display Truck.Display *doc(*Draws the truck in the display.)*

The class, called *Truck*, inherits variables and methods from both of its super classes *(Vehicle and CargoCarrier)*. The form of the definition here shows the additions and substitutions to inherited information. In this example a value for the class variable *tankCapacity* is introduced, and six instance variables *(highway, milePost, direction, cargoList, and totalWeight)* are defined, along with their default values. The Methods declaration names the procedures (Interlisp functions) that implement the methods. For example, *Truck Drive* is the name of a function that implements the *Drive* method for instances of *Truck*.

Example of a Class Definition in Loops.

Figure 1.

Variables in objects are used for storing state. Loops supports two kinds of variables: class variables and instance variables. Class variables are used to hold information shared by all instances of the class. Instance variables contain the information specific to a particular instance. Both kinds of variables have names, values, and other properties. We call the instance variable part of a class definition the *instance variable description,* because it specifies the names and default values of variables to be created in instances of the class. It acts as a template to guide the creation of instances. For example, the class *Point* might specify two instance variables, x and y with default values of 0, and a class variable associated with all points, *lastSelectedPoint.* Each instance of *Point* would have its own x and y instance variables, but all of the instances would use the same *lastSelectedPoint* class variable; any changes made to the value of *lastSelectedPoint* would be seen by all of the instances. Default values are the values that would be fetched from the instance variables if the variables have not been assigned values particular to the instance. As motivated by Smalltalk, Loops also has indexed instance variables, thus allowing some instances to behave like dynamically allocable arrays.

A class specifies the behavior of its instances in terms of their response to *messages.* A message is made of arguments and a *selector.* The class associates a selector (*e.g.*, the selector "Drive" in Figure 1.) with a *method,* a procedure to respond to the message. When a message is sent to an instance, its response is determined by using the selector to find the method in a symbol table in the class. The method is located in the symbol table and then executed. Since all instances of a class share the same methods, any difference in response by two instances is determined by a difference in the values of their instance variables.

What's in an Instance? Most objects in a Loops program are instances (that is, not classes). For example, in a traffic simulation program there may be one class named *Truck* and hundreds of instances of it representing trucks on the highways of a simulation world. All of these truck instances respond to the same messages, and share the class variables defined in the class *Truck.* What each instance holds privately are the values and properties of its instance variables, and perhaps an object name. The variables of an instance are initialized with a special token indicating to the Loops access functions that the variables have not been locally set yet. Figure 2 shows an example of an instance object.

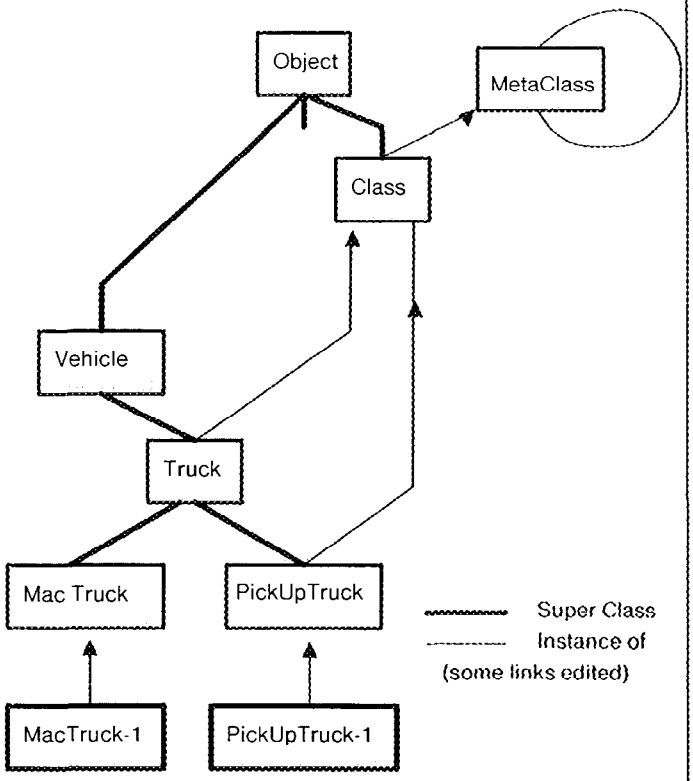
Instantiation and Metaclasses. The term *metaclass* is used in two ways: as a relationship applied to an instance, it refers to the class of the instance's class; as a noun it refers to a class all of whose instances are classes. The internal implementation of an instance is determined by its metaclass.

For example, to create a new instance of the class *MacTruck* a *New* message is sent to *MacTruck.* This creates a data structure representing the truck with space for all of the instance variables. The method for creating the

Automobile-1
Class Automobile
InstanceVariables
highway 66
milePost 38
direction East
driver Sanjay
fuel ?
...

Instances have local storage for their instance variables. If no local value has been set yet, default values for instance variables are obtained from the class. In this example, the "?" in the value place for the instance variable *fuel* indicates that the actual value is to be obtained by lookup from the class *Automobile*. Methods and class variables are not shown since they are accessed through the class.

Instance of an Automobile in a Traffic Simulation Model. Figure 2.



Instantiation is the process of making a new object. In Loops, this process begins when a *New* message is sent to a class. In this figure, the instance *MacTruck 1* is created by sending a *New* message to *MacTruck*. The line between *MacTruck-1* and *MacTruck* indicates the instance-of relationship between the instance and its class. Similarly, a class like *PickUpTruck* is created by sending a *New* message to *Class*. The line between *PickUpTruck* and *Class* indicates the instance-of relationship between the class *PickUpTruck* and its metaclass *Class*. Every object has an instance-of relationship with exactly one class. *Class* and *MetaClass* are called metaclasses, since all of their "instances" are classes. Another important metaclass is *AbstractClass* (not shown). *AbstractClass* is the metaClass for classes not meant to be instantiated (*e.g.*, like Vehicle)

Instantiation.

Figure 3.

data structure is found in *MacTruck's* metaclass—*Class*. To create a new class (*e.g.*, the class *PickUpTruck*, an instance of the class *Class*), a New message is sent to *Class*. The method for creating and installing the data structure is found in the metaclass of *Class* called *MetaClass*. The *New* method creates a data structure for the new class, with space for class variables, instance variable description, and a method lookup table. The new class is also installed in the inheritance network. Figure 3 illustrates this process of instantiation.

AbstractClass is an example of a useful metaclass in Loops. It is used for classes that are placeholders in the inheritance network that it would not make sense to instantiate. For example, its response to a *New* message is to cause an error. Other metaclasses can be created for representing some classes of objects as specialized LISP records and data structures.

Classes and Instances, Revisited.

The distinction between classes and instances is usual in object languages. In applications where instances greatly outnumber classes, a different internal representation allows economies of storage in representation. It also provides a natural boundary for display of inheritance, and hence helps to limit the visual clutter in presentations of the inheritance network.

The object language of KEE (Fikes & Kehler, 1985) does not provide distinct representations for classes and instances. All objects, or "units," as they are called in KEE, have the same status. Any object can be given "member" slots which will be inherited by instances of this object. Proponents of this more uniform approach have argued that for many applications, the distinction does little work and that it just adds unnecessary complications. Similarly there is no such distinction in actor languages (Lieberman, 1981). Inheritance of variables in these languges is generally replaced by a copying operation. ThingLab (Borning, 1979), a constraint driven object language used prototypes rather than classes to drive object creation, and specialization was simply instantiation followed by editing.

In Object Lisp (Drescher, 1985), the declarative structures conventionally associated with classes are dispensed with. Objects are simply binding environments, that is,

"closures." An operation is provided for creating and nesting these environments. Object variables are Lisp variables bound within an object environment; methods are function names bound within such an environment. To get the effect of message sending, one "ASKS" for a given form to be evaluated in the dynamic scope of a given object. Nested environments are used to achieve the layered "inheritance" effect of specialization. Use of an "ASKS" form, however, precludes the unification of message-sending and procedure call that is now appearing in other object languages.

Not all object languages have metaclasses. Since Flavors are not objects, instances of Flavors have no metaclasses. All Flavor instances are implemented the same way: as vectors. Loops uses metaclasses to allow variations in implementation for different classes. For example, some objects provide a level of indirection to their variable storage, allowing updating of the object if there are changes in its class definition. Smalltalk-80 has metaclasses, and uses them primarily to allow differential initialization at object creation time. CommonLoops (Bobrow, *et al.,* 1985) makes more extensive use of metaclasses than Loops, using them as a sort of "escape mechanism" for bringing flexibility to representation and notations of objects.

Another difference in systems is whether instance variables of an object can be accessed from other than a method of the object. Proponents of this strict *encapsulation,* as in Smalltalk, base their argument on limiting knowledge of the internal representation of an object, making the locus of responsibility for any problems with the object state well-bounded. A counter argument is that encapsulation can be done by convention. Loops allows direct access to object variables to support a knowledge representation style of programming. This is particularly useful, for example, in writing programs that compare two objects.

Not all object languages provide property annotations for variables. In Smalltalk, Flavors, and Object Lisp, variables have values and nothing more. However, languages intended primarily for knowledge engineering applications tend to support annotations. For example, KEE, STROBE (Smith, 1983), and Loops, which are all direct descendants of the Units Package, have this. Annotations are useful for storing auxiliary information such as dependency records, documentation, histories of past values, constraints, and certainty information.

In Loops, the approach to annotating variables has evolved over time. In its most recent incarnation, property annotations have been unified with a means for triggering procedure call on variable access (active values) (Stefik, Bobrow, & Kahn, 1986). These annotations are contained in objects and it is possible to annotate annotations recursively.

The distinction between class variables and instance variables varies across object languages. Smalltalk makes the same distinction as Loops (and was the source of the idea for the Loops developers). Flavors does not have class variables. KEE provides *own* and *member* declarations for slots, serving essentially the same purposes as the distinction between class and instance variables. CommonLoops provides primitives for describing when, how, and where storage is allocated for variables. From these primitives, the important notions of class variables can be defined, except that they share the same name space as other object variables.

Most object languages treat variables and methods as distinct kinds of things: Variables are for storage and methods are for procedures. This distinction is blurred somewhat by active values in Loops, which make it possible to annotate the value of a variable in any object so that access will trigger a procedure. The distinction is blurred also in languages like KEE, STROBE, and the Units Package in which methods are procedure names stored in instance variables (which they call slots). In these languages there is an additional kind of message sending: sending a message to a slot. For specified kinds of messages, the value returned can be just the value of the slot.

Another important extension in the Units Package, KEE, and STROBE is that slots are annotated by datatypes. The datatype distinguishes the kind of data being kept in the slot, be it an integer, a list, a procedure, or something defined for an application. An object representing the datatype provides specialized methods for such operations as printing, editing, displaying, and matching. These datatype methods are activated when a slot message is not handled by a procedure attached to the slot itself. The form of message forwarding and representation of datatypes as objects provides another opportunity for factoring and sharing information. Thus, what all slots of a given type share, independent of where they occur, is characterized by methods in the corresponding datatype object.

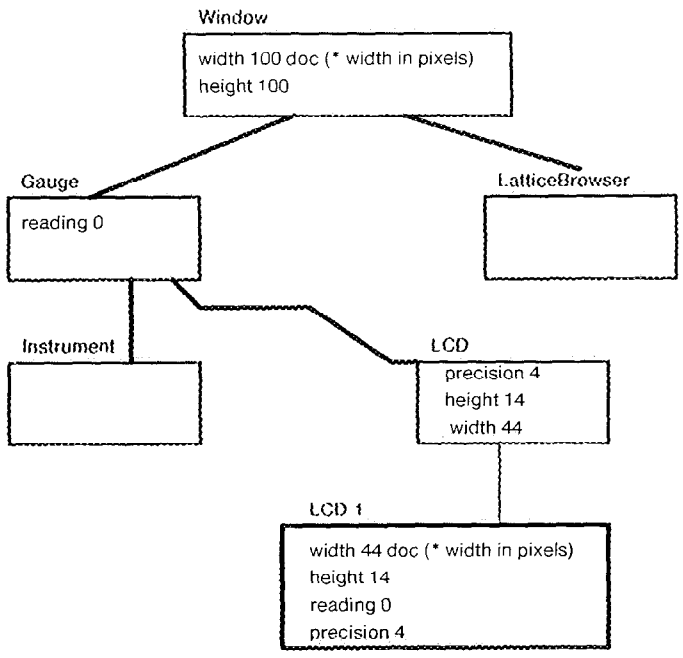
Inheritance

Inheritance is the concept in object languages that is used to define objects that are almost like other objects. Mechanisms like this are important because they make it possible to declare that certain specifications are shared by multiple parts of a program. Inheritance helps to keep programs shorter and more tightly organized. The concepts of inheritance arise in all object languages, whether they are based on specialization or delegation and copying.

We begin with the simplest model of inheritance, hierarchical inheritance. We will then consider multiple inheritance, as it is done in Loops and other languages.

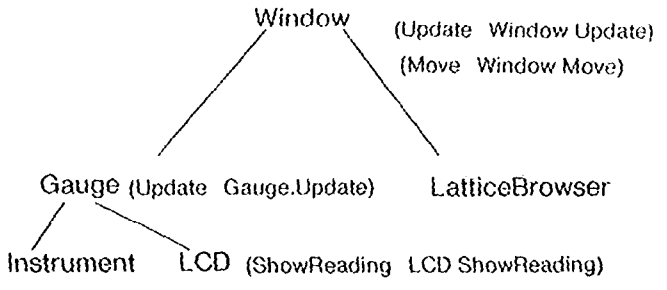
Hierarchical Inheritance.

In a hierarchy, a class is defined in terms of a single superclass. A specialized class modifies its superclass with



In hierarchical inheritance in Loops, the description of a class is inherited by every subclass, unless it is overridden in the subclass. The instances of the class *LCD* have a class precedence list for getting description *LCD, Gauge, Window, Object* The default value for the instance variable *width* is 44 (rather than 100), because *LCD* is closer in the search path than *Window*. In this example, the *doc* property of the instance variable *width* is still inherited by instances of *LCD* passing along documentation about the use of the variable.

Hierarchical Inheritance of Instance Variables. Figure 4.



In hierarchical inheritance in Loops, the description of a method is inherited by every subclass, unless it is overridden in the subclass This inheritance can be characterized as a *first-found* search through superclasses. In this figure, the Move method for the class *LCD* is inherited from *Window*.

Hierarchical Inheritance of Methods. Figure 5.

additions and substitution. *Addition* allows the introduction of new variables, properties, or methods in a class, which do not appear in one of its superclasses in the hierarchy *Substitution* (or overriding) is the specification of a new value of a variable or property, or a new method for a selector that already appears in some superclass. Both kinds of changes are covered by the following rule. All descriptions in a class (variables, properties, and methods) are inherited by a subclass unless overridden in the subclass. (See Figure 4.)

The values to be inherited can be characterized in terms of a *class precedence list* of superclasses of the class determinined by going up the hierarchy one step at a time. Default values of instances are determined by the closest class in the superclass hierarchy, that is from the first one in the class precedence list. Figure 5 illustrates essentially the same lookup process for methods.

There is always an issue about the *granularity of inheritance*. By this we mean the division of a description into independent parts, that can be changed without affecting other parts. In Loops, any named structural element can be changed independently—methods, variables, and their properties. For example, substituting a new default value for one instance variable does not affect the inheritance of the properties of that variable, or the inheritance of other instance variables. Figure 4 shows this for the independent inheritance of documentation when a default value is changed.

Several approaches for implementing inheritance are possible offering different tradeoffs in required storage, lookup time, work during updating, and work for compilers. For example, the lookup of default values need not involve a *run-time* search of the hierarchy. In Loops, the default values are cached in the class, and updated any time there is a change in the class hierarchy.

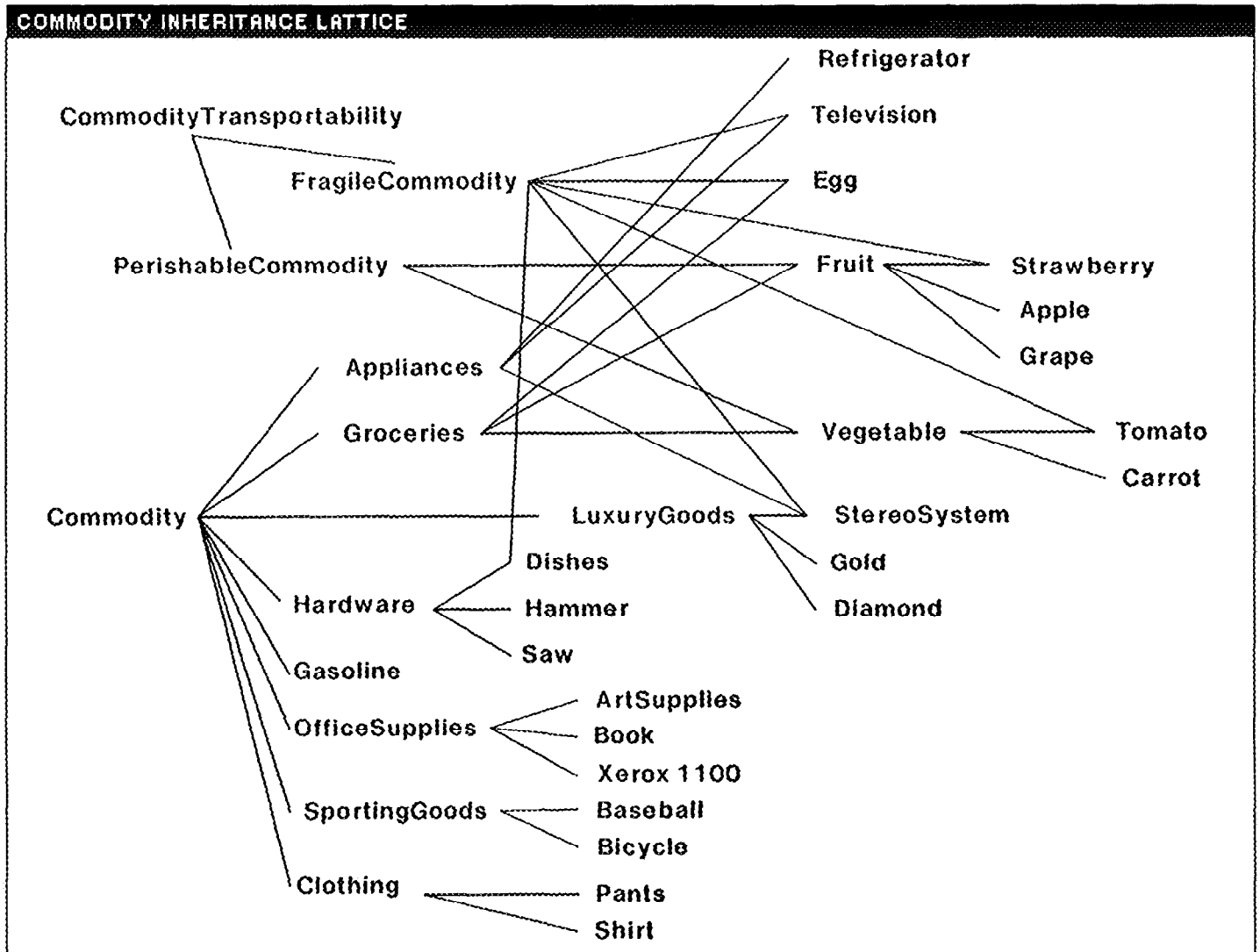
The position of a variable is determined by lookup in the class, but the position of this variable is cached at first lookup. Changes in the hierarchy affecting position of instance variables simply require clearing the cache.

In Flavors, the position of an instance variable is stored in a table associated with a method, and is accessed directly by the code. This gives faster access, but requires updating many method tables for some changes in the hierarchy.

Multiple Inheritance in a Lattice.

Inheritance is a mechanism for elision. The power of inheritance is in the economy of expression that results when a class shares description with its superclass. *Multiple* inheritance increases sharing by making it possible to combine descriptions from several classes.

Using multiple inheritance we can factor information in a way that is not possible in hierarchical inheritance. Figure 6 illustrates this in a lattice of commodities.



This lattice illustrates the use of multiple superclasses to factor inherited information in a network of classes. Multiple inheritance allows increased brevity in specifications by increasing the ability to share descriptions. For example, the class *StereoSystem* in this lattice inherits information from *LuxuryGoods*, *Appliances*, and *FragileCommodity*. In a strictly hierarchical system, it would be necessary to duplicate information in the hierarchy- for example by creating classes for *FragileAppliances* or *FragileLuxuryAppliances*. In a hierarchical scheme, the methods and variables associated with "fragility" would need to be replicated for each different use.

Multiple Inheritance in a Lattice.
Figure 6.

The class *StereoSystem* inherits descriptions from *LuxuryGoods*, *Appliances*, and *FragileCommodity*. The methods and variables describing *FragileCommodity* are also used for *Egg*, *StereoSystem*, and other classes. It would be necessary to duplicate the information about fragility in several classes such as *FragileAppliances*, *FragileGroceries*, *FragileFruit*, and *FragileLuxuryGoodsAppliances* in a strictly hierarachical system. In contrast, multiple inheritance lets us package together the methods and variables for *FragileCommodity* for use at any class in the network.

A class inherits the union of variables and methods from all its super classes. If there is a conflict, then we use a class precedence list to determine precedence for the variable description or method. The class precedence list is computed by starting with the first (leftmost) superclass in the supers specification and proceeding depth-first *up to joins*. For example, the precedence order for *DigiMeter* in Figure 7 first visits the classes in the left branch *(DigiMeter, Meter, Instrument)*, and then the right branch *(LCD)*, and then the join *(Gauge)*, and up from there.

The left-to-right provision of the precedence ordering makes it possible to indicate which classes take precedence

in the name space. The "up to joins" provision can be understood by looking at examples of mixins. Mixins often stand for classes that it would not make sense to instantiate by themselves. Mixins are special classes that bundle up descriptions and are "mixed in" to the supers lists of other classes in order to systematically modify their behavior. For example, *PerishableCommodity* and *FragileCommodity* are mixins in Figure 6 that add to other classes the protocols for being perishable or fragile. Another example of a mixin is the class *NamedInstance,* which adds the instance variable name to its subclasses, and overrides methods from *Object* so that appropriate actions in the Loops symbol tables take place whenever the value of a *name* instance variable is changed. *DatedObject* is another mixin which adds instance variables that reflect the date and creator of an object. Mixins usually precede other classes in the list of supers, and are often used to add independent kinds of behavior.

When mixins are independent, the order of their inclusion in a supers list should not matter. Like all classes in Loops, mixins are subclasses of *Object.* If the "up to joins" provision was eliminated from the precedence ordering, then the depth-first search starting from the first mixin would cause all the other default behaviors for *Object* to be inherited—interfering with other mixins later in the supers list that may need to override some other part of *Object.* Changing the order of mixins would not eliminate such interference, since most mixins need to override the behavior of *Object* in some way. The "up-to-joins" provision fixes this problem by insuring that *Object* will be the last place from which things are inherited. Although this effect could also be achieved by treating *Object* specially, we have found that analogous requirements arise whenever several subclasses of a common class are used as mixins. The up-to-joins provision is a general approach for meeting this requirement.

Multiple Inheritance, Revisited.

A major source of variation in object languages that provide multiple inheritance is their stand towards precedence relations. In Smalltalk, multiple inheritance is provided, but not used much or institutionalized. Smalltalk-80 takes the position that no simple precedence relationship for multiple inheritance will work for all the cases, so none should be assumed at all. Whenever a method is provided by more than one superclass, the user must explicitly indicate which one dominates. This approach diminishes the value of mixins to override default behavior.

Flavors and Loops both use a fixed precedence relationship, but differ in the details. The two approaches can be seen as variations on an algorithm that first linearizes the list of superclasses (using depth-first traversal) and then eliminates duplicates to create a class precedence list. In Flavors, all but the first appearance of a duplicate are eliminated. In Loops, all but the last appearance of a duplicate are eliminated.

CommonLoops takes the position that experimentation with precedence relationships is an open issue in object-oriented programming. In CommonLoops, the precedence relation for any given class is determined by its metaclass, which provides message protocols for computing the class precedence list.

Method Specialization and Combination

One way to specialize a class is to define a local method. This is useful for adding a method or for substituting for an inherited one. In either case a message sent to an instance of the class will invoke the local method. The grain size of change in this approach is the entire method.

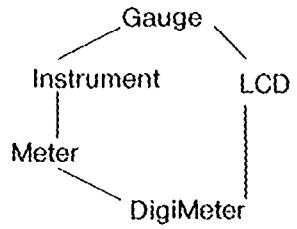
A powerful extension to this is the *incremental specialization* of methods, that is, the ability to make incremental additions to inherited methods. This is important in object-oriented programming because it enables fine grained modification of message protocols. In the following we consider two mechanisms for mixing of inherited behavior. The first mechanism ← *Super* (pronounced "send super") allows procedural combination of new and inherited behavior. It derives initially from Smalltalk and is used heavily in the Loops language. Then we will consider an interesting and complementary approach pioneered by the Flavors system in which there is a declarative language for combining methods.

Procedural Specialization of Methods.

Incremental modification requires language features beyond method definition and message sending. In Loops, ← *Super* in a method for selector M1 invokes the method for M1 that would have been inherited. Regular message sending (←) in a local method can not work for this, because the message would just invoke the local method again, recursively.

An example of its use is shown in Figure 8. In this example, *Gauge* is a subclass of *Window.* The method for updating a *Gauge* needs to do whatever the method for *Window* does, plus some initial setting of parameters and some other calculations after the update. The idiom for doing this is to create an *Update* method in *Gauge* that includes a ← *Super* construct to invoke *Window's* method. This is better than duplicating the code from *Window* (which might need to be changed), or invoking *Window's* method by procedure name (since other classes might later be inserted between *Window* and *Gauge).*

← *Super* provides a way of specializing a method without knowing exactly what is done in the higher method, or how it is implemented. ← *Super* uses the class precedence list to choose when a method appears in more than one superclass. The precedence ordering is the same as that used for object variables.



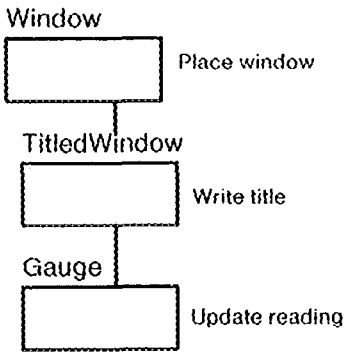
Local supers of DigiMeter: (Meter LCD)

Order of Inheritance:

(DigiMeter Meter Instrument LCD Gauge...)

In multiple inheritance it is possible to inherit things from several superclasses. The precedence of different inherited values is determined by a search as shown.

Order of Inheritance in a Lattice.
Figure 7.




```
[Gauge.Update (self)
    (* First update the guage parameters *)
  (← self SetParameters)
    (* Now update using the method from a super class *)
  (←Super self Update)
    (* Now do other things *)
      . ]
```

In this example, *Gauge* is a subclass of other classes (say *Window*), which have their own methods of updating. The *Update* method for *Gauge* needs to do whatever the method for *Window* does, except that some parameters need to be set first and then some other computations need to be done afterwards. This effect is achieved by using the ← *Super* construct, which allows embedding an invocation for *Window*'s method inside new method code for *Gauge*.

Example of Using ←Super.
Figure 8.

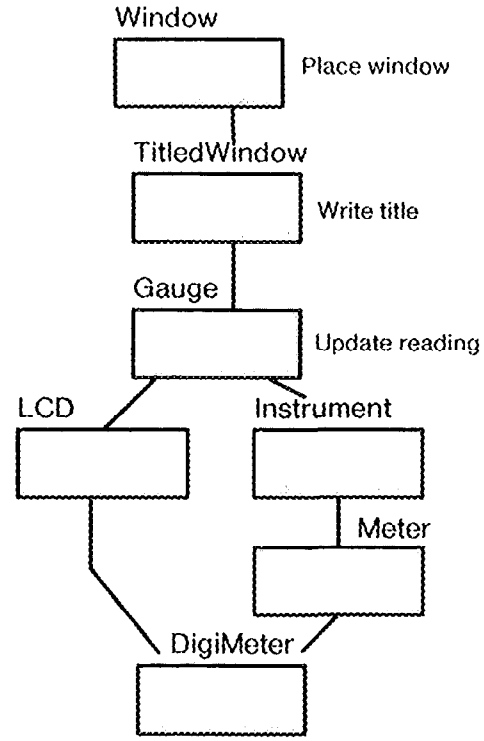
← *Super* uses the class precedence list in order to preserve the correctness of protocols under changes to the inheritance lattice. The most obvious definition of ← *Super* would be to search for the super-method from the beginning of the class precedence list. This fails for nested versions of ← *Super*, and even for a ← *Super* in a method which is not defined locally, but is inherited. A second incorrect implementation would use the class precedence list of the class in which the method was found. This gives incorrect results for classes with multiple super classes. To insure that protocols work the right way in subclasses, ← *Super* starts the search in the object's class precedence list at the class from which the current method is inherited. Because ← *Super* is defined this way, inherited methods using ← *Super* consistently locate their "super methods" and common changes to the lattice yield invariant operation of the message protocols.

Combination of several inherited methods is also important. A simple version combines all of the most-local methods for a given selector, that is, all of the methods that have not themselves been specialized. These methods are called the *fringe* methods, and the construct for invoking them all is called ← *SuperFringe*. For example, in Figure 9 the class *DigiMeter* combines the updating processes for *LCD* and *Meter* by using ← *SuperFringe* to invoke the *ShowReading* methods of its superclasses.

For selective combination of methods from different classes Loops provides a construction called DoMethod. DoMethod allows the invocation of any method from any class on any object. It can be viewed as an escape mechanism, allowing one to get around the constraints imposed by message sending. It also steps outside the paradigm of object-oriented programming and opens the door to a wide variety of programming errors. When programs are written using standard message invocations, then protocols keep working even when common changes are made to the inheritance lattice. This happy situation is not the case when programs use DoMethod. Since DoMethod allows specification of the class in which the message will be found, it encourages the writing of methods that make strong assumptions about the names of other classes and the current configuration of the inheritance lattice. Programs that use DoMethod are likely to stop working under changes to the inheritance lattice.

Declarative Method Combination

Flavors supports a declarative language for combining methods at compile time. An important new distinction made in Flavors is that there can be three named parts to a method—a *before* part, an *after* part, and a *main* method, each of which is optional. By default, the *main* method




```
[DigiMeter.ShowReading (self)

    (* Show displayVal both on dial and on digits )

    (←SuperFringe self ShowReading reading)

    ]
```

In this example, *DigiMeter* combines the classes of *LCD* and *Meter*. To show a reading, a *DigiMeter* must carry out the *ShowReading* methods in both its *LCD* portion and its *Meter* portion. This combination of protocols can be done by using the ← *SuperFringe* construct to invoke the *ShowReading* methods for all of the superclasses. The method above invokes the original *ShowReading* methods of both *Meter* and *LCD*

Example of Using ←SuperFringe.
Figure 9.

overrides any inherited main method, but the before and after parts are all done in a nested order determined by the class precedence list. Thus a supplier of a method in a mixin can ensure that whatever the main method, its before method will be executed.

A declarative language used in a newly defined method can specify other than the default behavior for combination of inherited method parts. For example, *and* combination of before methods allows the execution of the entire method to stop if one of the before methods returns *nil*. The *defwhopper* combinator allows a compile time construction of the equivalent of ← *Super*.

The concept of organizing methods and variables into classes that can be mixed together for use in combination admits at least two distinct philosophies for assigning responsibility for the viability of the combination. In Flavors responsibility is assigned, at least in part, to the suppliers, that is, to the classes that are being combined. Combinator specifications include things like do this method before the main method, or do it after the main method, or do parts of it at both times. The intention is to get the specification right once in the supplier so that consumers need not know about it. When this kind of specification is successful, it reduces the total amount of code in the system since consumers need only specify the order of superclasses.

In Loops, responsibility for method combination is assigned to the consumer; that is, the local method uses the procedural language and the special form ← *Super* to combine the new behavior with behavior inherited.

It is important to consider the effects on program change when evaluating alternatives like this. How often are suppliers changed? consumers? To what extent are suppliers independent? Do mixins need notations for indicating what kinds of classes they are compatible with? What kinds of changes in the suppliers require changes in the consumer classes?

CommonLoops takes the position that both philosophies are worth exploring, and that continued experimentation in the refinement of method combinators is called for. It provides a primitive RunSuper (analogous to the Loops ← *Super*) in the kernel. Methods are represented as objects in CommonLoops; this means that a system can have different kinds of methods with different techniques for installing them or displaying their sources. Flavor-style methods would be a special kind of object with extra specifications for combinators. These specifications would be interpreted at appropriate installation and reading times by Flavors-style discriminator objects for those methods.

Composite Objects.

A composite object is a group of interconnected objects that are instantiated together, a recursive extension of the notion of object. A composite is defined by a template that describes the subobjects and their connections. Facilities for creating composite objects are not common in the object languages we know, although they are common in application languages such as those for describing circuits and layout of computer hardware. The current Loops facility is based on ideas in Trillium (Henderson, 1986), which is a language for describing how user interfaces for copiers are put together.

Principles for Composite Objects.

Composite objects in Loops have been designed with the following features:

- Composite objects are specified by a class containing a description indicating the classes of the parts and the interconnections among the parts.

The use of a class makes instantiation uniform so that composite objects are "first class" objects.

- Instantiation creates instances corresponding to all of the parts in the description.

The instantiation process keeps track of the correspondence between the parts of the description and the parts in the instantiated object. It fills in all of the connections between objects. It permits multiple distinct uses of identical parts.

- The instantiation process is recursive, so that composite objects can be used as parts.

For programming convenience, the instantiation process detects as an error the situation where a description specifies using another new instance of itself as a part, even indirectly. Instantiation of such a description would result in trying to build an object of unbounded size. An alternative is to instantiate subparts only on demand. This allows the use of a potentially unbounded object as far as needed.

- It is possible to specialize a description by adding new parts or substituting for existing parts.

This reflects the central role of specialization as a mechanism for elision in object-oriented programming. The language of description allows specialization of composite objects with a granularity of changes at the level of parts.

An Example of a Composite Object.

Composite objects are objects that contain other objects as parts. For example, a car may be described structurally as consisting of a body, a power system, and an electrical system. The body has two doors, a hood, a chassis, and other things. Parts can themselves contain other parts: a door has various panels, a window, and a locking system. Objects can also be parts of more than one container: the fan belt can be viewed as a component of the cooling system or of the electrical charging system.

The boxed figure in the next column shows the Loops class definition of *Mercedes240D* defined as a composite object.

Mercedes240D is a subclass of the mixin *CompositeObject* that supports protocols for instantiation that will interpret descriptions of parts. The value of the instance variable *engineSystem* will be filled by an instance of the class *DieselEngine*. In that instance of *DieselEngine*, the value of the instance variable *numCylinders* is initialized to 4 and transmission to *4Speed*.

The *body* instance variable of the *Mercedes240D* will be initialized to and instance of *Body300*. Its instance

```
Mercedes240D
MetaClass Class
  EditedBy (*dgb "15-Feb-82 14:32")
  doc (*This class is a CompositeObject representing a
car and its parts.)
Supers (CompositeObject Automobile)
ClassVariables
  Manufacturer DaimlerBenz
  StandardCarStuff ((color
(@color))(owner(@owner))...)
InstanceVariables
  yearManufactured NIL
  owner NIL
  style traditional
  color ivory
  engineSystem NIL
        part (DieselEngine (numCylinders 4)
              (transmission (QUOTE 4Speed))...)
  body NIL
        part (Body300
              (style (@style)
              (color (@color)
          StandardCarStuff))
```

variable *style* is set to the value of the *style* from the *Mercedes240D*, that is, *traditional*. In addition, the *color* property of the *style* instance variable will be set to ivory. These exemplify the propagation of values from the containing instance to those parts contained in it.

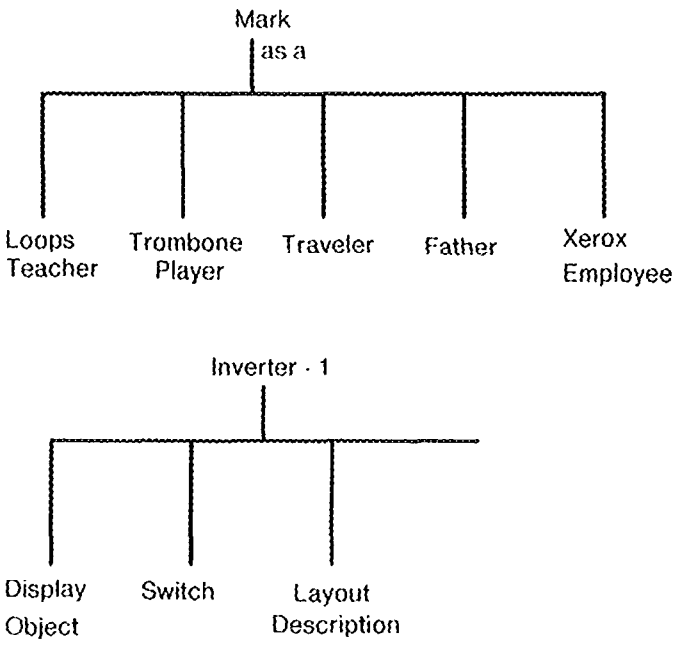
The class variable *StandardCarStuff* indicates a number of variables for the body part that will inherit values from the car. For example, the *color* of the body is the *color* of the car. Finally, the instance variable parts will be set to a list of all the immediate parts of the *Mercedes240D*. If any of the parts are themselves *CompositeObjects*, their parts will be instantiated too.

Perspectives.

Perspectives are a form of composite object interpreted as different views on the same conceptual entity. For example, one might represent the concept for "Joe" in terms of views for *JoeAsAMan, JoeAsAGolfer, JoeAsAWelder, JoeAsAFather*. We will first describe perspectives as they are used in Loops, and then contrast this with other languages.

Perspectives in Loops.

Perspectives in Loops are implemented by independent linked objects representing each of the views. One can access any of these by view name from each of the others.



For some applications, it is important to have different perspectives or views of an entity, with independent name spaces. In the upper portion of this figure, the entity *Mark* is shown with perspectives for *LoopsTeacher*, *TrombonePlayer*, *Traveler*, and so on. Each perspective offers a different view of *Mark* in a distinct object with its own variables and methods. For example, the *TrombonePlayer* perspective would contain information relevant to Mark's ability to make music. In the example of an inverter, shown in the lower part of the figure, variables with the same name, such as *xCoordinate* and *yCoordinate*, may have one meaning in the perspective *DisplayObject* and another meaning in the perspective *LayoutDescription*

Perspectives.
Figure 10.

Because the linked objects are independent, the same instance variable name in more than one of the objects can mean different things, and can be changed independently. For example, Figure 10 illustrates an object *Inverter-1* which has the perspective *DisplayObject* as well as the perspective *LayoutDescription*. Both perspectives may have instance variables named *xCoordinate* and *yCoordinate*, but with different interpretations. For a *DisplayObject*, the variables could refer to the coordinates in pixels on a workstation display. For a *LayoutDescription* perspective, the variables could refer to coordinates in the silicon chip on which the inverter is fabricated.

Perspectives were designed in Loops to have the following properties:

- Perspectives are accessed by perspective names.

Given an object, one can ask for its Traveler perspective using the name "Traveler." A given perspective name has at most one perspective of an object. Perspectives form a kind of equivalence class.

- Perspectives are instantiated on demand.

This contrasts with usual composite objects in which all the parts are created at instantiation time. Additional views can be added as needed to any object.

Perspectives can be compared with class inheritance. In inheritance only one variable is created when there is a "coincidence" in the names of variables inherited from different superclasses. Thus inheritance assumes that the same name is always intended to refer to the same variable. For perspectives, variables of the same name from different classes are used for different views and are distinct. When classes are combined by inheritance, all of the instances of the combined class have the same structure (that is, variables and methods) and all of the structure is created at once. For perspectives, the situation is different. Instances have varying substructure.

Variations on Perspectives.

The term perspective was first used for different views of the same conceptual object in KRL, and later in PIE (Goldstein & Bobrow, 1980). Each view had an independent name space for its slots. However, in neither PIE nor KRL was a perspective a full-fledged object; access to the view could only be obtained through the containing object.

Although the terminology of perspectives is not widespread, some other object languages (*e.g.*, Snyder, 1985) have a similar capability to combine the structure of multiple classes in this way. Snyder suggests that name coincidence in multiple inheritance ought not imply identity. He believes that this violates an important encapsulation principle of object oriented programming—that users of objects ought not to have privileged access to the internals of those objects. He extends that notion to classes which inherit from previously defined classes. For his language, inheritance from a super class means the embedding of an instance of the super class in the subclass. Messages of the super class are to be inherited explicitly, and implemented by passing the message on to the embedded instance.

Examples of Object-Oriented Programming

Examples of programming can be presented at several levels. This section considers three examples of object-oriented programming that illustrate important idioms of programming practice. The first illustrates the use of message sending and specialization. The second example illustrates choices among techniques of object combination. The third example illustrates common techniques for redistributing information among classes as programs evolve.

Programming the Box and the BorderedBox.

Object-oriented programming has been used for many programs in interactive graphics. The following example was

motivated by these applications. We will consider variations on a program for displaying rectangular boxes on a display screen. This example explores the use of message sending and specialization in a program that is being extended and debugged.

Figure 11 gives our initial class definition for the class *Box*. Instances of this class represent vertically aligned rectangular regions on a display screen. The four instance variables store the coordinate and size information of a box. The origin of a box in the coordinate system is determined by the variables *xOrigin* and *yOrigin* and the default origin is at (100, 200). The size of a box is determined by variables *xLength* and *yLength* and the default size is 10 x 30. Operations on a box include moving it to a new origin, changing its size, and changing the shading inside the box. In the following we will specialize the *Box* class and also uncover a bug in it.

Message protocols define an interface for interacting with boxes. Instances of *Box* are created by sending it a *New* message. Size and position of an instance are established by sending it a *Reshape* message. Shade is established by sending it a *Shade* message. These messages provide a structured discipline for interaction with boxes, that is, a data abstraction. Outside agents need only know the relevant messages. They need not know the implementation of a box in terms of its instance variables.

Suppose that we wanted to create another kind of box with a visible border that frames it in the display. This *BorderedBox* would be essentially a *Box* with a border. This suggests that we employ inheritance and specialize the class definition of *Box*.

In programming *BorderedBox* several choices about the interpretation and representation of the border need to be made. The foremost question is about the treatment of coordinates of the border, that is, whether the border frames the outside of the box or is included as part of the box. For example, is the border included in the length measurements? If the border is on the outside, is the origin on the inside or the outside of the border? The answers to these questions do not come from principles of object-oriented programming, but rather from our intentions about the meaning of the *BorderedBox* program. The answer affects the meaning of the instance variables *xOrigin, yOrigin, xLength,* and *yLength* inherited from *Box*. For this example, we will assume that the borders are intended only to make the boxes easier to visualize in the

Box

MetaClass Class

EditedBy *(* dgb "31-September-84 11:23")*

doc *(* Rectilinear box that can be displayed.)*

Supers (DisplayObject)

InstanceVariables

xLength 10 *doc (* length of the horizontal side.)*

yLength 30 *doc (* length of the vertical side.)*

xOrigin 100 *doc (* x coordinate of origin lower left corner.)*

yOrigin 200 *doc (* y coordinate of origin lower left corner.)*

Methods

Move Box.Move	*doc (* Moves box (change origin)in the display)*
	args (newXOrigin newYOrigin)
Reshape Box.Reshape	*doc (* Changes the location and axes of the box.)*
	args (newXOrigin newYOrigin newXLength newYLength)
Shade Box.Shade	*doc (* Fills the inside of the box with a new shade.) args (newShade)*
Draw Box.Draw	*doc (*Displays the box.)*

Instances of this class represent vertically aligned rectangular regions on a display screen The four instance variables store the coordinate and size information for a box. For example, the origin in the coordinate system is determined by the variables *xOrigin* and *yOrigin* and the default origin is at (100, 200) Operations on the box are defined by messages to the box. They include moving it to a new origin, changing the size of the box, and changing the shading inside the box.

Class Definition for the Box.
Figure 11.

display and that for this purpose they will be treated as part of the box.

The next step is to decide whether any of the methods of *Box* need to be specialized in *BorderedBox*. Since a border needs to be redrawn when a box is increased in size, it is clear that at least the *Reshape* method needs some revision. Figure 13 shows a specialized *Reshape* method that uses ← *Super* to invoke the *Reshape* method from *Box*. The specialized *Reshape* also invokes local methods to *Draw* and *Erase* the boundary. These methods plus one for setting the size of the boundary must be added to *BorderedBox*. The *Draw* and *Erase* methods are for internal use, but the *SetBorder* method will become part of the external protocol. Figure 12 shows these methods together with a new instance variable for recording the size of the border.

The use of a variable for *borderSize* brings up a question of how the methods of the original *Box* class work for shading. In fact, they cannot work if the shade is not saved as part of the state of an instance (or is otherwise computable). Box's *Reshape* method should use the current shade in order to fill new areas when a box is expanded. To fix this deficiency, we can now go back to the definition of *Box* to add a *shade* instance variable that will be saved by the *Shade* method. We can also modify *Box's Reshape* method to use this new variable.

After the shade bug is fixed, we should ask whether the specialized class *BorderedBox* must also be changed. *BorderedBox* will inherit the shade instance variable and the revised Shade method. Furthermore, the specialized Reshape method in *BorderedBox*, which uses ← *Super*, will effectively "inherit" the shade changes from *Box's Reshape* method. In this example, the inheritance mechanisms of the language work for us in just the right way. This illustrates how language features can provide leverage for accommodating change.

Programming the DigiMeter.

Gauges are favorite pedagogical examples in Loops because they use features of both object-oriented and access-oriented programming. They are defined as Loops classes and are driven by active values.

Figure 14 illustrates a collection of gauges in Loops. Gauges are displayed in a *window*, an active rectangular region in the bitmap display. They have a black title bar for labels and a rectangular center region in which they display values. Instances of *LCD* (for "little character display") show their values digitally, but most gauges simulate analog motion to attract visual attention when they change. For example, subclasses of *VerticalScale* and *HorizontalScale* simulate the movement of "mercury" as in a thermometer. Instances of subclasses of *RoundScale* move a "needle" in a round face.

For some purposes it is convenient to combine digital and analog output in a single gauge. The digital output makes it easy to read an exact value from the gauge. The analog output makes it easy to notice when the gauge is changing and to estimate the position of the current value in a fixed range. With gauges like this it is easy to tell at a glance that something is "half full."

The programming of combination gauges gives rise to a choice of programming techniques for combining classes. Figure 14 shows the *DigiMeter* as an inheritance combination of a *Meter* and an *LCD*. Such a gauge needs to combine the programmed features of the two classes. In the following we will consider the arguments for choosing an appropriate technique of object combination.

Here are some goals bearing on the design of a *DigiMeter*:

- The *DigiMeter* should respond in the standard way to gauge protocols.

For example, a single request to *Set* the *DigiMeter* should suffice, without having to send separate messages for the meter and the LCD. Both gauges should display the same correct value.

- The *DigiMeter* should use a single window to display both gauges.

The combination gauge should not have two separate windows and bars. The component gauges should appear in a single window on the display screen that is large enough for both of them.

- The combined description should make direct use of the classes for *LCD* and *Meter*.

DigiMeter should use some method for combining *Meter* and *LCD*. This does not preclude making changes to the classes for *Meter* and *LCD* in order to make them compatible for combination, but we do not want to duplicate code or descriptions in *DigiMeter*. The class descriptions should continue to work whether the classes are used alone or in combination.

The three techniques of object combination supported in Loops are perspectives, composite objects, and multiple inheritance.

Using *perspectives* for combination, we would create a *DigiMeter* with one perspective for the meter and one for the LCD. Unfortunately, the direct approach to this would result in the creation of separate windows for each gauge. We could fix this for all of the gauges in the lattice, for example, by making a window be a perspective of a gauge. The main utility of perspectives is that they support switching among multiple views and instantiating these views on demand. In this application, we always need to create all of the views and the views are very close-

BorderedBox

MetaClass Class

EditedBy *(* mjs "1-Oct-84 01:67")*

doc *(* Like a Box except displays a black border. The origin is the outside of the border.)*

Supers (Box)

InstanceVariables

borderSize 2 *doc (* width of the border.)*

Methods

Reshape BorderedBox.Reshape

SetBorder BorderedBox.SetBorder *doc (* Set a new border size.) args (newBorderSize)*

EraseBorder BorderedBox.EraseBorder

DrawBorder BorderedBox.DrawBorder

A *BorderedBox* is like a box except that it is drawn with a variable sized border in the display. *BorderedBox* is implemented by specializing *Box* A new instance variable *borderSize* is added to record the size of border. The width of the border is included as part of the dimensions of the box.

Class Definition for the BorderedBox.

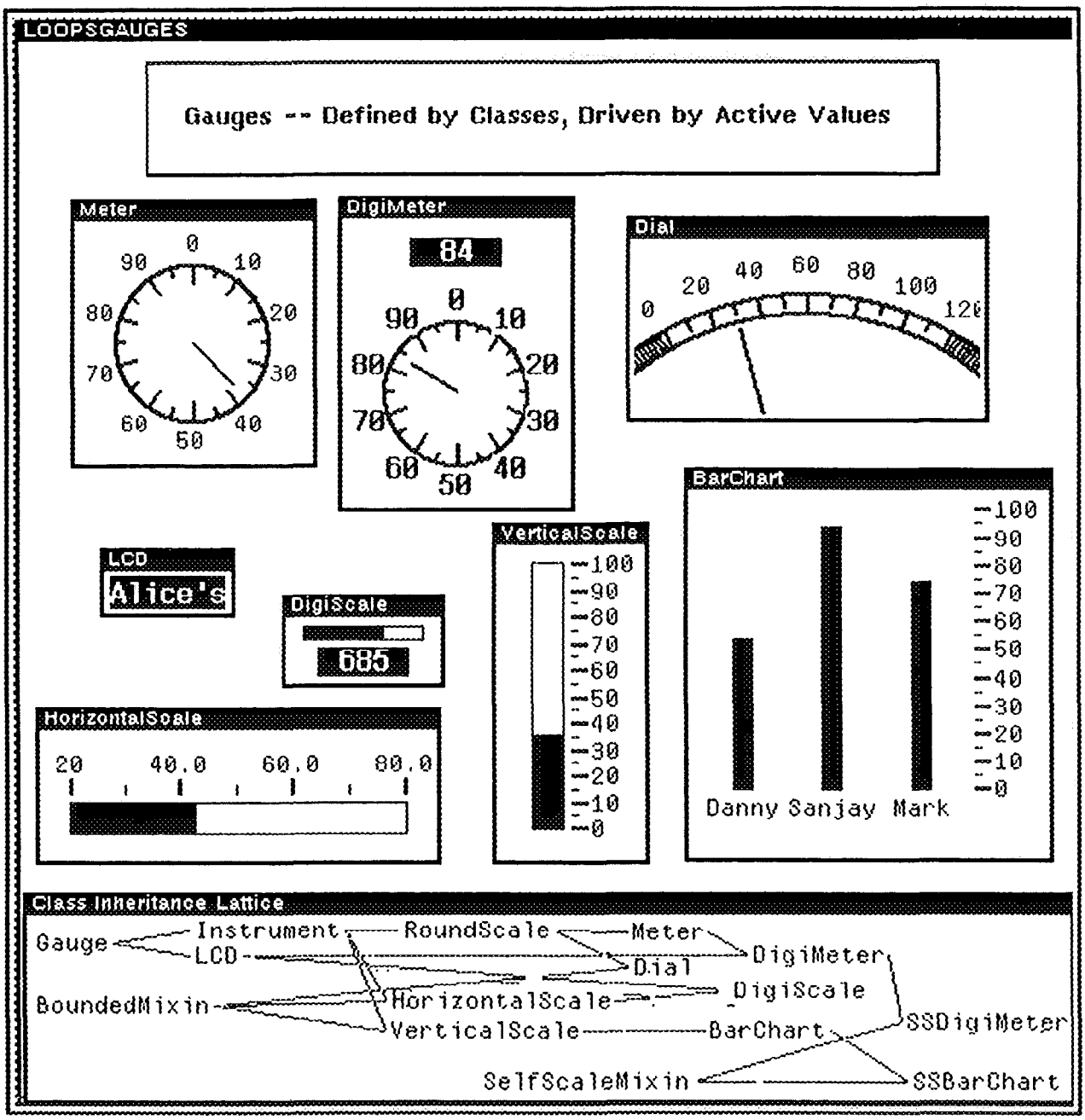
Figure 12.

```
(BorderedBox.Reshape (self newXOrigin newYOrigin newXLength newYLength)

        (← self EraseBorder) (* Erase old border.)

        (* Now Reshape box as before)

        (← Super self Reshape newXOrigin newYOrigin newXLength newYLength)

        (← self DrawBorder) (* Draw new border.)

        ]
```

The specialized Reshape method needs to redisplay a revised border when the shape of the box changes. ← *Super* is used to invoke the *Reshape* method from *Box*. The specialized *Reshape* also invokes methods to *Draw* and *Erase* the boundary. These methods plus one for setting the size of the boundary must be added to *BorderedBox*. The *Draw* and *Erase* methods are for internal use and the *SetBorder* method is part of the external protocol.

Reshape Method for BorderedBox.

Figure 13.



Gauges are tools used to monitor the values of object variables. They can be thought of as having probes that can be inserted on to the variables of an arbitrary Loops program. Gauges are defined in Loops as classes and driven by active values the computational mechanism behind access-oriented programming in Loops. A browser at the bottom of the figure illustrates the relationships between the classes of gauges. From this figure, we can see that the *DigiMeter* is a combination of a *Meter* and an *LCD*. Other kinds of gauges, mimicing oscilloscope traces or chart recorders, would also be useful.

Gauges in Loops.
Figure 14.

ly associated. Hence, the main features of perspectives don't do much work for us. Using *composite objects* as the method combination, we would create a *DigiMeter* with a meter as one part and an *LCD* as another part. Again, the straightforward combination would yield a separate window for each gauge. As before, we could revise all of the gauges in the lattice, perhaps treating a window as a part of a gauge. In addition, the *DigiMeter* description would need to identify the window parts of the meter and LCD as referring to the same window. The main benefit of composite objects is to describe for instantiation a richly connected set of objects and to differentiate between objects and their parts. In this application, the connections between the parts are relatively sparse and the part/whole distinction doesn't do much work for us.

Using multiple inheritance for combination, we would create a *DigiMeter* as a class combining an LCD and a meter. Since the *LCD* and *Meter* classes inherit their window descriptions from the same place, multiple inheritance yields exactly one window. As in the other cases, we may need to tune parts of the window description to make sure that it is large enough for both gauges, but this is a straightforward use of the inheritance notion. In multiple inheritance it is important to ask whether same-named variables in the combined class refer to the same thing. For this application, we need to be on the alert for the use of variables in *Meter* and *LCD* that have the same name but different meanings, but there are no such conflicts in this case.

The preceding arguments suggest that multiple inheritance is the most appropriate technique of object combination for this application. The next step in designing a *DigiMeter* is to understand and design the interactions between the constituents. The main interactions are:

- The window should be large enough to accommodate both gauges.
- The methods for displaying both gauges should be invoked together.

The first interaction can be handled by specializing the method *(UpdateParameters)* that establishes the window parameters. The major window sizing constraints come from the *Meter,* which must provide room for the calibrated circle and its interior needle. In the Loops implementation the *DigiMeter* method uses ← *Super* to invoke the parameter-setting code for the *Meter* and then revises them to allow extra room at the top of the window for the *LCD.*

The second interaction can be handled by specializing the *ShowReading* method for showing a reading. As shown earlier in Figure 9, this method consists of a simple application of ← *SuperFringe* which invokes the original *ShowReading* methods of both *Meter* and *LCD.* In Flavors this would have involved the application of a *progn* method combinator.

The Evolution of Classes—Gauge Examples.

Most of our applications of Loops take place in a research environment in which new goals and ideas are always surfacing. In such an environment frequent revisions and extensions are a constant part of programming. To cover the kinds of reorganizations that we carry out in our work we have developed some idioms for systematic program change. This section considers three cycles of revision in the design of Loops gauges. Each cycle of revision has the following steps:

- A new goal or requirement is introduced for the design.
- A conflict in the current organization is recognized between sharing of code and flexibility.
- A new factoring of information is chosen to ease the conflict.

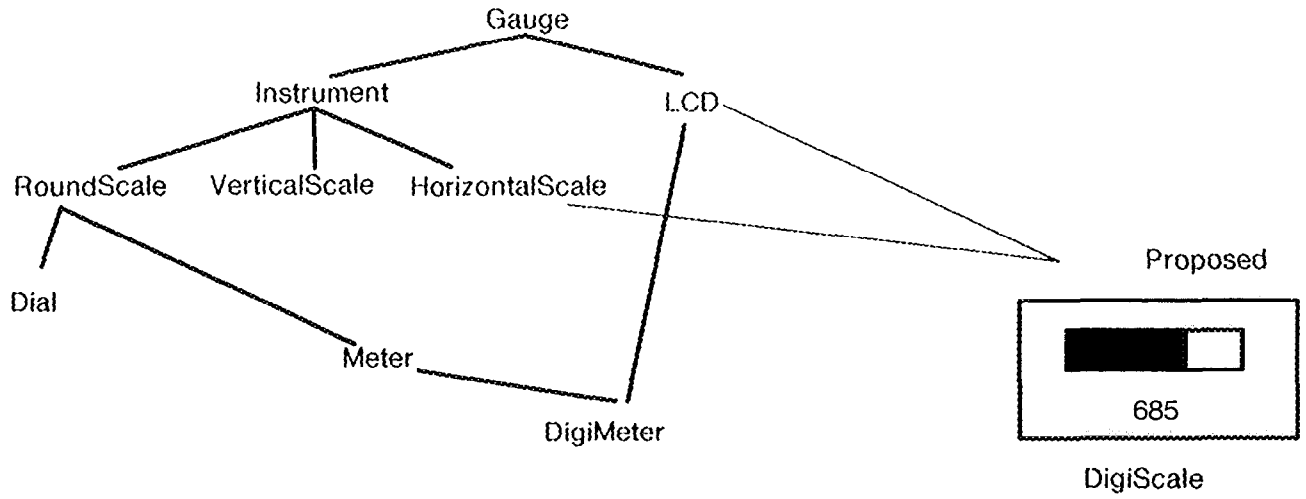
Cycle 1. In our first example we will consider the addition of a *DigiScale* to the class inheritance lattice. A *DigiScale* will be a combination of a HorizontalScale and an *LCD* as in Figure 15. A major design constraint for this example is that the *DigiScale* must be visually compact. To make it small we want to omit the tick marks and labels from the horizontal scale portion. Such a gauge would present both an exact digital value and an analog indication of the value within its range.

However, the plan of omitting the tick marks also interacts with the inheritance of existing code from it HorizontalScale and *Instrument.* In particular, the display of instruments is governed by the *ShowInstrument* method of *Instrument,* which carries out the following sequence of steps:

- Draw the instrument structure (circular dials, an so forth)
- Draw and label the tick marks.
- Print the scale factor.

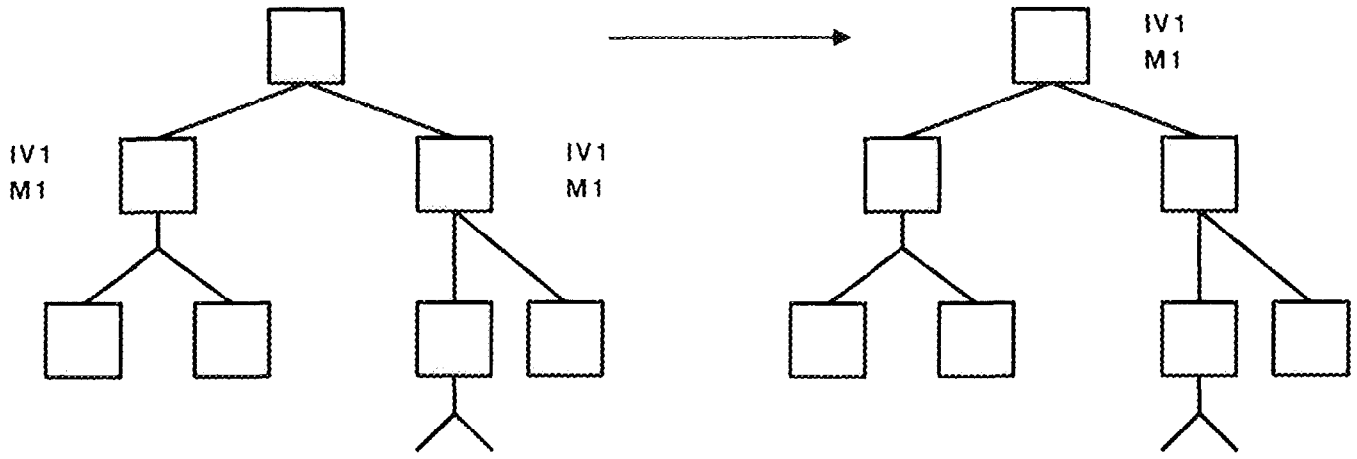
For the *DigiScale* this organization is too coarse. This illustrates a common situation where inheritance in a subclass requires finer granularity of description than was provided in the super class. The situation arises often enough in our programming that we have a name for it—a *grainsize conflict.* In Loops, pieces of description which are intended to be independently inherited must be independently named—*e.g.*, methods have their own selectors and instance variables have their own variable names. A specific fix in this case is to decompose the *ShowInstrument* method into several smaller methods that we can independently specialize, reorder, or omit.

Cycle 2. Sometimes a grainsize conflict is the first stage in recognizing new possibilities in a design. In the previous example, we considered the creation of a special kind of horizontal scale that has no tick marks. We could generalize that idea to have vertical scales or even round scales without tick marks. Another observation in



The proposed *DigiScale* will be a combination of a *HorizontalScale* and an *LCD* An unusual programming constraint in this case is that we want to omit the tick marks from the horizontal scale portion. At issue is the fact that the code inherited from *Instrument* bundles in one chunk the drawing of the gauge and the drawing and labeling of the tick marks.

Programming the DigiScale.
Figure 15.



Often there is a motivation to move structure up in the class lattice in order to increase sharing. We call this *promotion* of structure. In this case a method M_1 and an instance variable IV_1 are initially duplicated in two sibling classes. Promotion would move them to their common superclass.

Promoting Methods and Variables.
Figure 16.

the same vein is that the round scale gauges differ from the others in the way that they indicate their values. Round scale gauges use a needle. Vertical and horizontal scale gauges use space filling—like the sliding of a column of mercury. Several other kinds of gauges are possible—such as a PieScale gauge—a round scale gauge that uses an

expanding "slice of pie" to indicate its value.

This suggests that there are some independent properties of gauges that we could recognize:

- *Calibration*—gauges can have tick marks and scale factors or not.
- *Indicator style*—gauges can use needles or space filling to present their values.

The recognition that a particular distinction arising in a subclass can be generalized is a common occurrence in object-oriented programming. Often there is a motivation to move structure up in the lattice to increase the amount of sharing. We call this *promotion* of structure. Figure 16 illustrates a simple case of this where a method M^1 and an instance variable IV^1 are initially duplicated in two sibling classes.

Promotion would move them to their common superclass. The Loops environment encourages and facilitates such activities by making them easy to do with interactive browsers that show the inheritance structure, and allowing menu driven operations to make changes.

Figure 17 shows a first attempt to organize a class lattice for these distinctions. In this attempt, instruments are partitioned into *CalibratedInstrument* and *UncalibratedInstrument*. This partitioning tries to exploit the observation that the best-looking uncalibrated instruments are also the space-filling ones. The classes *VerticalGraph, HorizontalGraph,* and *PieGraph* are created as uncalibrated space-filling gauges. The main problem with this approach is the duplication of code. For example, code is duplicated between *HorizontalScale* and *HorizontalGraph,* and between *VerticalScale* and *VerticalGraph.* This leads to a different proposal for a lattice as shown in Figure 18.

In the second proposal, a mixin is created for the code that generates tick marks and labels. The gauge lattice appears essentially the same as before the reorganization except that the classes are now uncalibrated. Classes like *VerticalGraph* have calibrated subclasses like *VerticalScale* that use the *CalibratedScale* mixin. The mixin establishes the procedural connection between instrument drawing and tick mark drawing. Each subclass also supplies specialized local methods for arranging the tick marks and labels.

Cycle 3. Gauges have upper and lower bounds for the values that they display. When data go out of range, the standard behavior is to light up an "out of range" indicator and to "pin" the gauge to the maximum or minimum value. This highlights a nuisance with analog gauges. Their readings become useless when data go out of range. One idea is to have gauges automatically recompute their extreme points and scaling factors as needed. For example, if a gauge goes out of bounds, it could automatically increase the maximum reading by about 25 percent of the new high value subject to some constraints of display aesthetics.

The rescaling requirement is *independent* of the style of display, that is, it is independent of whether we are spinning a needle or driving mercury up and down. This suggests using a uniform technique for revising the scale for all the gauges. The natural choice for additive behavior is a *mixin.*

Unfortunately there is a difficulty in doing this for the *BarChart.* The *BarChart* is unique among the gauges in Figure 14 in that it displays several values at once. A *SelfScalingMixin* could be easily defined that would work for all of the gauges except the *BarChart.* This mixin would just use the value of the gauge in computing a new maximum. For a *BarChart,* it is necessary to look at all of the bars to determine the maximum. This seems to lead to the following design choices for using mixins:

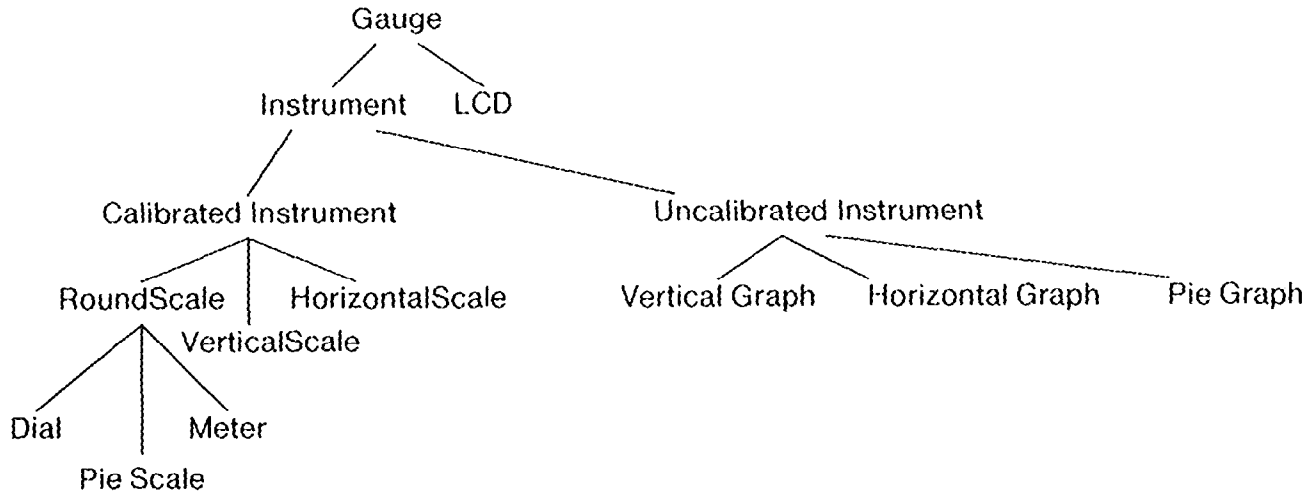
- We could design two mixins. One for the *BarChart* and one for all the other gauges. (Equivalently we could just mandate that the gauge mixin should not be used with *BarChart.*)
- We could design one mixin that worked differently for the *BarChart* and the other gauges. The method for computing the maximum would need to check whether it was being used in a class with *BarChart* as a super class.
- We could modify the definition of the single-value gauges by adding a method to simply return the value when asked for the "maximum" value.

The first two choices do not extend well if we later add additional multi-value gauges. In our Loops implementation, we chose the third option.

Usually we think of mixins as classes that we can mix in with any class whatsoever. For example, when the *DatedObject* mixin is added to class it causes instances to have a *date* instance variable initialized with their date of creation. Some kinds of mixins are designed to be used with a more limited set of classes. This example illustrates a case where the "mixees" can fruitfully be modified slightly to accommodate the mixin. The modification broadens the set of classes with which the mixin is compatible.

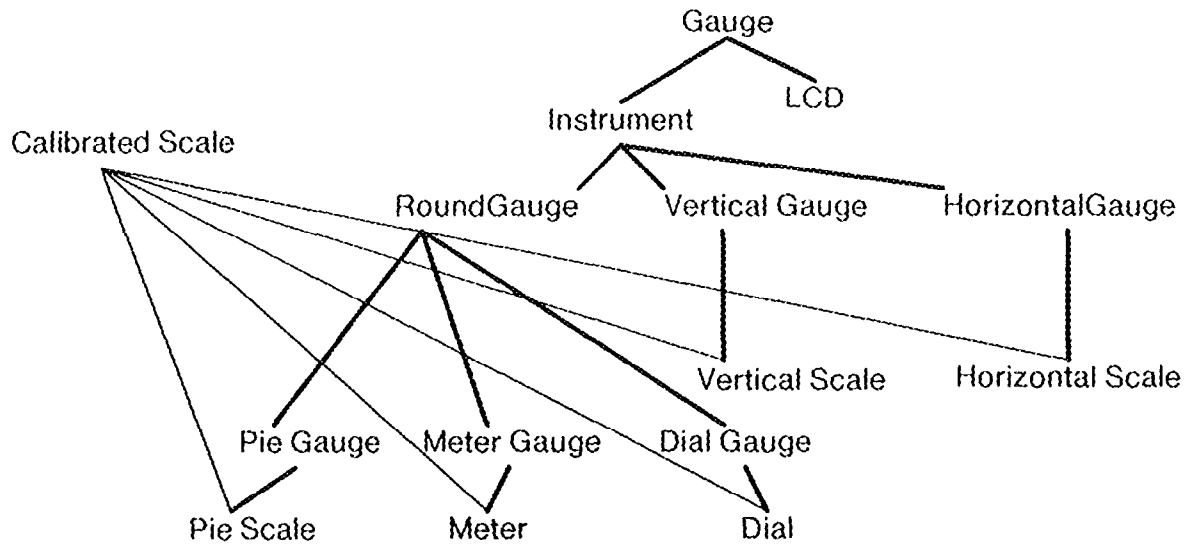
Conclusion and Summary

Objects are a uniform programming element for computing and saving state. This makes them ideal for simulation problems where it is necessary to represent collections of things that interact. They have also been advocated for applications in systems programming since many things with state must be represented, such as processes, directories and files. Augmented by mechanisms for annotation, they have also become important in the current tools for knowledge engineering.



This lattice shows gauges partitioned into *CalibratedInstrument* and *UncalibratedInstrument* which exploits the observation that the best-looking uncalibrated instruments are also the space-filling ones. However, code is duplicated, for example, between *HorizontalScale* and *HorizontalGraph*.

Rearranging the Lattice of Gauges.
Figure 17.



In this arrangement, a CalibratedScale mixin is created for tick marks and labels. In this reorganization there is a *CalibratedScale* mixin which is added to uncalibrated classes to create tick marks. For example, *VerticalScale* is a subclass of *VerticalGraph* and uses the *CalibratedScale* mixin.

Rearranging the Lattice of Gauges, Again.
Figure 18.

As object languages have become widespread, considerable interest has been expressed in developing standards so that objects could be used as a portable base for programs and knowledge bases. Towards this end, the Common Lisp Object-Oriented Programming SubCommittee is now considering several proposals to extend Common Lisp with objects.

The diversity of language concepts discussed here suggests that research is very active in this area. Standards will need to provide the kind of open-endedness and flexibility that enables languages to endure.

As object-oriented programming has taken hold in the mainstream of AI languages, they have reinforced a more general principle. There are multiple paradigms for programming. Procedure-oriented programming and object-oriented programming are but two of a larger set of possibilities that includes: rule-based programming, access-oriented programming, logic-based programming, and constraint-based programming. Different paradigms are for different purposes and fill different representational niches.

In this article we have not tried to describe all of the ways in which features of object oriented programming have been achieved in the context of other systems. For example, logic programming has inspired a number of interesting mergers. In Concurrent Prolog (Shapiro, 1983), objects are represented by processes, and messages are passed to the process along a stream. Delegation is used to achieve the effect of inheritance. In Uranus (Nakashima, 1982) objects are bundles of axioms in a database. Inheritance is done by following links in the databases, using a logic-based language to express the methods.

In Uniform (Kahn, 1981) objects are represented by expressions, and methods as operations on objects that would unify with the "head" of the method. Inheritance is implemented by viewing one expression as another (through an axiom that states, for example, that (SQUARE X) is equivalent to (RECTANGLE X X)). This has the nice property that "inheritance" can go in both directions—from specialization to super, and from super with the right parameters to specialization.

Languages that combine multiple paradigms gracefully are known as hybrid or integrated languages. Languages that succeed less well might be called "smorgasbord" languages. In any case, language paradigms are no longer going their separate ways and attempting to do all things. Separate paradigms now co-exist and are beginning to co-evolve.

References

Bobrow, D G., Kahn, K , Kiczales, G , Masinter, L , Stefik, M , & Zdybel, F (1985) CommonLoops: Merging Common Lisp and Object-oriented programming Xerox Palo Alto Research Center: Intelligent Systems Laboratory Series ISL-85-8, August 1985

Bobrow, D.G , and Stefik, M (1981) The Loops manual. Tech Rep. KB-VLSI-81-13 Knowledge Systems Area Xerox Palo Alto Research Center

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

Borning, A (1979) A constraint-oriented simulation laboratory Stanford University. Stanford Computer Science Department Report: STAN-CS-79-746

Dahl, O.J & Nygaard, K. (1966) SIMULA—an algol-based simulation language *Communications of the ACM* 9: 671-678

Drescher, G L (1985) The ObjectLisp USER Manual (preliminary) Cambridge: LMI Corporation.

Henderson, D A (1986) The Trillium user interface design environment (submitted to Computer Human Interaction Boston: April.)

Fikes, R , & Kehler, T. (1985) The role of frame-based representation in reasoning *Communications of the ACM* 28:9, pp 904-920

Goldberg, A , & Robson, D (1983) *Smalltalk-80: The language and its implementation.* Reading, Massachusetts: Addison-Wesley

Goldstein, I , & Bobrow, D G , (1980) *Descriptions for a programming environment.* AAAI-1

Goldstein, I P , & Roberts, R B. (1977) NUDGE, a knowledge-based scheduling program. IJCAI-5, pp 257-263.

Kahn, K , (1981) *Uniform—A Language based upon Unification which unifies (much of) Lisp, Prolog, and Act 1* IJCAI-7, pp 933-939.

Lieberman, H , (1981) A Preview of Act 1. Massachusetts Institute of Technology. Artificial Intelligence Laboratory Memo No 625 June

Minsky, M A (1975) A framework for representing knowledge In P Winston (Ed), *The psychology of computer vision* New York: McGraw-Hill.

Nakashima, H (1982) Prolog/KR—language features Proceedings of the First International Logic Programming Conference Marseille, France: ADDP-GIA.

Rees, J A , Adams, N I , & Meehan, J R (1984) The T Manual, 4th edition Tech Rep. Yale University January 1984

Shapiro, E , & Takeuchi, A (1983) Object oriented programming in concurrent prolog. *New Generation Computing* 1, 25-48

Smith, R G (1983) Structured object programming in STROBE Schlumberger-Doll Research: Artificial Intelligence Publications: AI Memo No 18 September

Snyder, A. (1985) Object-oriented proposal for Common Lisp ATC-85-1 Palo Alto: Hewlett Packard Laboratories

Stefik, M , Bobrow, D G , & Kahn, K (1986) Integrating access-oriented programming into a multi-paradigm environment To appear in *IEEE Software*

Stefik, M (1979) *An examination of a frame-structured representation system* IJCAI-79, pp. 845-852

Weinreb, D , & Moon, D , (1981) Lisp Machine Manual. Symbolics Inc