1. INTRODUCTION

LOOPS integrates several programming paradigms to facilitate the design of artificial intelligence applications.

- Object-oriented programming, in which information is organized in terms of objects. Every object belongs to a cass, and the classes are arranged in an inheritance lattice which allows complex objects to be described simply. Objects communicate with each other by sending messages. When an object receives a message, it performs some action, which can include sending messages to other objects.
- Procedure-oriented programming, in which smaller subroutines build larger procedures and in which data and instructions are kept separate.
- Access-oriented programming, in which accessing a value triggers an action. This paradgm is useful to monitor certain values.
- Rule-oriented programming, in which programs are organized around recursively composable sets of pattern-action rules. These rules provide a convenient way to describe flexible responses to a wide range of events. This part of LOOPS is included in the users' modules.

As a new user of LOOPS, you first must become familiar with its terminology and with the fundamental concepts described by that terminology. This chapter presents the terminology and related concepts.

1.1 Introduction to Objects

This section shows the LOOPS hierarchy, called a lattice, in Figure 1-1, and describes the key terms in separate subsections. Terms appear in order of increasing complexity, with simpler terms described first and subsequent terms building on these simpler terms.

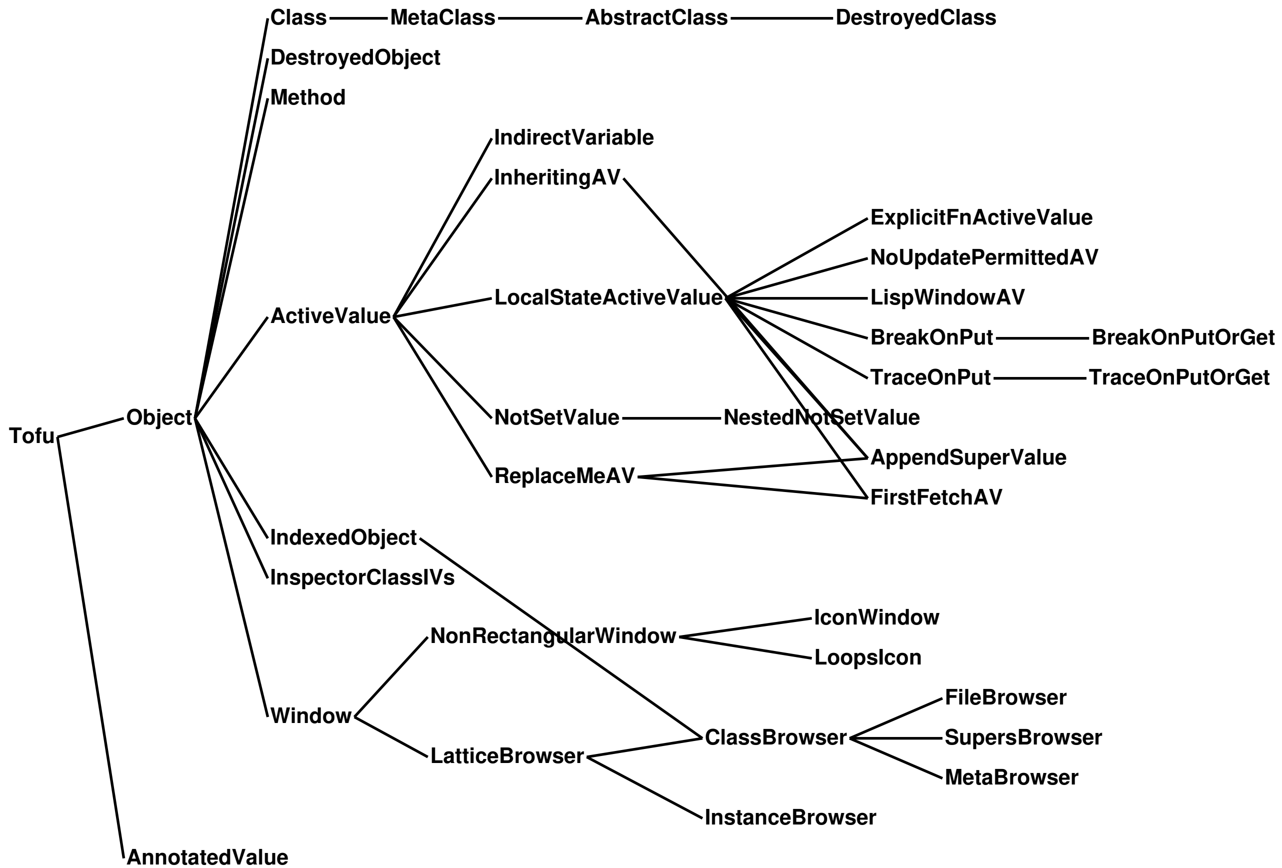

Figure 1-1. LOOPS Lattice

1.1.1 Object

As shown in Figure 1-2, an object is a structure consisting of data and a pointer to functionality that can manipulate the data. In procedure-oriented programming, data and functionality are considered as separate entities.

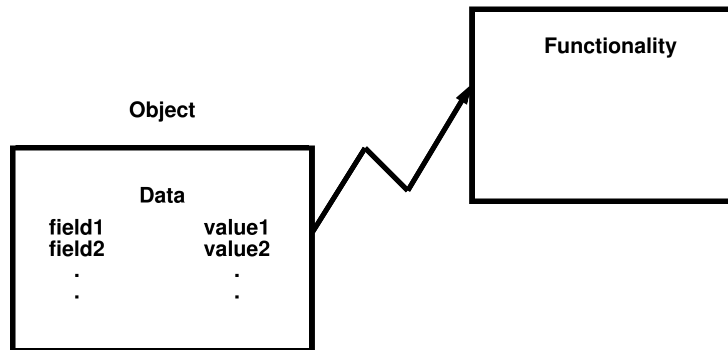

Figure 1-2. An Object

1.1 INTRODUCTION TO OBJECTS

1.1 INTRODUCTION TO OBJECTS

1.1.2 Message

Sending messages to objects provides an alternative to invoking procedures or calling functions. An object responds to a message by computing a value to be returned to the sender of the message, as shown in Figure 1-3. As a side effect, the data within an object may change during the computation. Messages contain a selector for the desired functionality. Messages may also contain arguments, as do procedures.

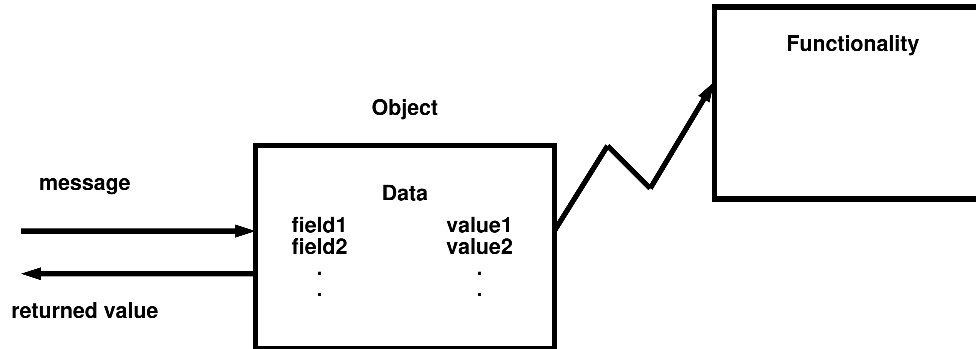

Figure 1-3. An Object Responding to a Message

1.1.3 Method

When an object receives a message, it determines what functionality it must apply to the arguments of the message. This functionality is called a method and is very similar to a procedure. A key concept that distinguishes methods from procedures is that in procedure-oriented programming, the calling routine determines which procedure to apply. In object-oriented programming, you determine the message to send and the object receiving the message determines the method to apply.

1.1.4 Selector

A message that is sent to an object contains a selector. The object uses the selector to determine which method is appropriate to apply to the message arguments. As shown in Figure 1-4, when an object receives a message with a specific selector, the object searches a lookup table containing selectors and methods to find the method associated with that particular selector.

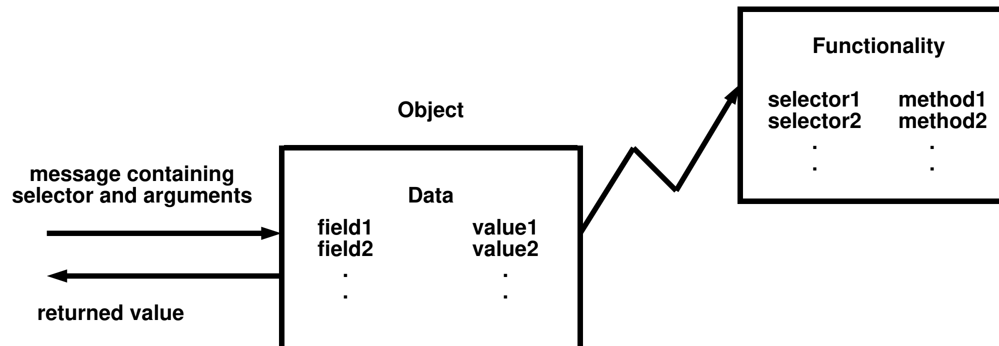

Figure 1-4. A Message Containing a Selector

1.1.5 Class

A class describes objects that are similar; that is, objects containing the same type of data fields and responding to the same messages, as shown in Figure 1-5. Think of the class that describes an object as being a template

for the functionality of its objects. When an object is sent a message, the class that describes that object handles the message, not the object itself. Different objects of the same class can respond to messages in the same way; that is, they apply the same method in response to receiving the same message.

To create new objects, send a message to a class requesting that a new object be created. Classes respond to messages because they are also objects.

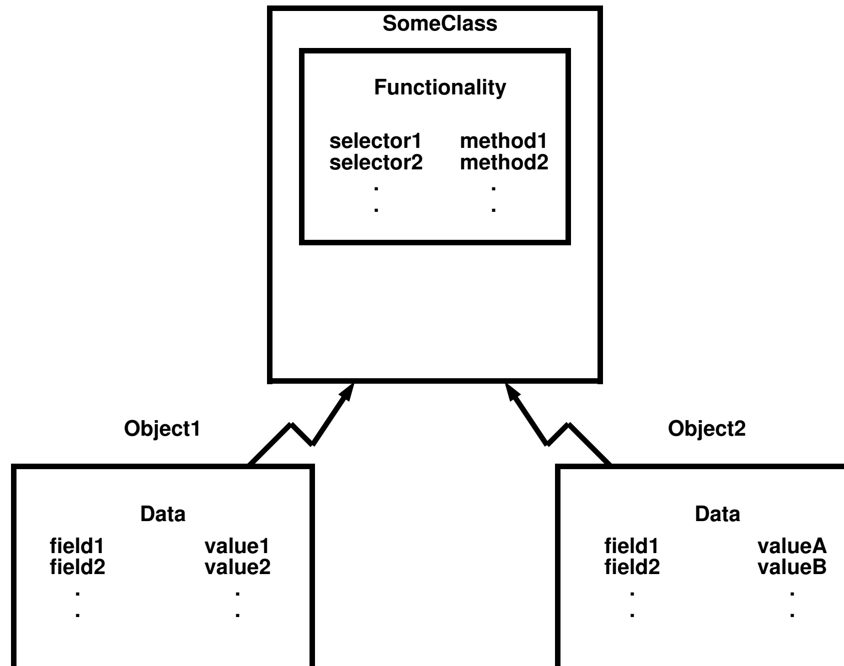

Figure 1-5. Class with Several Objects

1.1.6 Instance

An instance is an object described by a particular class. Every object within LOOPS is an instance of exactly one class.

1.2 Storage of Data in Objects

The data associated with an object is called an object's variables. Methods can change the values of these variables.

1.2 STORAGE OF DATA IN OBJECTS

1.2.1 Class Variables and Instance Variables

LOOPS supports two kinds of variables:

- Instance variables, often abbreviated IVs.

  Instance variables contain the information specific to an instance.

- Class variables, often abbreviated CVs.

  Class variables contain information shared by all instances of the class. A class variable is typically used for information about a class taken as a whole.

Both kinds of variables have names, values, and other properties. For example, the class for **Point** could specify two instance variables, **x** and **y**, and a class variable, **lastPoint**, used by methods associated with all points.

For any particular instance, you can access the values for the instance variables specific to that instance. You can also access the values for the class variables that are available to all instances of the same class.

Determining the value of a class variable requires a similar lookup procedure to that occurring when searching for a method to execute. Instance variable values are stored within the instances, and class variable values are stored within the class.

A class describes the structure of its instances by specifying the names and default values of instance variables, as shown in Figure 1-6. In this way, when a message is sent to a class to create a new instance, LOOPS can determine from the class description the number of instance variables for which it must allocate space the the initial values for those variables.

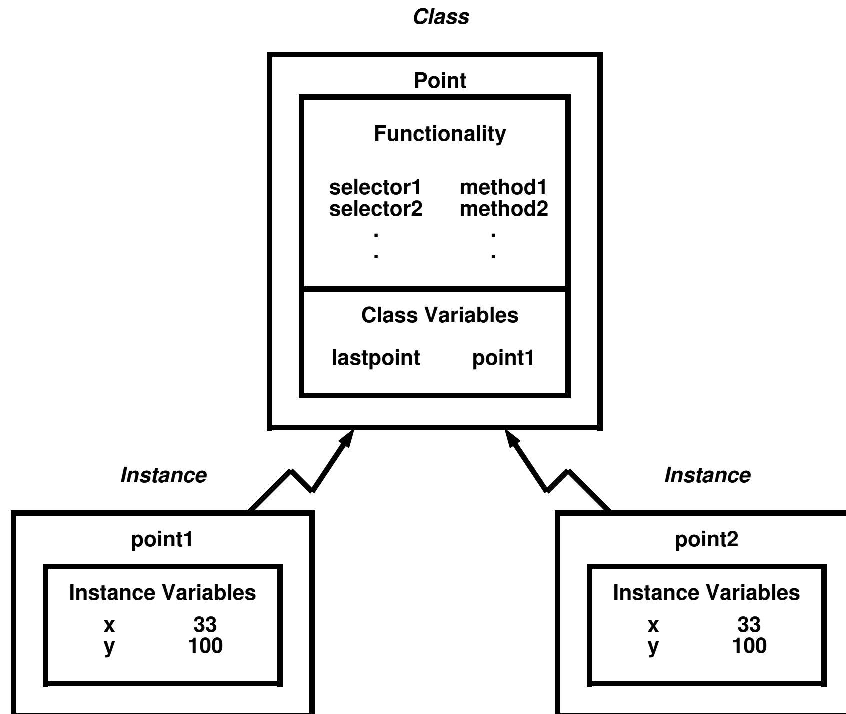

Figure 1-6. Class Variables and Instance Variables

1.2.2 Properties

LOOPS provides extensible property lists for classes, their variables, and their methods. Property lists provide places for storing documentation and additional kinds of information. For example, in a knowledge engineering

application, a property list for an instance variable could be used to store the following information:

- Support (reasons for believing a value)
- Certainty factor (numeric assessments of degree of belie■)
- Constraints on values
- Dependencies (relationships to other variables)
- Histories (previous values)

1.3 Metaclasses

Classes themselves are instances of some class. Metaclasses are classes whose instances are classes. When a class is sent a message, its metaclass determines the response. For example, instances of a class are created by sending the class the message **New**. This message is handled by the class that describes the class receiving the message. For most classes, this method is provided by the standard metaclass for classes **Class**.

To create a new class, send a message to the class **Class**. The class that handles this message is **MetaClass**. Instances of **MetaClass** are classes that describe objects which are classes. Instances of **Class** are classes whose instances are not classes. Figure 1-7 shows an instance of **MetaClass**, which is a class named **Class**, and instances of Class, which are named **Window** and **Point**.

Another class available in the system in **AbstractClass**. This is useful when creating classes that implement general functionality, which must then be specialized into instantiable classes. Instances of this class are classes that are impossible to instantiate. An example of an **AbstractClass** is **ActiveValue**, which is described in Chapter 5, Active Values.

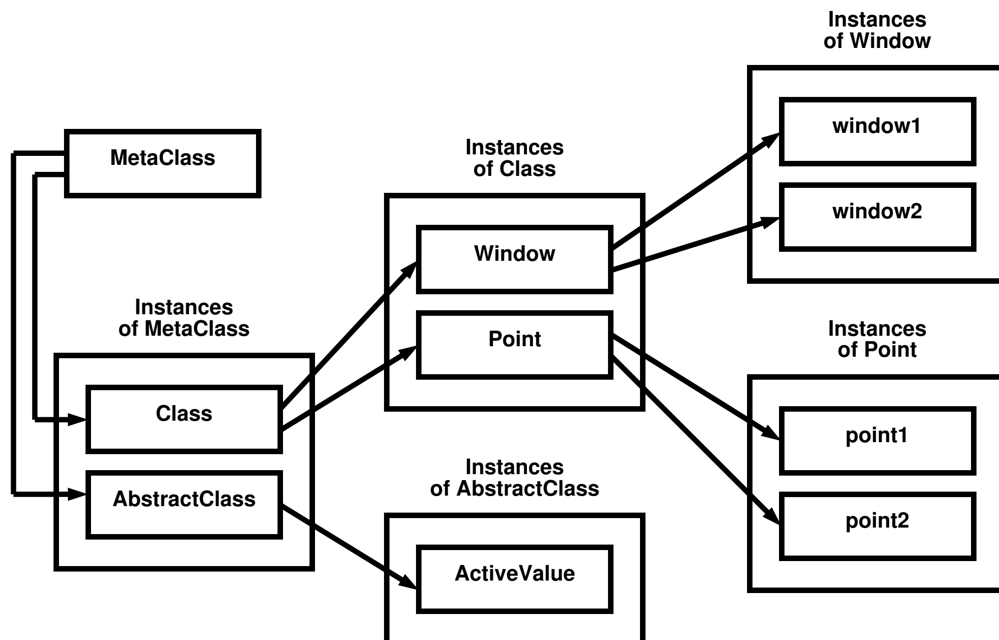

Figure 1-7. A Metaclass and its Instances

1.4 Introduction to Inheritance

Inheritance allows you to organize information in objects. With a few incremental changes, you can use inheritance to create objects that are almost like other objects. Inheritance allows you to avoid specifying redundant information and simplifies updating, since information that is common to several objects need be changed in only one place.

LOOPS objects exist in an inheritance network of classes. Figure 1-8 shows an example in which a class **3DPoint** is a subclass of another class **Point**. Instances of **3DPoint** contain instance variables that are defined in

1.4 INTRODUCTION TO INHERITANCE

Point as well as **3DPoint**. **Point** is referred to as a superclass of **3DPoint**. When an instance of **3DPoint** is created, the instance variables it contains and the messages to which it responds are not limited to those instance variables or methods as defined in the class **3DPoint**. For example, the object **pt2** contains three instance varriables; two of them are inherited from the class **Point** and the other defined in the class **3DPoint**. This instance can also respond to three different messages containing one of the three different selectors: **selector1**, **selector2**, or **selector3**.

All descriptions in a class are inherited by a subclass unless overridden in the subclass. For methods and class variables, this is implemented by a runtime search for the information, looking first in the class, and then at the superclasses specified by its supers list. For instance variables, no search is made at run time. Default values are cached in the class, and are updated if any superclass is changed, thus maintaining the same semantics as the search. Each class can specify inheritance of structure and behavior from any number of superclasses.

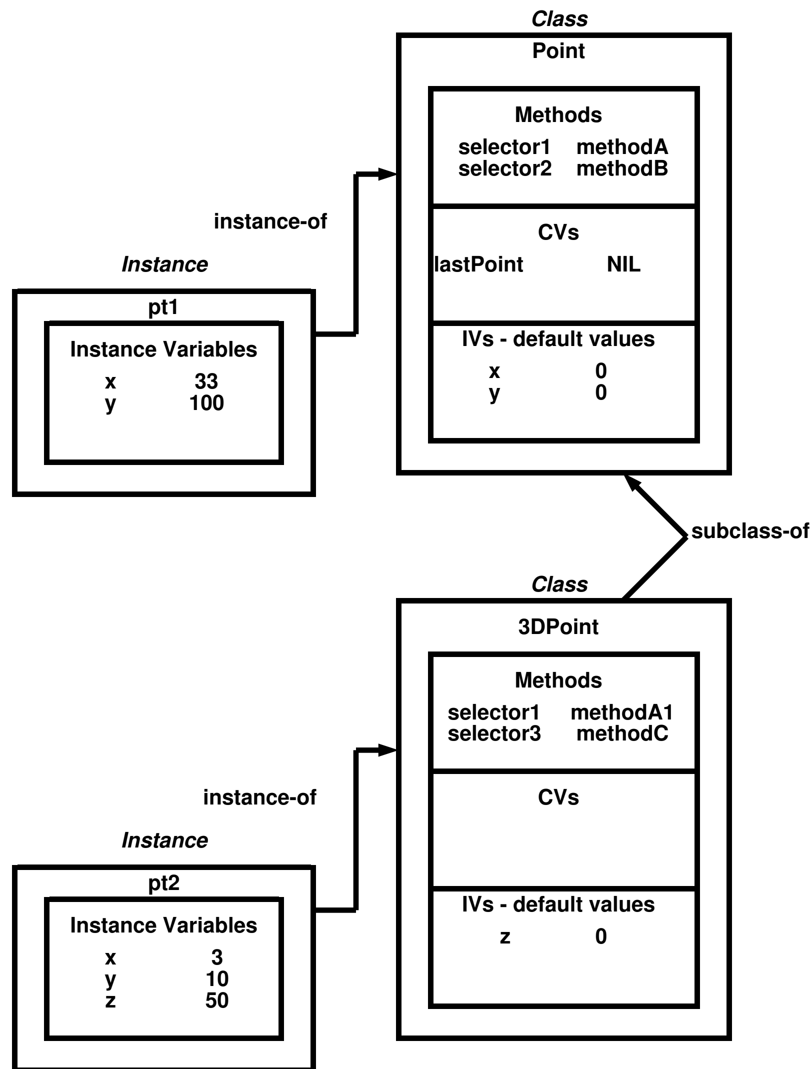

Figure 1-8. A Sample Inheritance Network

1.4.1 Single Superclasses

In the simplest case, each class specifies only one superclass. If the class **A** has the supers list **(B)**, which is a one-element list containing **B**, then all of the instance variables specified local to **A** are added to those specified for **B**, recursively. That is, **A** gets all those instance variables described in **B** and all of **B**'s supers. For example, in Figure 1-9, **A** has instance variables **x**, **z**, and **B1**.

Any conflict of variable names is resolved by using the description closer to **A** in traversing up the hierarchy to its top at the class **Object**. Method lookup uses the same conflict resolutiion. The method to respond to a message is obtained by first searching in **A**, and then searching recursively in **A**'s supers list. For example, in Figure 1-9, the method **selector2** uses **methodA2** instead of **methodB2**.

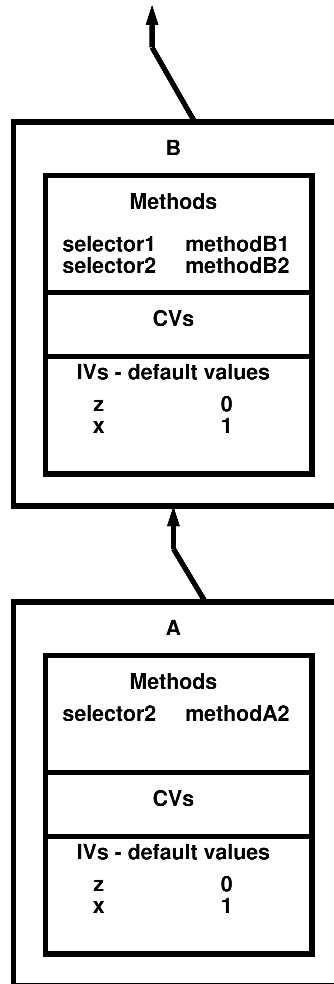

Figure 1-9. A Class with a Single Superclass

1.4.2 Multiple Superclasses

Classes in LOOPS can have more than one class specified on their supers list. Multiple superclasses permit a modular programming style where the following conditions hold:

- Methods and associated variables for implementing a particular feature are placed in a single class.
- Objects requiring combinations of independent features inherit them from multiple supers.

As in Figure 1-10, if **A** has the supers list **(B C)**, first the description from **A** is used, then the description from **B** and its supers is inherited, and finally the description from **C** and its supers. In the simplest usage, the different features have unique variable names and selectors in each super. In case of a name conflict, LOOPS uses a depth first left-to-right precedence.

For example, if any super of **B** had a method for **selector3**, then it would be used instead of the method **methodC3** from **C**. In every case, inheritance from **Object** is only considered after all other classes on the recursively defined supers list. The general rule is left-to-right, depth first, up to where the separate branches of the hierarchy join together; that is, up to any class that is repeated. Alternatively, consider the list as generated by listing all the

superclasses in a depth first left-to-right order, eliminating all but the last occurrence of a class in the list.

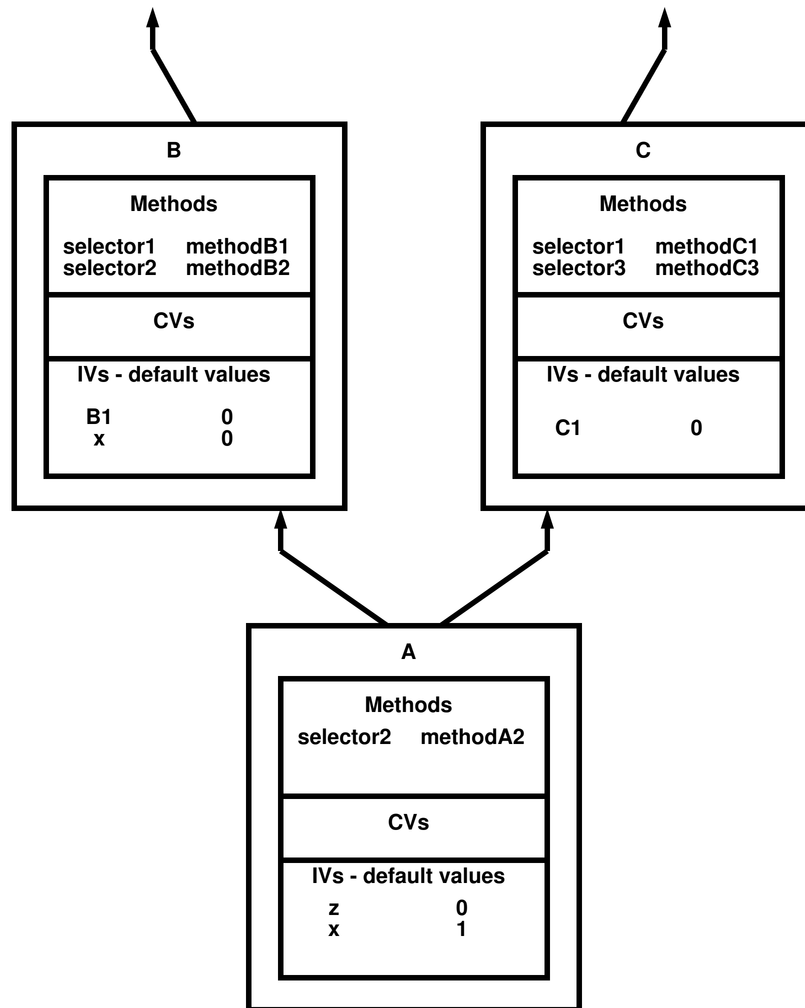

Figure 1-10. A Class with Multiple Superclasses

1.5 Introduction to Access-Oriented Programming: Using Active Values

In access-oriented programming, you can specify a particular procedure to invoke for reqad or write access of any variable of an object. LOOPS checks every object variable access to determine whether the value is marked as an active value. An active value is a LOOPS object. If a variable is marked as an active value, then aa message is sent to the active value object whenever the variable is read or set. This mechanism is dual to the notion of sending messages. Messages are a way of telling objects to perform operations, which can change their variables as a side effect. Active values are a way of accessing variables, which can send messages as a side effect.

The messages sent to the active value object will depend on the type of access If you try to read a variable, the message **GettingWrapped Value** is sent to the active value object. If you try to set a variable, the message **Putting Wrapped Value** is sent. The object receiving the message may or may not trigger side effects as the result of receiving these messages. In this way, you have control over the side effects that may occur as a result of accessing data.

Active values enable one process to monitor another one. For example, LOOPS has debugging tools that use active values to trace and trap references to object variables. A graphics module updates views of particular objects on a display when their variables are changed. In both cases, the monitoring process is invisible to, and isolated from, the monitored process. No changes to the code of the monitored object are necessary to enable monitoring.

Active values can also be used to maintain constraints among data in a system. As one piece of data changes, the active value associated with that data can contain functionality that updates other data within the system. Examples of this are spreadsheets or electric circuit modeling.

A powerful feature of active values is that they can be nested to yield a natural composition of the access functions.

1.6 Introduction to the LOOPS User Interface

A key feature of LOOPS is its smooth integration with the Venue Medley environment. Many of the tools within Medley have been extended to provide the necessary functionality for manipulating objects. Among these tools are the following:

- SEdit
- The inspector
- Masterscope
- The File Manager
- The Library Module Grapher

This section describes how LOOPS interfaces with each of these Medley tools.

Another aspect of LOOPS is that objects have a name space that is separate from the Lisp name space. LOOPS names are Interlisp symbols. Applying a LOOPS function **$** to a Medley symbol extracts a pointer to a LOOPS object. Objects can also be pointed to as a Lisp value.

1.6.1 SEdit

Class structures are Lisp data types. To change a class structure, LOOPS creates a list structure source for the class definition. This list can then be edited easily by SEdit. Upon exiting SEdit, the list structure is converted back to a data type. This process of converting to and from a list is hidden from view.

1.6.2 Inspector

Inspector macros have been defined within LOOPS that allow you to view the necessary class and instance data while hiding implementation details. Inspectors opened on classes or instances also provide functionality for changing the way one views an object. As an example, you can inspect a class and see or not see information inherited from its superclasses.

1.6.3 Masterscope

The Library Module Masterscope has been extended to a LOOPS Library Module so that message sending and the use of instance and class variables are understood. The functionality of CHECK has been extended to allow consistency checking of LOOPS methods.

1.6.4 File Manager

Additional File Manager commands have been added to allow you to save classes, instances, and methods on files.

1.6.5 Grapher Module

An important part of the LOOPS interface is its ability to show relationships between objects and to enable the programmer to easily manipulate those objects. Browsers of various kinds are in the system to allow you to understand the relationships between classes and how those classes are related to files. The browsers are built upon the Library module Grapher. You can easily extend the built-in browsers to create views onto any object relationship. An example of this is a decision tree where each node was an object representing a particular state of a system.

[This page intentionally left blank]