New Directions in Object Oriented Programming: Concepts, Issues and Alternatives

Mark Paul Keefe

1986

Cognitive Studies Research Papers

WtKNfclilfc-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

Q ~] (Q

New Directions in Object Oriented Programming: Concepts, Issues, and Alternatives

Mark Paul Keefe

Cognitive Studies Programme
The University of Sussex

Support for this project was provided by Systems Designers, Plc. in the form of a research fellowship to investigate potential future enhancements in the form of object oriented programming capabilities to the POPLOG Artificial Intelligence Development System in development at the Cognitive Studies Programme, University of Sussex.

October 21, 1986

Contents

Chapter 1 — Introduction

One of the most compelling results to emerge from over 25 years of research in the area of Artificial Intelligence is that intelligent systems require very large amounts of domain-specific knowledge. Contrary to early assumptions that the discovery of a small number of powerful underlying principles of problem solving behavior would result in flexible, general techniques that could cope with a wide range of situations, it has been found that the most critical factor influencing the performance of intelligent systems is the amount of knowledge present.

Besides tending to come in large and unwieldy chunks, knowledge is both hard to characterize accurately and is constantly changing [Rich 83]. Thus, a correlate to the finding that intelligent systems need to be knowledge intensive is that the early techniques for representing and manipulating knowledge, which included the use of first-order predicate calculus and "semantic" networks, were lacking either in terms of expressive power, ease of use, uniformity or efficiency.

The object oriented paradigm, which grew out of semantic networks and frame-based languages, addresses several of the above criteria through its ability to capture complex structural information in the form of objects and their associated attributes; generalizations, taxonomies, and exceptions; the distinction between generic concepts and individual instances of such concepts; and the association of general and specific behaviors with individual instances.

Object oriented programming has become a powerful tool for the representation of domain-specific knowledge. Nonetheless, it not well suited for representing all kinds of knowledge; for example, it provides only limited capabilities for expressing universal and existential quantification, and none at all for expressing most negative and disjunctive facts [IntelliCorp 84]. In addition, it does not yet have a well defined semantics, as does predicate calculus. In terms of efficiency, object oriented programming languages are pushing very hard at the capabilities of modern computer architectures. However, used in conjunction with other proven AI techniques such as rule-based programming, it has shown itself to be a necessary and useful addition to the AI developer's bag of tricks.

There are a growing number of object oriented programming systems and languages in academic and commercial use, and new languages are still being proposed. Having been around for a relatively small amount of time, object oriented programming concepts are still being defined and developed, and consideration of the issues of expressiveness and efficiency have in recent years led to a number of innovations. A growing number of people are becoming interested in developing some form of standardization, but as yet a date for such a standard is unforeseeable due to disagreement regarding the semantics of basic concepts and lack of understanding of the kinds of problems to which the methodology may be usefully applied. A framework is needed that is general enough to support the variety of different approaches to constructing object oriented programming systems and can capture and distinguish the fundamental underlying assumptions and features embodied in the object oriented programming model. It is to this need that this paper addresses itself.

The following chapters work toward the establishment of such a framework by exploring some of the basic concepts of object oriented programming, following the evolution of some of the major systems and languages in existence, and describing the most important innovations recently proposed.

Chapter 2 presents a glossary of terms that appear repeatedly throughout this report. An attempt is made to keep the definitions as general as possible in order to accommodate the incorporation of the ideas presented in subsequent chapters.

Chapter 3 is intended to illuminate some of the distinctions between the many object-oriented programming systems and languages that have been proposed and implemented in the past decade or so. The following is one useful way of categorizing these systems:

- Completely uniform programming frameworks (Smalltalk)
- Extensions of various Lisp dialects (Eg. Zetalisp-Flavors. Interlisp-Loops)
- Systems built as add-on packages to other languages, notably Lisp (Eg. KEE, ART, Knowledge Craft)

Most of the work in implementing object oriented programming concepts has appeared in Smalltalk and Lisp, to which are devoted entire sections. Other work covered in this chapter includes implementations based on Prolog. C, and Pascal.

Chapter 4 examines a number of recent innovations which may have a significant impact on the power and efficiency of object oriented programming systems and languages in which they are incorporated. Each innovation is distinguished as to whether it essentially represents a wide or narrow interpretation of one or more basic concepts of object oriented programming, and is examined in terms of the advantages and disadvantages it has to offer.

In Chapter 5, basic concepts and important innovations are drawn together into a proposed framework that attempts to meet the needs of disparate and occasionally conflicting features, is powerful enough to support many of the existing approaches to implementing object oriented programming systems, and is open-ended enough to support future innovations. A layered approach is utilized which distinguishes fundamental or universal assumptions and capabilites from those which are tied to particular systems, approaches and objectives.

Chapter 6 closes by providing a few case studies of major object oriented languages. A uniform comparison of strengths and weaknesses along various dimensions is not possible at this time due to the lack of documentation and reported applications experience available, however, it is considered that the information presented in this chapter is of potential use to those who are considering the inclusion of object oriented programming capabilities within their own development systems.

Chapter 2 — Basic concepts

In this chapter, a glossary of terms referring to some of the most common notions of object oriented programming systems is provided. It is worth noting that not everybody is in agreement on the basic concepts of object oriented programming or their semantics. I have therefore tried to keep the glossary as general, and as minimal, as seems compatible with the goal of providing a useful explanation.

Data Encapsulation

The packaging of structure and functionality into a conceptual unit, or *object*, is a means of supporting the principle of ***data encapsulation***, which states that users of objects should not be granted privileged access to the private parts of such objects. The terms, *class*, *prototype*, *module*, *package*, *unit*, *frame*, *schema* and *node* are used in various languages to refer to general descriptions of such conceptual units. The term *instance* is most often used to refer to an actual entity whose structure conforms to the general description and whose attributes have specific values. For example, the term, "elephant", describes a class of animals having a *color* attibute, of which "Clyde" is an instance whose color is pink.

Object

The fundamental unit in object oriented programming systems is the ***object***. It is a data structure which contains unique memory locations for a set of attributes (instance variables) that describe its current state, and is associated with procedural code (methods) that implement operations defined on similar objects. Access to the object's attributes is normally mediated through an interface. The interface causes a request for an object to perform one of its operations to be interpreted as a request for the object to select and invoke a method to carry out the operation.

Class

Objects are most often organized into groupings of similar objects. Such a grouping is referred to as a ***class***. The class of an object determines the object's structure as well as defining the interface to the object by determining the set of methods the object may invoke to carry out operations. Two objects are said to be similar if they belong to the same class, that is, if they have the same structure. The structure of an object can be considered to mean the way in which the object's attributes and behavior is implemented. An object that is a member of a given class is called an *instance* of that class. Information local to each instance is represented in the same manner in all instances of the class.

Class Inheritance

Classes themselves are often organized into a hierarchy, or more generally a lattice. The directed links in the lattice indicate a *specialization* relationship between neighboring class nodes in the lattice. The specialization relation is transitive. A class that is a specialization of another class ***inherits*** structural information from the more general class, or "superclass", as well as the interface of the superclass. This means that instances of the inheriting class, in addition to their own attributes and methods, have all of the attributes and methods defined for instances of the superclass. Inherited methods and attributes are indistinguishable from those defined on the inheriting class.

Instance

The term ***instance*** usually refers implictly to the relationship between an object and its class (cf, Data Encapsulation). In several languages, including Smalltalk and Loops, all objects (including classes) are implemented as instances of some class. When discussing languages in which classes are implemented in the same way as instances (ie, as objects), the term "instance" is often used to refer to objects that are not classes.

Instance variable

An object has storage allocated to hold a set of attributes. These are usually referred to as ***instance variables*** or ***slots***. Each instance variable associated with an object is defined in a class that the object is an instance of (it may also have been defined in an inherited class), and has storage allocated in the object. Thus, if a class A, or one of the classes it inherits from, defines an instance variable x, then all instances of A and of classes inheriting from A will have unique memory locations for storing their own value for the variable x.

State

A particular combination of values for the attributes of an object at some point in time is referred to as the object's ***state***.

Method

The code that implements a particular operation defined on a class of objects is referred to as a ***method***.[1] The method may have been directly defined on that class of objects, or it may have been defined on an ancestor of the class in the inheritance lattice; in either case, it is said to be defined for that class of objects.[2] When an operation on a particular object is requested, it is necessary to locate an appropriate method with which to perform the operation. The method can sometimes be determined statically, but in general a dynamic lookup is necessary (namely, when the object's class cannot be determined at compile-time).[3] The way in which an object finds an appropriate method with which to respond to a particular request is generally determined by the position of the object's class in the inheritance lattice. In particular, the object's class is usually examined to see if it is directly associated with an appropriate method. If it is, that method is used; if not, a search from the object's class through its inherited classes is conducted until an appropriate method is found.[4]

Interface

The set of operations defined on a class of objects is called the ***interface*** to objects of that class. The interface provides a way of mediating between "outside" requests for an object to perform one of its operations, and the selection of the code with which to perform it.[5]

Generic Operation

An operation that may be requested in the same (syntactic) manner for different classes of object in order to acheive some "standard" behavior (ie, expected, or grossly similar) is referred to as a ***generic***

---

[1] Note that the terms "method" and "operation" are not synonymous. I will use the term "operation" when referring to some abstract action or sequence of actions (eg, object X is requested to perform its print-self operation), and the term "method" when referring to the actual code used to perform such an operation.

[2] In Common Objects it is possible for one class to inherit from another without inheriting all of the methods defined on the inherited class.

[3] This kind of dynamic binding scheme is critical for the support of modularly designed programs, that is, programs whose parts may later be reused, without modification, in constructing new applications.

[4] Dialects vary on what happens if none is found; in Smalltalk and many other dialects it is simply an error, however, other dialects may make it possible to cause some actions to occur before allowing control to be returned to the caller (eg, through the use of wrappers in Flavors).

[5] Protection of an object's private attributes can be guaranteed by forcing outside access to the attributes of the object to be performed by going through the interface (ie, by requesting an access operation). The object's structure can be safely modified, by redefining attributes and methods in the object's class or superclasses, without changing the interface.

operation. Generic operations differ from ordinary operations in that they hare a distibuted definition rather than a single definition; that is, there is generally more than one method that implements the operation. The definition used to perform the operation varies from call to call, whereas with ordinary operations, the same definition is always used.

Computational Metaphor

The way In which objects art requested to invoke methods for performing operations in a given language is sometimes referred to as the *computational mmtaphor* of the language. Message passing (see below) is one example of a computational metaphor. Procedure call, in which one of the formal arguments is treated a* special (le, given responsibility for determining a method to implement the operation) is another. In some languages (eg, Smalltalk) message passing is the only computational metaphor. In others, all computations are performed by procedure calls, with "object" arguments treated specially.* Some other languages prefer to mix the two metaphors, with message passing used to request generic operations on objects (implying dynamic method lookup, cf Method) and procedure call used to request non-generic operations on any kind of data.

Protocol

Object interfaces can be used to design a standard *protocol* so as to treat instances of different classes in a uniform manner. This can be achieved by specifying a generic operation such that the operation's performance is requested in the same manner regardless of the class of object to which it is to be applied.

Message passing

In many but not all languages, operations on objects are requested through the use of *messag* passing.* The set of messages a class of objects is able to respond to can be regarded as the interface to instances of the class. A "message" is composed of a symbol identifying an operation whose performance b requested (the message "selector"), an object responsible for performing the operation (the message "receiver") and any formal arguments whose values are necessary. Thus, the sender must know the name of the message selector and the number of arguments required for the operation. An instance of a particular class may successfully respond to a message for which an applicable method has been defined on the class (the instance may be said to "handle" the message) by interpreting the selector and invoking the appropriate method with the arguments supplied.[7]

Object oriented programming supports the principle of data encapsulation by providing a means of packaging the attributes associated with a conceptual entity, along with specific methods to access the attributes, into a physical data structure. Direct access to an object's attributes is possible from methods defined on the object's class. Outside access requests can be channelled through a well-defined interface which causes such requests to be interpreted as requests for the object to select and invoke an appropriate method.

---

[6] For example, C++ introduces the notion of *virtual* functions, which are called in the same manner as ordinary C functions, but can cause a dynamic dispatch to the actual code used to implement the function.

[7] The set of messages which a class *documents* as being defined for its instances (as well as instances of subclasses of the class) constitute the message interface to instances of that class; it is the channel through which an instance may be asked to carry out one of its operations.

Object oriented programming enables programs to be defined in a modular style by allowing the specification of generic (virtual) operations whose invocation is independent of the class of object upon which they are to be performed. In order for code to be reusable, requests to perform such operations must not need modification when the implementation of objects has been modified. This implies that the caller should not require knowledge of the object's representation, nor of how the operation is actually performed. The caller should only need to know what operation it wants performed and how to request it. Performing the same operation on dissimilar objects should be possible without the caller needing to know that the objects are dissimilar.

Object oriented programming is partly a matter of facilites provided by a language and partly a matter of design methodology. Different object oriented programming languages take different views on how strongly the twin principles of data encapsulation and program modularity should be upheld. For example, in some languages, it is very difficult to write code that violates these principles. Other languages merely make it possible to write programs that adhere to them, but also provide ways of getting around such restrictions.

Chapter 3 — History

It is useful to have some idea of how various concepts, languages, and systems evolved in order to compare different approaches, to assess the staying power of new ideas.[1] and so forth. In this chapter, brief summaries are given of the evolution of some of the major object oriented programming systems and languages.

1. Smalltalk

The earliest versions of Smalltalk [Kay 72] were influenced both by earlier languages (especially Simula-€7. [Dahl 66. Dahl 70]) that provided constructs to support data encapsulation, as well as by theoretical work, mainly at MIT. that advocated the use of message passing as a control structure [Hewitt 76].

Ideas culled from the research of Bartlett. Quillian. Minsky. Schank. Norman, Rumelhart, Rosch, Bobrow. Winograd. and others — whose work on structured representations of memory, perception, inference, and general concepts went under such names as *Schemata, Frames, Semantic* or *Associative Networks. Scripts,* and *Prototypes* — contributed greatly to the concepts of

- attributes, features, or properties of an object;
- attachment (in some sense) of procedural behavior to an object;
- organization of objects into networks representing relationships between objects through which information can be shared.

Smalltalk has been completely redesigned twice [Ingalls 81, Goldberg 83], with the following major evolutionary changes:

(1) The language itself has become more and more "object-oriented" to the extent that everything in the Smalltalk-80 system is an object (from numbers, characters, and strings, to compilers, editors, and file systems, to classes and contexts) and all processing is effected by message passing [cf. Ingalls 81].

(2) More and more attention has been paid to the Smalltalk user interface, especially as machine address space, processor power, and bit-mapped graphics hardware and software have improved.

2. Extensions to Lisp

Lisp is a particularly good language for extending, partly due to its simple but general syntactic structure, partly because there is no difference between built-in and user-defined features, and partly because programs can be manipulated as data. For example, a simple Lisp interpreter can be written in Lisp relatively straightfowardly; this can then be extended to interpret larger and larger supersets of the Lisp language. There are various ways of providing support for object oriented

[1] Flavors, Smalltalk, and Loops are the most widely available object programming systems, having

programming in Lisp, differing mainly in the level of integration with the language.

Extensions to Lisp implementing "frame systems" [Bobrow 77], "Scripts" "Schemata" etc, began to appear around 1975. These systems are similar in that they all use the notion of "structured objects" as the basis of a representational formalism for capturing various kinds of knowledge [Rich 83].

The CLU language, developed at MIT around 1979, appears to have been the first of the Lisp dialects to offer object oriented programming facilities. Its major innovation was in drawing a strong distinction between *mutable* and *immutable* objects, the latter being those whose internal state is never modified (eg, integers and booleans). The Units system, devised by Stefik [Stefik 79], emerged at about the same time as CLU, but emphasized the use of knowledge-engineering tools as opposed to general data abstraction facilities. Units was the precursor to Loops, KEE, and Strobe.

The Zetalisp (Symbolics) Flavor System was released in 1981 [Symbolics, Weinreb 81, Weinreb 85], providing deep level system support and a wide range of features. Flavors appears to be the first commercially available system to provide multiple inheritance. It allows the user to specify complex patterns of combination for methods inherited from more than one superclass.

Bobrow and Stefik at Xerox PARC, developed Loops [Bobrow 83] which provides multiple inheritance, metaclasses, active values, and composite objects. Since then, countless object extensions to Lisp dialects as well as object languages based on Lisp have emerged, including Strobe, NIL, VLISP, Mering, LeLisp, bVLISP, and XLISP.

Between 1983 and 1985, a handful of high-level knowledge engineering tools began to appear from the commercial AI startups, notably KEE [IntelliCorp, KEE 84], ART [Inference Corporation, Williams 85 Clayton 85], Knowledge Craft [Carnegie Group, Pepper 86] and Omega [Delphi SpA, Attardi 86]. These and others (ABE?, Units, Strobe) have grown out of work on the provision of high-level tools with which to build expert systems shells. Research in cognitive engineering and knowledge-based systems have played important, complementary roles, particularly in determining how the interface between the user and the system ought to appear and behave.

With the emergence of Common Lisp has come a more recent spate of proposed "standard" object extensions, including ObjectLISP [LMI, Drescher 85], CommonLoops [Xerox PARC, Bobrow 85], and Common Objects [Hewlett-Packard Labs, Snyder 85]. Both Common Loops and Common Objects rely on extending the lisp `defstruct` form (used to create Pascal-like, aggregate record types) for creating user-defined types. ObjectLISP uses nested Lisp `closures` to implement lexical environments in which variables and functions can have distinct bindings.[2] Symbolics are also revising their Flavors System [Keene 85], although not yet proposing that it be adopted as a standard. They, like most others, have adopted a "wait-and-see" attitude. A Common Lisp Object Oriented Programming Subcommittee has been formed in "recognition of widespread recognition of the opportunity and timeliness of integrating objects into Common Lisp".

[2] Performing operations on ObjectLISP objects is acheived not so much by message passing as by restoring an object's lexical environment in order to access its function bindings (methods).

Flavors, Common Loops, and Common Objects are examined more closely in the case studies section.

3. Extensions to Prolog

There have been a number of proposals for extending Prolog with OOP facilities, including ESP [Chikayama 84], Concurrent Prolog [Shapiro 83], and some others that don't even have a name [Zaniolo 84]. These have not been nearly as successful in terms of popularity, and have been somewhat more controversial than Lisp extensions, perhaps in part because it is not clear whether the object oriented programming paradigm is fundamentally compatible with the logic programming paradigm, for example, with regard to creating new objects, changing the values of variables, etc. Some very interesting work has been done, however, in an attempt to address these problems, by those working on Concurrent Prolog [Shapiro 84].

4. Extensions to the C language

After some initial skepticism by those who were unwilling or unable to accept the huge resource requirements of Smalltalk-80, the C community has been quick to jump on the bandwagon. Work on "C with classes" at Bell Laboratories dates back to 1982, with the main influence reported to be the Simula-67 class concept, and has evolved into what is now called C++ [Stroustrup 86].

> The intent was to create data abstraction facilities which are both expressive enough to be of significant help in structuring large systems, and at the same time useful in areas where C's terseness and ability to express low level detail are great assets. Consequently, while C classes provide general and flexible structuring mechanisms, great care has been taken to ensure that their use does not cause run time or storage overhead which could have been avoided in old C. [Stroustrup 84a]

C++ is discussed in greater detail in the case studies section.

A second approach to supporting object oriented programming in C appears in Objective-C, developed by Productivity Products International [PPI 84]. Objective-C also claims to be a superset of C. Objective-C extends the C syntax to allow a "message expression" to appear wherever C functions may appear. Objective-C message expressions have a format very similar to simple Smalltalk message expressions except that they are surrounded by square brackets (allowing for complex, nested expressions) and cannot be binary expressions. Objective-C is also dicussed in the case studies section.

Apple are also investigating the addition of object oriented extensions to the C programming language. They believe Objective-C and C++ are not simple enough to gain widespread acceptance amongst C programmers. Norm Meyrowitz at Brown University [Meyrowitz 85] has designed a C extension that follows Apple's Pascal extension, discussed below. Modula-2 is claimed to be easily extendible in a similar manner.

5. Other language extensions

Object Pascal is Apple's object oriented extension to Pascal [Tesler 85]. Object Pascal is a descendant of Apple's original extension to Pascal, known as Clascal.

Apple was unhappy with the Clascal syntax, having found it difficult to learn and use. Out of consultations with Niklaus Wirth, the inventor of Pascal, came the design for Object Pascal. Simplicity was a main objective, another being that the design should not be clouded by implementation issues. Apple's experience with Clascal had indicated that certain concepts, such as metaclasses and class methods, were "of limited usefulness" and "tended to confuse new users". An example of Object Pascal syntax is provided in appendix A.

Table 1 is an attempt to summarize some of the information presented in this section.

System	Derivation	Origin	First Appearance
Simula-67	†	K. Nygaard	1966
Smalltalk	†	Xerox PARC	1972
Units	*Lisp*	M. Stefik	1979
CLU		MIT CS	
Flavors	*Zetalisp*	Symbolics	1981
Act	†	MIT AI	
PSL	*Lisp*	University of Utah	1982
T		Yale University	
GLISP		G. Novak	
Loops	*Interlisp*	Xerox PARC	1983
Strobe		Schlumberger-Doll Research	
NIL	*Lisp*	MIT CS	
VLISP		P. Cointe	
Mering		J. Ferber	
LeLisp		J. Chailloux	
KEE	*Zetalisp*	IntelliCorp	1984
ART		Inference	
ESP	*Prolog*	ICOT	
C++	*C*	Bell Labs	
Objective-C		Productivity Products	
ObjectLISP	*Common Lisp*	LMI	1985
Common Loops		Xerox PARC	
Common Objects		Hewlett-Packard Labs	
CRL		Carnegie Group	
XLISP		D. Betz	
Object Pascal	*Pascal*	Apple	
BLOBS	*Pop11*	Cambridge Consultants	
Omega	*Common Lisp*	Delphi SpA	1986
POOPS	*Prolog*	University of Aberdeen	

† *Original language*

Table 1: Timescale and origin of object oriented programming systems and language extensions.

The derivation is the original language the new system is built on (if any). The origin is either the place where the language was developed, or the author of the manual or report describing the language, if known. The year of appearance represents the date a manual or report was first published.

Chapter 4 — Recent trends

Recent trends in the development of object oriented programming systems fall broadly into two categories: those that attempt to generalize and those that attempt to narrow the interpretation certain fundamental OOP concepts. This chapter highlights some of the major innovations on both sides to emerge in the past couple of years and examines some of the issues underlying them.

1. Generalizations

The most central notions of object oriented programming, as described in the glossary presented in chapter 2, can be extended in several ways. Inheritance of structure and behavior from a single superclass can be generalized to multiple inheritance, provided that some conflict resolution scheme is devised to deal with name clashes. This is typically achieved by defining a class precedence list of the superclasses associated with each class. Inheritance may also be generalized by allowing structure and behavior to be shared through arbitrary relationships between classes, for example, through *partitive* relationships. The general notion of knowledge sharing may also be realized by allowing an object performing some operation to create new objects and to delegate parts of the operation to them, or to copy itself and forward the original request for the operation to be performed to its replacement.

Instance variables can be extended to have arbitrary *extra* information associated with them, other than merely values — an idea stemming from the notion of property lists in Lisp. One particular use of such information is the incorporation of *daemons* which can be triggered upon instance variable access. These can themselves be generalized to be arbitrarily nested, such that the act of triggering a daemon can itself trigger another daemon, and so on.

Method discrimination based on the message name (selector) and the class of the receiving object can be generalized by allowing method selection to be influenced by the classes, or types, of any number of message arguments. The semantics of message passing can be viewed as a special kind of procedure call. These emerging lines of research are discussed in the following sections.

1.1. Inheritance

Inheritance is usually tied to the notion of a class *specialization hierarchy*, with more specific classes in the hierarchy inheriting structural information from more general classes. The most obvious generalization of this scheme is to allow ***multiple inheritance***, in which a class is allowed to inherit structural information from more than one superclass, with what was formerly an inheritance hierarchy now becoming an *inheritance lattice*.

Most of the major object oriented languages (Common Loops, Loops, Flavors, ObjectLISP, Common Objects) already provide multiple inheritance. It is rumored that Smalltalk will provide multiple inheritance, but there still appears to be some amount of disagreement amongst some people regarding the general power and utility of this approach to the sharing of information. Other notable languages lacking

multiple inheritance are C++[1], Objective-C, and XLISP.

Notice that it is possible to shift the focus of attention from talking about *how* an inheritance mechanism may be implemented (eg. through a class specialization scheme) to simply talking about *where* structure and behavior are inherited from, that is how this information is shared amongst various classes of objects. This suggests that specialization is but one way to factor and organize shared information. Suppose you point to a green automobile and ask one to guess the color of the door on the hidden side. A good guess would be "green", because the door bears a partitive relationship to the automobile, and the automobile is green. In this example, the inference process is aided by an arbitrary relationship between an object about which some particular information is desired and an object about which some relevant information is known.

The point is that specialization is only one kind of potentially useful relationship amongst (classes of) objects. While it provides a natural organizational mapping onto a wide variety of knowledge-sharing situations, there are a potentially large number of other kinds of relationships that may prove superior on a host of occasions. One of the most frequent of these occurs when the need arises to access default information, as the automobile example illustrates. Strobe [Smith 83], Mering [Ferber 84], UNITS, and ART are examples of languages that make use of this approach. Loops, KEE, and ESP provide some support for partitive relationships through the ability to define *composite objects* through the use of ***templates***.

Specialization, as a way of organizing classes so that they may inherit structure and protocol, has been seen to be a special case of the idea of allowing objects to respond to certain messages while passing along those for which a response is not locally defined to other kinds of object. The more general notion of forwarding messages to other objects is implemented in the Actor languages [Lieberman, 1986] through the use of ***delegation***. In the Actor languages, an object (actor) is not confined to the "hard-wired" paths of a class inheritance lattice. Instead, it may delegate all or parts of a message to arbitrary other actors. The decision of who to delegate the message to can be made at run time, rather than at actor creation time or compile time. Programming cliches are provided that "emulate the usual forms of inheritance from conventional object languages [Stefik 86, AI Magazine]." At present, it is difficult to try to assess just how much the delegation approach might suffer by not allowing as much advantage to be taken of the organizational benefits that doubtless result from the use of specialization networks.

It should be mentioned that specialization raises several implications that can be difficult to disentangle:

(1) It provides a nice way of implementing the inheritance of structure, protocol, and default information.

(2) It allows "natural" boundaries to be imposed on the organization of a program, around which abstract problems are more easily decomposed and solutions composed. Programs are consequently easier to read, grasp, and therefore maintain and modify.

(3) It provides a means of expressing meta-relationships, that is not only is low-level knowledge — say, how to locate a default value — expressed, but at a

[1] C++ and Smalltalk-80 have raised enough discussion and controversy amongst both applications and systems developers to warrant having their own USENET newsgroups — net.lang.c++ and net.lang.st80,

much higher level, a programmer can more readily discern how different parts of a program relate to each other.

The second and third implication may be seen as different ways of expressing the same idea, and even the first implication is. in some sense, equivalent to the other two. The three differ in their *conceptual* point of view, and all three viewpoints need to be addressed in attempting to decide how an inheritance mechanism ought to be realized.

1.2. Instance **variables and** methods

Instance variables hold values that represent the internal state of an object. In completely uniform languages, such as Smalltalk, the value is itself an object. In other kinds of object languages, instance variables may take on any of the type of values a variable is allowed to take on in the language, including procedures in some languages.

One kind of generalization then is to implement methods as instance variables, provided the syntax is transparent enough to hide the distinction between instance variable access and method invocation. This is the approach taken in KEE, Strobe, and the UNITS package, amongst others. In these languages, message sending is extended to include the notion of sending a message to a slot. The use of Lisp as a base language makes such an approach particularly easy to implement becaxxse of the uniform manner in which variables (symbols) and symbolic expressions (eg. message expressions) are evaluated to produce results.

Loops, like KEE and Strobe is a direct descendant of the UNITS package. These languages are intended to be used primarily in "knowledge engineering" applications, and as such, one of their design objectives is to provide a mechanism allowing supplementary information to be associated with a slot (instance or class variable) that describes the kind of value that can-be stored there. In the Loops system, these extra bits of information describing a slot are referred to as *propetty annotations.*

Examples of information which may naturally be stored with a slot are documentation, history of previous values, value constraints, certainty information and (in Units. KEE and Strobe) the datatype of the value. The datatype can be used to provide auxiliary methods for printing, matching, application, etc. which are activated when a procedure attached to the slot itself is not provided.[2] as well as to restrict the kinds of values that can be stored in the slot.

In Loops, instance variables may be annotated with daemons, known as *active values,* that trigger procedure calls upon instance variable access [Stefik 86. IEEE Software]. The annotations may be annotated recursively. For example, it is possible to implement an active value that is accessed by calling a "get" function. The value the get function gets may itself be implemented as an active value, thus triggering another get function, and so on. The same is true for "put" functions.

---

respectively .

[2] Of course, In a uniform framework such annotations would be unnecessary because the slot value would itself be an object with appropriate methods for these operations.

The POPLOG package LIB FLAVOURS allows daemons to be triggered before and/or after instance variable access, by allowing `before` and `after` methods to be defined on instance variables as well as on primary methods. The daemon methods are combined and run in the usual Flavors-style `:daemon` method combination order, so that they may be invoked before and/or after instance variable access.

1.3. Method discrimination

Method discrimination refers to the process of selecting an appropriate method with which to respond to some particular message. *Classical methods* are those in which method discrimination is based on the method selector and the class of the receiving object. Assuming for the moment that message sending has the same syntax as procedure calls and the first argument is interpreted as the receiver, then classical method discrimination means discriminating on only the method selector and the first argument.

Common Loops generalizes the notion of method discrimination by allowing method selection based on any number of arguments, including all of them or none of them, hence, the notion of ***multi-methods***. The discrimination is controlled by specifying the type (class) of arguments in the method definition. Object space is seen as an extension of the Common Lisp type space, so, for method discrimination purposes, the message receiver is treated as just another argument. For a given set of arguments to a message, there can be a continuum of applicable methods, from simple functions to those whose arguments are fully specified. The most specific applicable method is chosen to handle the message; only when no other method matches is a method with no type specifications on its arguments invoked.

From the caller's point of view, it makes no difference whether it is a method or a function that is being invoked.[3] The advantages are that

- The efficiency of compile-time type-checking is made possible,
- message arguments are treated more uniformly, and
- methods can be more cleanly decomposed.[4]

Suggested extensions to the Common Loops kernel include allowing type specifications to permit range constraints, disjunctive cases, and so on.

A problem with making use of the Common Lisp type space is that its types are not completely hierarchical, raising the possibility of ambiguities in type specifications. That is, a message argument might be of a type that is a descendant of more than one ancestral type; for instance, the message

```
( foo  obj  nil )
```

invokes a method whose second argument is a subtype of every type in Common Lisp. Thus, two or more methods named foo, which agree on the type specification of the first argument, obj, but disagree on the specification of the second argument, would be equally applicable for purposes of method discrimination, due to the lack of precedence definitions in the Common Lisp type space. Common Loops tackles

[3] Thus, less knowledge is required of how a particular operation is implemented.

[4] This is because it is no longer necessary to use potentially large, arbitrarily nested case selectors to deal with the message arguments.

this problem by

> ...making precedences explicit where possible, and by disallowing the use of particular types in type-specifications of methods where no precedence could be reasonable imposed.

For example, Common Loops assigns `list` and `sequence` to have precedence over `symbol`. Thus a foo method whose second argument is specified to be of type `list` is more applicable than one whose second argument must be of type `symbol`.

Common Lisp also allows `and`, `or`, `mod`, `null`, and `member` as type specifiers. While Bobrow. et al, disallow these as type specifiers in the core language, they suggest allowing some of them as a possible extension to the Common Loops kernel. Common Lisp also allows types to be specified by defining a relationship that the type `satisfies`. Common Loops disallows `satisfies` as a type specifier because "...it is not possible to determine a precedence relation between two `Satisfies` specifiers [Bobrow 85]'*. Other Common Lisp type specifiers are disallowed because their implementation is not defined in the Common Lisp specification; these include: `package`, `pathname`, `stream`, `random-state`, `readtable`. and `compiled-function`.

The Flavors system provides a more limited capability to discriminate on other message arguments, known as the `:case` type of method combination. It allows subsidiary message names (keyword selectors) to cause dispatching in the same way that the main message selector (method name) caused the primary dispatch. This is a less general solution than that advocated by Common Loops; in particular, in does not appear to conform to the Flavors system principle that adding new classes of object should only require adding new code, not modifying existing code [Weinreb 85].

2. Restrictions

Narrower interpretations of the basic concepts of object oriented programming systems are primarily concerned with the operational semantics of inheritance schemes. To restate what is normally meant by inheritance, a class may inherit both instance variables and methods from one or more other classes. Each instance of the inheriting class contains storage for the union of

(a) all instance variables defined in the class itself, and

(b) all instance variables defined in the inherited classes and their superclasses.

The inheriting class cannot distinguish between instance variables defined locally and those it has inherited. For all practical purposes, an inherited instance variable *belongs* to the class.[5]

An instance variable name that appears more than once in the inheritance lattice is inherited at most once by the inheriting class. In other words, every instance variable has a unique value. For an instance variable defined more than once by the ancestors of the the inheriting class, the ancestor it will be inherited from is predetermined (at compile-time) by a traversal algorithm for the lattice.

[5] From an implementation viewpoint, this implies that the class is responsible for managing the

Local storage of inherited methods is not strictly necessary, since inherited methods can be accessed by traversing links in the inheritance lattice. Each instance of the inheriting class can invoke those methods defined by the class, as well as those (implicitly) inherited from superclasses whose definitions have not been overridden. Again, each method thus associated with the inheriting class has a unique definition which is determined by the ordering of the traversal in a manner analogous to that for instance variables.

2.1. Inheritance of instance variables and methods

Having stated the above, we are now able to examine an alternative view of inheritance, advocated by Snyder and implemented in the design of HP Common Objects [Snyder 85]. Common Objects defines inheritance of instance variables as follows (for "type" may be substituted "class"):

> ...if the type child inherits from a type parent, then each object of type child automatically includes as part of its representation the...instance variables of an object of type parent. These inherited instance variables are distinct from any...defined by the type child and any...inherited by child from other types, regardless of the names of the instance variables. Thus, for example, if the type child inherits from types parent1 and parent2, and the type parent2 also inherits from parent1, then objects of type child will include *two* distinct sets of the instance variables of type parent1.

This definition of multiple inheritance is quite different from that of the Flavors system, as well as most other systems, in which there can be only one inherited instance variable for a particular name, irrespective of how many superclasses define that instance variable.

A further difference is with regard to the ability to access inherited instance variables:

> If a method defined on the type child wishes to access an instance variable inherited from the type parent (or an ancestor of parent), it may do so only by invoking methods defined on the type parent (including methods inherited by parent).

In contrast, Smalltalk, Flavors, and many other systems allow direct access of inherited instance variables. As Snyder argues,

> Our approach is motivated by the principles of data abstraction and encapsulation. From the software engineering point of view, *inheriting* a type is no different than *using* a type: if an inheriting type can access the instance variables defined by the inherited type, then the instance variables would have to be part of the external interface of the inherited type.

According to the principles of data abstraction, strictly speaking, an object of type child that inherits an instance variable x, from type parent, must not make any assumptions about the implementation and internal representation of x. This principle is upheld in Common Objects by specifying that the only way for the object to access the value of x is by invoking a method belonging to parent.[6] This also upholds the principle of data encapsulation, namely that users of objects ought not to have privileged access to the internals of those objects.

---

instance variable, ie, describing how to access it, restricting the kinds of values it can hold, etc.

[6] The method invoked need not be inherited by child.

It is important to point out that since inherited instance variables cannot be accessed directly by name, name conflicts are impossible. This property allows a powerful mechanism for supporting multiple, distinct ***perspectives*** on an object. The value of **x** can depend on the object's current context, or the perspective from which it is being viewed.

2.2. Data access

Common Objects also differs from most systems in its treatment of method inheritance by use of a model of indirect method invocation, as explained below. Noting that when a type **child** is to inherit from a type **parent**, it is usually the case that one wants instances of **child** to define the same operations as instances of **parent**,

> ...facilities are provided for automatically defining on type child methods corresponding to the operations defined on type parent, passing in the same parameters, and returning the results returned by the method on parent. In effect, the methods defined on type parent are inherited by type child. This...is semantically more accurate than saying that child shares the inherited method with parent.

Thus, in Common Objects, inheriting from a type (class) does not necessarily mean that all methods belonging to the inherited class now belong to the inheriting class as well. Options provided for specifying which methods are going to be redefined and a set of methods to inherit exclusively can be used to prevent some or all methods of the inherited type from being inherited.

Thus, a method *belongs* to a class only if,

(a) the method is directly defined by the class, or

(b) the method has been explicitly inherited by the class.

By default, an inheriting class automatically inherits all of the methods defined by the inherited class, except for certain automatically generated "universal methods" (to avoid naming conflicts). It is an error to attempt to inherit the same method from more than one type.

A third major difference is that since a determination is made at compile-time which instance variables and methods are to be inherited, subsequent modifications (eg, new methods) to inherited types do not propagate to inheriting types. The motivations for compile-time determination of inherited instance variables and methods include:

> ...simplicity of the definition, improved efficiency, and the ability to perform compile-time type checking for certain errors....Further flexibility could be provided by a programming environment that supports automatic propagation of changes to all inheriting types, at the user's option.

In discussing efficiency considerations, it should be noted that methods for accessing inherited instance variables can be open-coded (declared to be "inline-methods") so that the compiled code may actually directly access the instance variables, at no extra cost. Optimizing indirect method invocation is left up to the implementor, with the following footnote:

> It is intended that these methods on type child be implemented very efficiently. Some implementations may choose to define a separate function for the method on

type child, but do it in such a way that the additional overhead resulting from the indirection is very small. Other implementations may choose to use the same underlying function to implement the method on type child and the method on type parent, so that performing an operation on an object of type child is no more expensive than...on an object of type parent; in this case, more work must be performed by the implementation to present the illusion that each type has its own method [Snyder 85, pp. 17-19].

Chapter 5 — A system framework

In this chapter. I propose a general framework for designing an object oriented programming system within the POPLOG AI development environment. A layered approach is examined in which various features appearing in existing object oriented programming systems are assigned to different levels in the overall architecture (see Fig. 1).

Section 1 identifies three key features of object oriented programming systems: data hiding and abstraction, dynamic method lookup, and information sharing through inheritance. These features provide the foundation of the framework and comprise a common base upon which a number of different approaches to object oriented programming can be implemented. As such, they are referred to as base-level features.

Section 2 examines a number of higher level features found in specific object oriented programming systems, focusing on the notions of *doss, instance,* and *subclass* as the most common mechanisms used to define the semantics of inheritance, ie, the sharing of information through specialization and instantiation.

Section 3 discusses the most critical issues relating to implementational efficiency and low-level support: primitive slot access and method lookup.

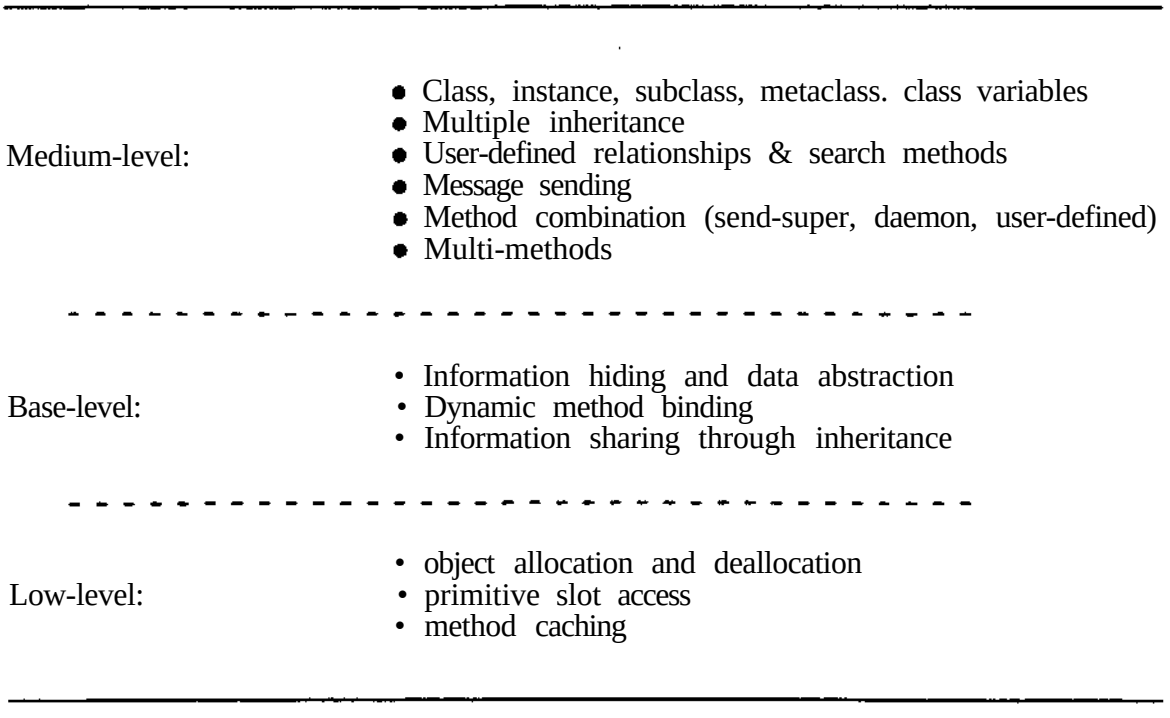

Figure 1: Levels of support for object oriented programming features.

The motivation for an attempt to classify the more familiar as well as the less well-known features of existing systems in terms of levels of support in an overall framework arises not only from the need to identify a common base capable of supporting a wide variety of experimental and proven approaches to object oriented programming but also because such a layered approach provides the additional leverage afforded by allowing the application developer to choose a level to work at that is appropriate to the needs dictated by the application.

1. Base-level features

It is a tricky business trying to determine exactly what is and what is not an object oriented programming system. Nowadays, object oriented programming capabilities are widely seen as being highy desirable, and many different notions of the essential features of object systems have been put forward. As for my own views, I prefer to picture the situation as follows.

There is a set of base-level or "core" features, characteristic to all object oriented languages and programming environments. Any language claiming to be object oriented must provide a set of features which includes the set of base-level features as a proper subset. The features I view as strictly necessary for object-oriented programming are,

- Data Hiding and Abstraction
- Dynamic Binding (Run-time Method Lookup)
- Information Sharing (Inheritance)

The following three sections discuss the properties of these features, focusing on their respective benefits with regard to designing and developing complex programming applications.

1.1. Data Hiding and Abstraction

The most central idea of object oriented programming is that of wrapping up related pieces of information along with the typical operations on that information into a single, uniform data structure. The structure is referred to as an object, the related pieces of information are known as object attributes or properties (also variously called slots, fields, instance variables, etc), and the code segments that implement the operations are called methods. The most distinguishing characteristic of such a scheme is the association between the object and the methods that implement operations on the object. Whenever an operation needs to be performed that involves the object's attributes, the "user" issues a request for the object to perform the operation on itself, rather than requesting a procedure to perform the operation on the object. Such a request causes a method which is associated with that kind of object and appropriate for the requested operation to be selected and invoked. This is in contrast to non-object-oriented systems where the user selects a procedure in advance to perform the operation. The essential difference is one of placing responsibility for choosing the right method with the object rather than with the user. Such systems not only provide a means of delegating this responsibility to the objects involved, but typically encourage or enforce the constraint that only the methods associated with an object may access or modify that object's attributes.

The principal advantages of this model of programming are,

(1) It becomes much easier to design and modify the behavior of program segments using this object, since all of the procedures that depend on the object's representation are localised, both lexically and conceptually (an object's methods need not all appear in one place, but can nonetheless be viewed as belonging to the same lexical environment — one in which all of the object's attributes are bound to a set of values representing its current state).

(2) The object can be ignored when designing or modifying program segments that do not logically involve the object. This leads to a reduced risk of introducing bugs into existing code in the process.

(3) It is possible to ensure the integrity of information privately maintained by the object. This, in turn, leads to an improved ability to verify the correctness of programming segments involoving the object.

(4) It is often more natural for a programmer to think in terms of objects with attributes and associated operations or behaviors. Adopting a more natural representation leads to a reduction in cognitive demands on the programmer, who is then able to divert a larger proportion of his or her limited mental resources to the problem at hand. This leads to more efficient problem solving and design with fewer oversights and mistakes.

(5) Procedures that operate on the object can be more eficient in many cases, since they can be designed to deal with a particular kind of data rather than many different kinds.

(6) The way in which the object represents and maintains its attributes can be modified without affecting program segments involving the object, since only the procedures associated with the object depend on its representation. An object's methods thus act as a layer of insulation between the object's internal information and the rest of the program.

1.2. Dynamic Binding

A second base-level feature necessary for object oriented programming is the ability to support dynamic association of methods to requests. That is. method-lookup needs to be performed at run-time to ensure high reusability of program code segments.

The motivation for dynamic or run-time method lookup is partly necessitated by the first criterion, that of data encapsulation, since users can no longer specify in advance which procedure should be invoked to handle a given operation involving an object. However, a second factor motivating the need for such a feature is that it enables the design and use of generic operations — operations with the same basic behavior which may be performed in different ways depending on the kind of object involved.

The way in which the operation is carried out can be made transparent to the user by providing a syntax for issuing a reqest to an object to perform the operation. The system handles the request by determining the kind of object receiving it and the name of the operation and using these two pieces of information to select and invoke the appropriate method. The syntax allows only for the name of the operation and the object which is to receive the request; the kind of object is not

usually specified. The selection of a method for responding to such a request must be made at run-time, so that more than one kind of object will be able to handle the same request. Some of the benefits that acrue from the ability to dynamically associate methods with requests are,

(1) It enables the design of program segments which do not have to be modified in order to incorporate new kinds of objects associated with the same generic operations as existing objects. In non-object-oriented systems, "generic" operations can only be approximated by using a complex conditional or case switching statement which selects an appropriate procedure or code segment based on the kind of data item involved. Adding a new type of data item very often neccessitates modifications to one or more such switching statements. The use of dynamic binding in method lookup obviates the need for such modifications by relinquishing control of method selection to the object involved in the operation.

(2) Program segments which request objects to perform operations can be easier to understand and modify. Many different methods, associated with different kinds of objects, can share a single symbolic name, thus clarifying the intentions of the programmer and the program's behavior.

(3) When an object dynamically changes from one kind of object into another, it can use a new method associated with the new kind of object. Program segments requesting the object to perform an operation needn't "know" that the object has changed.

1.3. Information Sharing

The final base-level feature of object oriented programming systems is an economical mechanism for sharing information common amongst objects. Such a mechanism eliminates duplication and redundancy of code, allowing both structural information — such as the names of object attributes — and procedural information (methods) to appear textually in one place rather than many. Relationships between individual objects and groups of objects serve as the basis for shared information.

Various kinds of relationships will naturally occur between objects, forming a complex network where links represent the relationships and nodes represent objects or groups of objects. Of these kinds of relationships, a subset are special in that they can be used to indicate paths in the network along which information can be "inherited" from one node by another. Some of the main advantages afforded by using such an inheritance scheme for sharing information are,

(1) Programs can occupy less secondary and primary storage as a result of eliminating much redundant code.

(2) Objects can require fewer memory locations by sharing structural information.

(3) Less duplication of code means less program text segments to be concerned with when modifications become necessary.

(4) Naturally occurring relationships amongst objects are explicitly represented, further clarifying the programmer's intentions.

The three features outlined in the preceding subsections, data hiding and abstraction, dynamic binding of methods, and information sharing through

inheritance, together constitute the most critical and distinguishing characteristics of object oriented programming systems. These are the key features which support and enable all of the benefits enumerated in this chapter, and the common base upon which higher-level features found in different object oriented programming systems and languages can be built.

2* Medium-level features

In this section I discuss a number of medium-level features found in some of the major object oriented programming systems.

In the previous section I outlined what I believe to be the most important concepts necessary to object-oriented programming systems: data abstraction, dynamic binding of methods, and information sharing through inheritance. Underlying all three of these ideas is the notion of the "kind of object". Most object-oriented programming systems provide for objects to be organised into classes. From here on I will[4] use the phrase, "class of object", or simply, "class." when referring to objects that are similar (have the same structure and behavior)[1]. Similar objects use the same method for responding to a particular request and have the same structural attributes.

The notion of *doss* is a great help in specifying how to implement the key concepts mentioned above. In particular, it provides a convenient way of describing the structure of simila objects. Along with the notion of class comes the notion of *subclass,* which provides a means of specifying a class of objects that is just like another class of objects, *except* for a few differences. A third related notion is that of *instance* — an object that belongs to some class is said to be an instance of the class. Instances of a subclass may have some different atttributes in addition to those found in instances of the superclass, or may be able to perform additional operations, or may have slightly extended or altogether different methods for performing the same operations. The notion of class, and the relationship subclass-of provide a basis for one means of implementing an inheritance mechanism, which I will proceed to dicuss in this section.

I have deliberately excluded class, instance and subclass from the set of base-level features in the proposed object oriented programming framework, because I don't wish to preclude from my definition those OOP systems in which various kinds of relations (eg. *part-of. child-of, precedes,* and other arbitrary specified relationships) are allowed as the basis of a particular inheritance strategy. *In my view, one of the main goals of a POPLOG OOP system is that it should provide a common base upon which different kinds of inheritance mechanisms can be built.* With that point in mind, the notions of class, subclass and instance should be kept separate from the notion of information sharing through inheritance. *A second important goal is that the kernel should be small and easy to integrate into the core of POPLOG.* An attempt should be made to keep special features such as active values, and flavors style method combination out of the kernel. *A third major goal is that the kernel can be implemented to run efficiently.* This topic is examined in section 3.

[1] From a taxonomical lUndpoint, it is clear that an object is capable of being an instance of more than one class, though OOP literature usually treats objects as though only the opposite statement is possible (ie, an object is an instance of only one class). Therefore, note that when I refer to "the class of X" I am always referring to the class which was instantiated to create X.

2.1. Representation of Objects

In OOP systems which use class, subclass and instance as the basis of the inheritance mechanism, inheritance of structural, default, and behavioral information is shared via the relationships *instance-of* and *class-of*. All classes that are subclasses of some class have access to (inherit) information associated with that class in addition to their own information. All objects that are instances of some class share the information associated with that class (ie, the information accessible from that class).

Thus, the representation of a class needs to include a means of accessing inherited information, typically a pointer or list of pointers to a superclass or set of superclasses. The representation of an instance needs to include a means of accessing its class.

It should be noted that we have been dicussing classes and general objects as though they were in fact two kinds of structural entities. In fact, there are several good reasons why it is better to view a class as a special kind of object, the main one being that in doing so all of the previously discussed advantages of object-oriented programming systems become available to us in designing and implementing the system itself.

For example, a class object could have a method for instantiating (creating an instance of) itself and perhaps a method for initializing a new instance. This would allow the class object to have control over how its instances are represented. Changing the representation of instances of the class (eg, adding and removing instance attributes) can be achieved by requesting the class object to do so. We can allow subclasses of class objects to represent their instances in a slightly different manner, but respond to the same protocols for making modifications. And of course we can keep all of the structural, default, and behavioral information pertinent to a class of objects where it naturally belongs — in the class object.

Representing classes with class objects means that we must define a class of class objects so that we can describe their structure and behavior to the system. Such a class is referred to as a metaclass. Metaclasses are provided in many existing systems, including Smalltalk, LOOPS, CommonLoops (but not Flavors). Metaclasses have some other attractve uses, such as,

(a) The ability to represent one kind of class variable (a variable that is shared amongst all instances of a class) — the kind that is allocated once for each class[2].

(b) The ability to determine a "precedence relation" amongst classes in systems that allow multiple inheritance. That is, when a new class is defined, its class object can be requested to produce an ordering on the set of classes from which the new class inherits information. Such an ordering is critical for resolving the kinds of conflicts that arise in programs that make use of multiple inheritance.

---

[2] The more typical knd of class variable is that which is allocated only once for a given class *and* all its subclasses. These two different extents of the meaning of class variable — once per class and once per class-and-subclasses — are directly related to the two different uses of the word instance alluded to in the previous footnote.

Because a primary objective is that the OOP system provide a common base for implementing different kinds of inheritance mechanisms, many of which will include multiple inheritance schemes, the system will benefit greatly from a highly modular way of computing precedence relations. Class objects are the natural way to provide this modularity in an object oriented programming system.

The POPLOG library package, LIB FLAVOURS, provides something approaching class objects called *metaflavours.* However, they are not currently used as class objects in the sense that I propose in that they do not themselves represent the information associated with a class. Rather, they are instances of a special class of object which provides methods for implementing some of the desirable kinds of behavior mentioned above (eg. they have methods for creating instances, adding and removing attributes and methods, etc) but does not describe attributes for storing information about the class. In effect, they act as a convenient interface to the data structure that actually represents a class (a flavour_record). So they provide many of the benefits of the three key concepts of object-oriented programming systems with regard to manipulating knowledge about the system itself — they can even be used to compute class precedence relations, though this important feature is not treated uniformly at present — but the structural and behavioral information about classes is not localised; it is dispersed into two places (flavour_records and metaflavours). *The LIB FLAVOURS library might benefit substantially and dramatically in terms of expressive power by applying the principle of data abstraction to the system's knowledge about itself.*

2.2. Method combination

In a non-object-oriented language that provides the ability to associate operations with data-types, one can define an operation which is carried out differently for each different type of data. The dependency between the method for performing an operation and the type of data for which the method is appropriate is captured explicitly, allowing for a reduction in run-time errors, but if one wants to add a new global operation — one which can be applied to all types of data — one must define as many different methods as there are data types. This can be a heavy burden both in terms of ease of applications development and in space requirements.

Inheritance reduces this burden by allowing slightly different types, or classes, to share the same methods where appropriate. When it is *inappropriate* for an inheriting class to use a method provided by one of its ancestors, it can override, or *shadow,* the inherited method by providing one of its own. However, in many cases the shadowing method must augment the behavior provided by the shadowed method; that is, it needs to do the same thing that the shadowed method would have done, plus something extra.

To illustrate, suppose one defines a class 2-d-point. having instance variables x and y. and a print method that prints the values of x and y; then one defines a class 3-d-point. which inherits from 2-d-point, and provides an instance variable z. Clearly, instances of 3-d-point should not merely use 2-d-point's print method without alterations because their value for z wouldn't be printed. On the other hand, completely shadowing 2-d-point's print method would be wasteful because most of the code would have to be duplicated. This is where method combination becomes extremely useful.

What one wants is to have instances of 3-d-point respond to print requests by using 2-d-point's print method to print their values for x and y, and then use their own method for printing the value of z.

Method combination is in fact so useful that practically every object oriented language provides a mechanism supporting it. The two most common styles of method combination are "send-super" and Flavors' "daemon" combination.

Send-super is the most general as well as the most prominent of the two styles of method combination, appearing in Smalltalk-80, LOOPS, CommonLoops, HP Common Objects, Object Lisp, and others. In this scheme, method combination is specified by including a syntactic statement in a shadowing method. The statement indicates that processing of the current method is to be suspended while the request that caused it to be invoked is re-issued to the same object — only this time the object is asked to respond by invoking whatever method would have been invoked in the first place had it not been shadowed by the current method. When processing of the shadowed method is complete, processing of the current method resumes. Send-super statements can be dynamically nested, allowing any number of shadowed methods for the same operation on a particular class of object to be combined. Thus the print method for 3-d-points could be defined so as to first call send-super, and then print the value of the instance variable z. Defining a new print method on a class of 4-d-point objects, inheriting from 3-d-point, and contributing an instance variable w, would consist of first printing the value of w, then calling send-super to print the other values.

In the Flavors daemon style of method combination, methods are specified as being either "primary" or "daemon" methods. Daemon methods can be tagged as either `:before` or `:after`, indicating that they are to be run before or after the primary method, respectively. Each class, or flavor, can define at most one each of primary, before, and after methods. When a flavor is compiled, it is associated with a "combined method" for each operation its instances can perform. The combined method is a procedure that invokes all of the before daemons for the operation (ie, the before daemon defined by the flavor itself, if any, along with any inherited ones), followed by a single primary method, followed by all after daemons. The order of invocation normally (ie, by default) reflects the nesting of flavors in the precedence list of inherited flavors, so that before daemons are invoked in order of most specific to most general flavor while after daemons are invoked in the reverse order. The primary method invoked is the first primary method found in the precedence list, ie, that contributed by the most specific flavor to provide one. Thus, using daemon style method combination, one could implement the print method on 3-d-points by simply defining an after daemon which prints the value of the instance variable z. The combined method would invoke the primary print method, resulting in the values of x and y being printed, and then invoke the after daemon, resulting in the value of z being printed. The print method for 4-d-points could be a before daemon to print the value of w.

From the examples one should be able to observe the inherent modularity of both approaches. In each case, one need only add code that takes advantage of pre-existing code without redundancy or the need for modification.

The appeal of Flavors' daemon style method combination lies mainly in the way it allows operations to be decomposed into a chunk of information describing some general or basic behavior (the primary method) and one or more other chunks

describing some extra or more specific behaviors which are activated or triggered by a request for the operation (daemon methods). The syntax for specifying that a method is to be treated as primary, before, or after, also permits a number of other, less common styles of method combination. For example, in defining a flavor, one can indicate that one or more methods are to be combined using the `:or` style of combination (`:daemon` is the default style in the Flavors system). The combined method constructed for `:or` tagged methods invokes each method found in the precedence list in turn, until one returns a non-nil (Boolean true) valued result. Some of the other styles of method combination supported by the Flavors system include:

`:and`	Invoke each method until one returns nil.
`:progn`	Invoke each method and return last result (in the `:daemon` style, the combined method returns whatever the primary method returns).
`:list`	Return a list of results obtained by invoking each method.
`:case`	Dispatch on subsidiary method name.

It is also possible for users to define their own idiosyncratic styles of method combination; for example, one could define a style called `:max` which would return the maximum (numeric) result obtained by invoking each method.

There are two main problems with the Flavors approach to method combination:

(1) The combined method for a particular flavor may have to be recompiled when ancestral flavor descriptions are modified;

(2) It is more difficult to control the invocation of individual methods, since control is automated by the combined method.

Using combined methods allows method combination to be made very efficient since method lookup for primary and daemons can be performed when the combined method is created (at compile-time rather than run-time)[3]. A send-super mechanism requires shadowed methods to be located at run-time. However, if an inherited flavor redefines, or undefines, one of the individual methods used to construct the combined method, then the combined method has to be recompiled. This could lead to something of a bottleneck during the initial development phase of rapid prototyping.

Send-super statements are pure programming language expressions and, as such, can be conditionalized, can return results, etc. Using send-super, the programmer can retain control over how and when method combination takes place. For example, a shadowing method could contain a conditional call to send-super, allowing the programmer to control whether or not the shadowed method is invoked.

Conditional method combination is not supported in the daemon style of combination. Flavors does provide a mechanism more functionally similar to the send-super approach in the form of *wrappers*: `defwrapper` can be used to obtain the effect of wrapping some code around invocations of shadowed methods. An example

[3] Note that the combined method itself must still be dynamically located.

of the need for wrappers is when a before daemon may want to prevent the primary method from being executed — a wrapper can be used in place of the before daemon to conditionalize the execution of the primary method. Using wrappers in conjunction with daemons is more complex and somewhat inelegant as a model of method combination.

Another problem with automatically constructed combined methods is the inability of an individual method to modify arguments before the next method (in the combined method) "sees" them, or to make use of the results of previously invoked individual methods. Again, these capabilities are more easily handled with send-super, since send-super expressions can be called with arguments different from those used in the original request and their results can be used afterwards in the body of the shadowed method. Finally, a send-super expression can occur any number of times, or within a loop inside a method definition.

Both the send-super and the Flavors daemon styles of method combination appear to have many proponents. The daemon style trades off expressive power for efficiency as well as a cleaner decomposition, at least as far as looks go[4]. In order to regain the generality send-super provides, by leaving control over method combination in the hands of the programmer, something like wrappers must be provided, resulting in a less elegant model of method combination. However, the Flavors model of method combination has the advantage of readily extending itself to support alternative styles of method combination, even user-defined ones.

It might be worthwhile for a system designer to explore the possibility of combining the nice decompositional aspects of Flavors style method combination with the more general expressive capabilities afforded by send-super, while attempting to retain a clean and extensible model of method combination in general. In the meantime, my advice would be to provide both existing styles of method combination at the medium level of the framework, with precedence being given to send-super because of its greater expressiveness and its arguably wider popularity.

2.3. Multi-methods

One of the primary motivations underlying the principle of data abstraction is to capture the dependencies between procedures and the types of data they can be applied to.

If the programmer is thinking purely in terms of objects that perform operations on themselves, then it is appropriate to treat an individual object as the (sole) controller of the methods employed in performing those operations. This is true of systems that employ the notion of sending messages[5] (eg, Smalltalk, LOOPS, (old) Flavors, LIB FLAVOURS, LIB NEWOBJ).

---

[4] Conceptually, send-super decomposes combined behavior into the same sort of dynamically nested structure, although the decomposition may not be as cosmetically apparent.

[5] "Sending messages" is the more familiar way of referring to what I have been calling "issuing requests". I chose to avoid the former phrase because I don't regard it as one of the key concepts of object oriented programming systems; certain systems I would definitely regard as object oriented (eg, CommonLoops, (new) Flavors, ObjectLisp) never resort to using the phrase in their literature. The message-passing metaphor is not nearly as central to object oriented systems in my opinion as is the dynamic binding of methods — in fact, the use of message-sending syntax is just one way of indicating that a method should be selected dynamically in a manner controlled by the "message receiving object" to perform a particular operation.

In systems like CommonLoops and (new) Flavors, dynamic method invocation looks no different than ordianary function calls. In these systems, the user doesn't need to know whether something has been implemented as a method or as a function, and this can be an advantage (eg. programs can become incrementally "object oriented" by redefining functions as methods; users don't need to try to guess whether to achieve some operation by sending a message or calling a function).

Traditional method invocation is supported in these systems by having the item in the functional position of a list (LISP syntax) *discriminate* dynamically according to the class of the first argument. The obvious generalization, as discussed in the chapter on recent trends, is to allow discrimination based on zero or more arguments. In this way we can capture more of the dependencies between procedures and data-types (eg. dependencies involving n-ary functions).

It is certain that there are some classes of problem for which it is more expedient to think in terms of procedures than in terms of objects, and others for which thinking purely in terms of one or the other is less well-suited to the task than thinking in terms of both. In line with the current philosophy of building hybrid systems that allow more than one methodology or approach to be brought to bear on general problem solving. I think it would be wise to try to accommodate both kinds of orientation. Howver, because the orientation of systems like CommonLoops and (new) Flavors appears to be swinging back a little toward thinking in terms of procedures that perform operations on data. I believe that support for multi-methods does not belong in the base-level.

2.4. User-defined inheritance

As has been suggested a number of times throughout this report, the exclusive reliance on specialization and instantiation, based on the supporting notions of class, subclass, and instance, as a basis for the sharing of information poses a limitation on the kinds of problems that can be modeled naturally and effectively within the object oriented programming paradigm. Typically, a programmer wishing to adhere faithfully to the paradigm is forced to use specialization to achieve economy and modularity even in situations in which some other kind of relationship might in fact be more effective. In this section I provide a rough sketch of how a more general. user-defined inheritance mechanism might be provided.

As an illustration of the problem, suppose one is attempting to model a situation involving a number of projects, project leaders, and other people involved in working on the projects whom we will call project workers. We can start by defining the classes project, project-leader, and project-worker. Both projects and project workers are associated with a project leader. We can capture this by defining a slot for each of these two classes called lias-project-leader. Both project leaders and project workers are associated with a project. This could be represented by defining a slot for each of these two classes called lias-project. Note that there is already potentially a great deal of redundant information, since each project worker for a given project has the same project leader.

If we are limited to the use of specialization as a means of eliminating this redundancy, we will need to sit back and think about how to redesign the representation. One solution might be to define classes corresponding to each project which would capture information relevant to all project workers associated with that

project. For example, for project 1 there could be a class called project-1-workers with a class slot (shared amongst instances) called has-project-leader, eliminating the need for such a slot in instances of project-worker. Since each project-*X*-workers class would be similar, we could define a superclass to capture the similarity. However, this organization is not as flexible as before, since we now have classes that are tied to specific projects as well as instances of specific projects[6]. There is still some redundancy as well since both the object representing a project and the object representing those working on the project have has-project-leader attributes which will always contain the same instance of project-leader. This might suggest that a project and a project leader could be combined through multiple inheritance into a class we might call project-with-leader; however, we would then be dealing with a rather artificial entity — a thing that is at the same time both a project and a project leader. Either way, as the application grows more complex, we may well find that the class inheritance lattice needs to be restructured or redesigned many times.

The interesting thing to note is that there are a number of naturally occuring relationships amongst projects, project workers, and project leaders that could potentially serve as effective means of capturing and expressing shared information. What we would really like to be able to express is the equivalent of the english statement, "The project leader for someone working on project X is the same as the project leader for project X", or "To determine one's project leader, look up the information in one's project." Trying to express this kind of inference knowledge within the structure of a specialization lattice can result in less natural and hence less flexible knowledge representations.

In order to eliminate the rigidity of an inheritance mechanism tied to a particular kind of relationship, a means of explicitly representing general inheritance relationships (those along which information can be shared amongst objects) is called for.

Note that in most if not all cases, an object attribute can be considered to be the name of a relation which holds between the object and the attribute's value. So, for example, a project worker is related to a project by the has-project relation. A user-definable inheritance scheme would allow one to specify the relationships that can be used to draw particular inferences about an object; for example, one could indicate that one or more named pieces of information pertaining to a given object or object class can be obtained through a particular attribute if the information is not directly represented within the object. This would allow us to say that if an object representing a project worker does not have a has-project-leader slot, then its project leader can be determined through its has-project slot.

This kind of meta-knowledge — in this case knowledge of how to draw a particular inference — would appropriately be represented in the metaclass for project workers[7]. The metaclass for project workers could have a slot named inherits-via containing a table which associates a request for the project leader with the attribute has-project.

Conceptually, every class object would have a slot named subclass-of whose value would be a list of class objects. Every instance would have a slot named

---

[6] The principle that as much information relevant to a particular entity as possible should be represented within a single object has been violated.

[7] That is, in an object representing information about the project-worker class, as opposed to information about its instances.

instance-of whose value would be the class object from which it was instantiated. This would allow specialization to be used as a default relationship when no other relation is specified for a particular search item.

Based on the design outlined above, the inheritance mechanism would work as follows: Given a request for a particular piece of information, determine whether the information is represented directly in the object; if so, return it; if not, use the **instance-of** attribute/relation to determine the class of the object. If the class object found therein has an **inherits-via** attribute, use the table there to determine which attribute/relation to proceed searching in; if the class object has no **inherits-via** attribute, or if the table doesn't have a mapping for the request, proceed by searching in the **subclass-of** attribute/relation of the class object.

In order for the inheritance mechanism to maintain some level of integrity as well as control over complexity, it is necessary to impose the restriction that once a particular kind of relation has been decided upon (through the advice of the **inherits-via** attribute) the system uses that relation throughout. This means that having determined the class of the object in question, if inheritance initially goes through the **subclass-of** relation, then it will proceed only via **subclass-of** relations; similarly, if it initially goes through the **has-project** relation, it will stick to that kind of relationship.

Since relationships are not always of the binary kind, it is necessary to have a means of guiding the search for information through multiple paths in a particular relationship lattice. For example, a project worker might be associated with a list of projects rather than only one, raising the question of which project to pursue first. Furthermore, the search process may need to continue as a result of going through a given relation[8]. In order to have maximum leverage over the search strategy, one needs to be able to define a method whose function is to determine a search path through a relationship lattice. Pre-defined functions could be provided for standard search strategies such as depth-first, breadth-first, progressive deepening, etc.

Search methods can be considered as part of an inheritance relation, suggesting the need for a pre-defined class of objects used to represent information about relations. Other attributes that might usefully be linked to relationships are

- inverse-relationships
  A slot indicating the inverse of a relationship: eg, the inverse of **instance-of** is **has-instances**.
- transformations
  A slot containing a mapping table used as a filter to transform values inherited via the relationship.

Inheritable relations can be categorized within the class lattice as a kind of attribute through which information can pass[9]. Thus, in addition to the kinds of attributes mentioned above, they would pick up other useful slot attributes such as cardinality, type restrictions, daemons, defaults, and documentation (by being a subclass of the class **slot**, which would be used to describe such meta-attributes).

---

[8] as implied by the **subclass-of** relation with multiple superclasses.

[9] The inverse of an inheritable relation is usually not an inheritable relation, so this variety would be differentiated from inheritable relations.

3. Implementational Issues

3.1. Efficient method lookup

In typical object oriented programming systems, each class of objects is associated with a set of methods which implement the operations or behaviors defined on instances of the class. Methods can be shared by defining inheritance relationships between classes so that instances of a class have access not only to the methods associated with that class, but also the methods associated with other classes from which the class directly or indirectly inherits methods. When an instance of a class is requested to perform an operation, its method for responding to the request must be dynamically located by searching through an ordered sequence of classes from which its class inherits methods[10]

The dynamic lookup of methods typically represents the main botteleneck in performance of OOP systems, and therefore a good deal of effort has gone into determining what can be done to make method lookup more efficient. The main result of these efforts has been a moderately successful solution known as ***method caching***.

In a Central Processing Unit, the term "cache" refers to a portion of very fast storage, generally transparent to user programs. It is kept constantly updated to hold the most frequent memory references, so as to greatly speed future references, thereby facilitating dynamic address translation. In object-oriented programming systems, the term "method caching" refers to the technique of maintaining a fast-access store containing the addresses of frequently executed methods. By storing a fixed number of addresses to the most freqently invoked methods, each method can usually be located in a relatively small number of steps, thus significantly reducing the aggregate time spent performing method lookup.

A fair amount has been written concerning method caching [cf. Krasner 83]. In this section, I merely give an outline of a fairly simple method caching scheme in order to put across the general idea.

Since every method can be uniquely identified by specifying its name (selector) and the class with which it is associated, the simplest approach to implementing a method cache is to use a hash table, where the hashing function takes a class and selector as arguments and returns an index into the table.

The entries in the cache should be fixed size sequences (I will assume they are linked lists) containing method records[11]

Assuming that the number of methods defined is fairly constant and the size of the cache is considerably smaller than the total number of methods, the most reasonable strategy for improving method lookup is to fill the cache with the most frequently used methods. One way of doing this is for each method invocation,

(1) Apply the hash function to the class of the object and the message selector (method name) to obtain an index i.

---

[10] Note that classes are usually considered to inherit from themselves — so the instance's class is in the sequence.

[11] The method records should be structures having fields for the method name, class name, and a pointer to the method, for reasons which will become apparent shortly.

(2) Search the linked-list at i to determine whether the method is present in the cache.

(a) If the method is in the table, bring it to the front of the list at location i before proceeding to execute the method.

(b) If the method is not in the table, attempt to locate the method by searching the inheritance chain for the object's class. If this succeeds in locating the method, update the cache by constructing an appropriate method record for the method (if necessary) and pushing this onto the front of the list at location i before proceeding to execute the method. Note that in order for the list to remain a fixed size (actually not to exceed a maximum size, since it will start out empty) one of the list elements may have to exit the list (and hence the cache) as a result of the update; this should be the element at the opposite end of the list. If the method was not located as a result of the search through the inheritance chain, take whatever action is specified as occurring when a method cannot be located for a particular request (eg. signal an error).

It should be noted that whenever a method is defined, the cache may need to be updated (ie, only when a method is being redefined and the address of the old definition is in the table).

How does the above algorithm adhere to the strategy of retaining the most frequently used methods in the method cache? By constraining each list in the cache to have a maximum size, methods must begin to exit the cache as each list reaches this maximum during updates. The lists are maintained in such a way as to reflect the ordering of invocation frequency by placing the most recently used method at the head of a list and discarding the method at the tail when a list is full. Frequently used methods will tend to remain in the cache while infrequently used ones will soon exit to make way for new ones.[12]

As indicated above, the method cache needn't use physical list structures (the overhead of removing and inserting methods might be too high); it is sufficient to map a set of priorities for leaving the cache onto the set of methods that hash to the same location. Method records containing class and method names, as opposed to merely the address, are necessary to distinguish between all of the methods that hash to the same location.

This scheme is similar to the "working set" model used in paged virtual memory systems, and therefore works best when

(a) a certain set of methods are invoked significantly more often than the rest;

(b) the cardinality of this set is the same as the cache size;

(c) the set is relatively stable for long periods of time.

As one or more of these ideal conditions is relaxed, the improvement in performance of method lookup tends to be eaten up by the overhead of maintaining the cache. Assuming that the majority of application programs will possess something resembling working sets of methods, the main problem becomes identification of the average size of these sets so that the optimal cache size can be determined. To my knowledege there are no theoretical results for determining the optimal cache size, however a few empirical results appear in [Krasner 83].

[12] Of course, this is only an approximation of most frequently used. At given times the cache may not contain the most frequently uied methods, but most of the time it will. It would be more

It is clear that the optimal cache size will be application dependent, and might also depend on language and implementation specific factors, such as the number of predefined methods and the way in which method priorities (for leaving the cache) are updated. Probably the best advice is to experiment with different cache sizes over a wide range of large and small applications. It should also be pointed out that certain applications may not have the equivalent of working sets of methods; for example, an application might be designed that causes each method in the program to be invoked in repeated sequence (overlooking the fact that such a design would be unable to take advantage of many obvious benefits of OOP, such as the ability to *add* methods without modifying code). Applications in this category would suffer even worse than normal due to the overhead imposed by a method caching scheme like that outlined above. For this reason, it may be desirable to give the programmer some control over method caching, for example by allowing cache parameters to be user-modifiable.

accurately termed "most recently used", but that label doesn't indicate the intent of the strategy.

1. HP Common Objects

1.1. Comparisons with other languages

(1) In most object oriented programming languages, objects contain storage for no more than one instance variable with a given name, regardless of the number of superclasses of the object which define that instance variable. In Common Objects, instances of the class contain as many distinct storage locations for an instance variable as the number of superclasses that define it (including the class itself as well as duplicate superclasses). So, if instance variable x is defined in classes c1, c2, and c3; c2 inherits from c1; c3 inherits from c2 and $c1_f$ then instances of c3 contain *four* distinct instance variables named x (one from c3, one from c2, and two from c1).

(2) Most object oriented programming languages allow any method defined on a class to directly access instance variables inherited from superclasses (ie. perform "get" and "put** operations on them just like an ordinary variable). Common Objects disallows direct access of an inherited instance variable from any methods other than those defined on the class which defines the inherited instance variable (including inherited methods). Thus, if y is an instance variable defined by class c1, and class c2 inherits from c1, then methods defined by c2 may only access y by invoking methods defined on c1.

(3) In most object oriented programming languages, if two or more classes define methods with the same name, then a class which inherits from these classes only inherits one of these methods (selected according to some precedence relation defined on the superclasses of the inheriting class). In Common Objects, if a class attempts to inherit from two classes which define methods with the same name, an error is signaled, thus, the programmer must explicitly indicate from which of the superclasses the method is to be inherited.[1]

1.2. Efficiency

- In general, when invoking a method named foo, as in

  (-> x :foo argl arg2 arg3)

  a run time lookup is required to identify which method to invoke.[2] because the type of the object denoted by the variable x is not statically known, unless its

[1] **An ciror will not be signaled so long as the :ggthOflS *IGXO&pt* options to 11nherlt-fTQn\ o the !T9d9f^'nfti--ynflrh'h0fig option to dBf-llTO-typS are used to restrict which methods the class inherit in such a way that no more than one method with a given name is inherited.**

[2] **The need for dynamic binding of method names to code segments is one of the hallmarks of the object oriented programming paradigm.**

type happens to have been declared.[3] The table lookup is generally cheaper than a search through nested, multiple-superclass lists. The table can be reorganized in order to make certain time-critical methods cheaper to look up (at the cost of making others more expensive to look up).

- A method can be declared as "inline", causing the compiler to directly substitute the code that performs the method in all statically determinable invocations, thus avoiding the overhead of message sending. This may lead to increased code size in return for a possible payoff in speed (It is only recommended for very simple methods, such as those which merely return a result or send a single message). Certain automatically generated methods are inline by default, including those that access inherited instance variables (so, compiled code does in fact directly access inherited instance variables while source code does not).
- Invoking methods from within a method defined on the same type of object, or on a directly inherited type, using `call-method` or `apply-method` requires no run time method lookup since both the object type and the method name are statically known.
- The Common Objects model of method inheritance implies that if type A defines a method for the operation foo, and type B inherits the foo method defined on objects of type A, then a method is defined on objects of type B for the foo operation which simply applies the method defined on objects of type A. If t_i defines a message for the operation baz, and t_i inherits from t_{i-1} (for i an integer, $1 \leqslant i \leqslant n$), then n different methods exist for the operation baz. The Common Objects definition leaves it up to the implementor to decide whether to try to minimize the overhead of indirect method invocation[4] or to eliminate the overhead by having the compiler produce code in which all of t_i share the same method. In the latter case, Snyder points out, "more work must be performed by the implementation to present the illusion that each type has its own method".

2. Flavors

2.1. Implementation

Objects are implemented using a new, built-in data type `dtp-instance`. The first word of this structure points to an "instance descriptor" containing the internal data for the flavor. The remaining words are value cells containing the values of the instance variables. The instance descriptor is a `defstruct` appearing as a property of the flavor name, and containing information such as the name of the flavor, the size of an instance, a method dispatching table and information for accessing instance variables. Message sending is supported by defining a convention on the way objects are applied to arguments (this was true for the old Flavors system anyway).

---

[3] Note that a declaration to the compiler that x is of type t would mean that types that inherit from t could not be substituted for x, thus reducing the benefits of modularity.

[4] Note, for example, that the overhead of executing the baz method for objects of type t_k is greater than for objects of type t_j for $j < k$.

3. LOOPS

3.1. Efficiency

Loops uses a small method cache to improve performance during method lookup. Initial loading of a method is slow, due to method lookup being performed by searching the inheritance lattice. Methods in the cache are found quickly — approximately 2 to 3 times the amount of time of a normal Interlisp function call. The estimated hit rate is approximately 90 percent.

A Loops `GetValue` takes about 10 times the time to perform an Interlisp variable lookup (due to the check for `ActiveValues`). A large, tested example program spent about 30 percent of its time in `GetValue`.

Instances are created with space allocated for all their instance variables (except for `VirtualCopy` instances). The overhead above and beyond this space is 5 to 6 words per instance.

4. C++

4.1. Implementation

C++ is a superset of C (almost) which enables data abstraction by supporting user-defined types, called "classes." Classes are like C `structs`, the first difference being that instance slots (what C calls "members") can be functions; the second difference is that access restrictions can be placed on specified members. A class declaration is divided into "public" and "private" parts. The private members of an instance can only be accessed by public members, or by an explicitly declared set of functions, called "friends." When a class is defined, constructor functions are generated for creating and initializing instances of the class. Member functions can be overloaded, so that the compiler chooses one based on the types of the arguments. Operators can be defined as member functions, or friends, and can be overloaded. Dynamic storage handling can be hidden from users of objects. Classes can be arranged into abstraction hierarchies (single class inheritance) Functions can be declared as virtual to allow dynamic typing of objects. A new data type, "references" can be used to impose call-by-reference semantics on function call.

4.2. Features

One of the most important contributions of C++ may be the notion of *friend* functions. For example, a class of **complex** numbers can be defined having the usual **real** and **imaginary** instance variables and specifying (non-member) *friend* operator functions whose arguments must be complex numbers (the operators can be overloaded so as to be generic). According to Stroustrup, the need to occasionally give method selection responsibility to the function, rather than a "receiving" object (a là Smalltalk), is due here to the fact that "the inherent asymetry in the notion of objects does not match the traditional mathematical view of complex numbers". The problem of responsibility boils down to the question of asking for what kinds of N-ary operations is it the case that only one operand clearly should be given the

power to control the outcome of requests to perform the operation.

5. Objective-C

5.1. Implementation

A new. built-in type, `id`. is provided which can hold a pointer to an object.

An object in Objective-C is implemented as a block of memory consisting of a "shared part" and a "private part". The shared part of an object is shared by all instances of the object's class. It consists of a "template" defining the object's structure (storage for instance variables) and a "dispatch table" used to select the method (piece of executable code) that implements some named operation. The private part of an object is the set of storage locations for the object's instance variables. Only methods defined on an object (accessible via its dispatch table or the dispatch table of one of its superclasses) have access to the object's private part.

"Factory objects" are objects that can create "instance objects" of a given class (there is one factory object for each class). The `isa` instance variable, defined for all objects, is a pointer to the object's factory object. The shared part shared by all instances of a class is actually the private part of the factory object for the class.

A message sent to an object causes the `isa` link to be traversed, so as to access the (shared) dispatch table for the class. If an applicable method is not found there, the superclass's dispatch table is accessed, and so on. When an appropriate method is found, the method is executed and any value it returns is returned to the calling environment (ie, from which the message was sent). Thus, Objective-C provides dynamic binding.

The shared part of factory objects contains the dispatch table used in responding to messages sent to the factory object (eg. n6w) and the structural description of the factory object (the shared part of a factory object is only shared by itself, since there is only one per class).

There are several system-provided classes for example. `Object`, `Array`, `Bag`, `Collection`, etc.

Appendix A — Object Pascal

The following example of the Object Pascal syntax is taken from [Doyle 86].

```
Type
    Shape - Object                          {Like a record definition}
        bound1ngRect:     Rect;
        color:            integer;
        Procedure Draw;                     {...with method fields}
        Procedure Erase;
        Procedure MoveBy(deItaH, deltaV: integer);
        Function Area:    integer;
    End;

    Circle - Object(Shape)                  {Circle inherits from Shape}
        Procedure SetRadius(radius: integer);
        Procedure Draw; Override;{Override the Inherited Draw}
        Function Area: integer; Override;f...and Area methods!
    End;

Procedure Shape.Erase;
            {method implementation qualified with object type}
Begin
    EraseRect(boundingRect);
End;

Procedure Circle.SetRadius(radius);
Begin
    boundingRect.right :• boundIngRect.Ieft + radius * 2;
    boundingRect.bottom :- boundingRect.top + radius * 2;
End;
```

```
{Object variables ore references to the object}

Vor
     aShape:       Shape;
     oCtrcle:      Circle;

oShape.color  :• blue;
oCI re Ie.bound 1ngRect  :« myRect;
oShape.MoveBy(10.2e);                          {Method invocations

Procedure Triangle.MoveBy(deltoH# deltaV:  integer);
Begin
     MaverVertices(deltaH. del! :V);
     Inherited MoveBy(deItaH, oeitaV);
          {Call* Shape.MoveBy –  like  *-Super}
End;

{Allocate memory for the object data and type indicator!

New(aShape);
New(aCi re Ie);
```

BIBLIOGRAPHY

"The Smalltalk-80 System," BYTE, vol. 6, no. 8, pp. 36-49, August 1981. Describes how message sending objects are used in the Smalltalk-80 system.

The Portable Standard Lisp User's Manual, Utah Symbolic Computation Group, Department of Computer Science, University of Utah, January 1983.

KEE: The Knowledge Engineering Environment, IntelliCorp, Knowledge Systems Division, Menlo Park, California, 1984. A technical overview, presenting a demonstration application.

"Objective-C," Sample Reference Manual, Productivity Products International, Inc., Sandy Hook, CT, 1984. The reference manual for Objective-c, with selected chapters omitted.

Introduction to Knowledge Craft, Carnegie Group, Inc., Pittsburgh, PA, 1985.

"BLOBS is the language for simulation and reasoning," Sensor Review, pp. 187-192, October 1985. An Object-Oriented Blackboard package written in Pop11.

"Knowledge Craft Overview," Software Version 3.1, Carnegie Group Inc., Pittsburgh, PA, July 1986.

Agha, G.A., "Actors: A Model of Concurrent Computation in Distibuted Systems," PhD Thesis, MIT-TR 844, MIT AI Lab, June 1985. A PhD thesis dealing with operational semantics of Actor languages.

Allen, E.M., R.H. Trigg, and R.J. Wood, "Franz Lisp Environment, Variation 2," TR-1226, Maryland Artificial Intelligence Group, University of Maryland, College Park MD, November 1983.

Althoff, J.C., "Building Data Structures in the Smalltalk-80 System," BYTE, vol. 6, no. 8, pp. 230-285, August 1981. Demonstrates how easily many kinds of data structures can be added to the Smalltalk-80 system.

Anjewierden, A., "Object Orientation in PCE-Prolog," Journees Langages Orientes Objet, pp. 167-176, AFCET-Informatique, Paris, January 1986.

Attardi, G. and M. Simi, "A Description Oriented Logic for Building Knowledge Bases," ESP/86/3, Delphi SpA, Viareggio, Italy, April 1986.

Attardi, G., A. Corradini, S. Diomedi, and M, Simi, "Taxonomic Reasoning," ESP/86/2, Delphi SpA, Viareggio, Italy, March 1986.

Barrett, R.f A. Ramsay, and A. Sloman, POJD-IJL: A Practical Language for Artificial Intelligence, Ellis Horwood'^Ltd., Chichester UK, 1985. The reference~for the Pop-11 language.

Barstow, D.R., H.E. Shrobe, and E. Sandewall, Intera£tj.ve Programming Environments, McGraw-Hill, New York, 1984.

Betz, D., "XLISP: An Experimental Object-Oriented Language," Version 1.4, Manchester, NH, January 1985. Documentation for XLISP: a highly portable version of LISP with object-oriented extensions.

Betz, D., "An XLISP tutorial," BYTE, vol. 10, no. 3, pp. 221-236, March 1985. A revised version of [Betz 85a].

Bobrow, D.G., "An Overview of KRL: A Knowledge Representation language," CogTiiJtrve Science, vol. 1, no. 1, pp. 3-46, 1977. Describes KRL, a language based partly on Minsky's work on Frames.

Bobrow, D.G. and I.P. Goldstein, "Representing Design Alternatives," Proceedings of AIjSB?, Amsterdam, July 1980. A description of an experimental personal information environment which provides users with descriptive structures for programs and documents.

Bobrow, D.G. and M. Stefik, The LOOPS Manual, Xerox Corporation, Palo Alto, CA, December 1983. Reference manual for one of the major OOP systems in use today; a knowledge engineering environment that attempts to integrate several different programming paradigms (object oriented, rule oriented, access oriented, procedure oriented) on top of Interlisp.

Bobrow, D.G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, "CommonLoops: Merging Common Lisp and Object-Oriented Programming," ISL-85-8, Xerox Corporation, Palo Alto, CA, August 1985. Proposed object oriented programming facilities for Common Lisp, including, multiple, type-specified generic methods.

Bobrow, D.G. and M. Stefik, "Perspectives on Artificial Intelligence Programming Languages," Science, vol. ?, no. ?, pp. 945-950, February 28, 1986. More on~the familiar themes of integrating various programming paradigms and Loops features (browsers, guages, access-oriented programming, multiple inheritance, etc).

Bonnet, A., "Structured Objects," in Artificial intelligence. Promise and Performance, pp. 120-127, Prentice/Hall, Englewood Cliffs NJ, 1985. ~Short discussion of frames, schemata,

prototypes, and objects and their impact on knowledge representation.

Borning, A.H. and D.H.H. Ingalls, "Multiple Inheritance in Smalltalk-80," Unpublished Report, Software Concepts Group, Xerox PARC. A paper describing how to extend Smalltalk-80 to provide multiple inheritance.

Borning, A.H., "The Programming Language Aspects of ThingLab: A Constraint-Oriented Simulation Laboratory/[1] in ACM Transactions on Programming Languages, vol. 3, pp. 353-387, October 1981.

Brachman, R.J.,- "On the Epistemological Status of Semantic Networks," in N.V. Eindl^er (Ed), Associative Networks: Representation and Use~of KnowX£lg£]£L Commute;rs** PP* 3-50, Academic Press, New York, 1979. Contrasts capabilities and drawbacks of various network formalisms for representing knowledge.

Brachman, R.J., "I Lied about the Trees" Or, Defaults and Definitions in Knowledge Representation," The AI Majgazirie, vol. ?, no. ?, pp. 80-93, Fall 1985. On the~limits of inheritance with overriding for general knowledge representation especially, the inability to represent irrevocable, universal truths, and hence, definitions.

Burke, G., Introduction to NIL, MIT LCS, Cambridge MA, March 1983. NIL has limited, built-in, extensively-used Flavors capabilities; eg, complex method combination is replaced by [1]send-forward'.

Cannon, H.I., "Flavors: A Non-Hierarchical Approach to Object-Oriented Programming," Draft (unpublished), 1982. Influential, hard to find paper discussing conflict resoltion in multiple inheritance systems.

Chailloux, J., "VLISP: 10.3 Manuel de Reference," RT 16-78, Universite de Paris 8, Vincennes, 1978.

Charniak, E. and D. McDermott, "AI and Internal Representation," *n Introduction to Artificial *Inteiiigence*, pp. 1-31, Addison-Wesley, Reading MA, 1985.

Charniak, E. and D. McDermott, "Memory Organization and Deduction," in Introduction to Artificial. InteHjjence, pp. 442-351, Addison-Wesley, Reading MA, 1985. Deductive Inference using associative nets, etc.

Chikayama, T., "Unique Features of ESP," TM-0055, ICOT, Tokyo, April 1984. A discussion of object facilities and macro expansion in ESP.

Chikayama, T., ESP Refe£enc£ Manual, ICOT, \ A useful reference describing Japanese terns.

Clayton, B.D., ART Programming Primer, Inferen Angeles, CA, April 1, 1985.

Clayton, B.D. , ART Programming Tutorial,f Volume Corporation, Los Angeles, CA, March 15, 1981

Clayton, B.D., ART Programming Tutorial, Volume Two, Inference • Corporation, Los Angeles, CA, March 15, 1985.

Clocksin, W.F. and C.S. Mellish, Programming ijn Prolog, Springer Verlag, Berlin, 1981. The main reference for the Prolog language.

Cointe, P. and Rodet, "Formes: A New Object-Oriented Language for Managing a Hierarchy of Events,[11] Internal Report, Institute de Recherche et de Coordination Acoustique-Musique, Paris, 1983. Describes an object oriented language written in VLISP [cf$_t$ Chailloux 78] designed for music composition and synthesis.

Cointe, P., "The extension of VLISP through the objects," Sci. Comput. Program. (Netherlands), vol. 4, no. 3, pp. 291-322, December 1984. Yet another object oriented extension of Lisp.

Cointe, P., "FORMES: Composition and Scheduling of Processes," Convgut. Music J. (USA), vol. 8, no. 3, pp. 32-50, Fall 1984. Applications~of OOPS in computer music.

Cook, S., "Languages and Object-Oriented Design," H2E CoHosium *an* Object-based design (Dijjest No. 88), IEE. London, England, November 1984.

Cox, B. J., "Message/Object Programming: An Evolutionary Change in Programming," iEE^E Software, vol. 1, no. 1, pp. 51-61, January 1984. A discussion of the main differences between procedure oriented and object oriented programming.

Dahl, O-J., B. Myrhaug, and K. Nygaard, "SIMULA Common Base Language," Report S-22, Norweigian Computing Center, 1970.

Dahl, O.J. and K. Nygaard, "SIMULA - An Algol-based Simulation Language," rpmmnn "iratinne rlf the ATM xrni Q ^^ 671-678

1984.

DeSmedt, K., "Using object-oriented knowledge representation techniques in morphology and syntax programming," in ECAI-84: Advances in Artificial Intelligence, pp. 181-184, Elsevier Science Publishers, North Holland, 1984. This paper should be interesting to those wanting to use OOP techniques in computational linguistics.

Deutsch, L.P., "Building Control Structures in the Smalltalk-80 System," BYTE, vol. 6, no. 8, pp. 322-347, August 1981. Illustrates the ease with which complicated control structures are implemented in the Smalltalk-80 language.

diPrimio, F. and T Christaller, "A Poor Man's Flavor System - Part 1," Working Paper No. 47, ISSCO, Universite de Geneve, 1983. Describes a Flavors system Implemented in UCI Lisp [cf, Meehan 79].

Doyle, K., B. Haynes, M. Lentczner, and L. Rosenstein, "An Object Oriented Approach to Macintosh Application Development," Journees Langages Orientes Objet, pp. 46-54, AFCET-Informatique, Paris, January 1986. A description of MacApp, Object Pascal, Apple's work in Smalltalk.

Drescher, G. L., "ObjectLISP for experienced LISP programmers," 1401-0000, Lisp Machines International, Cambridge, MA, August 1985. Proposed object oriented programming facilities for Common Lisp.

Duff, C.B., "Designing an Efficient Language," BYTE, vol. 11, no. 8, pp. 211-226, McGraw-Hill, August, 1986.

Ferber, J., "MERING: An Object-Oriented Language for Knowledge Representation," in ECAI-84: Advances in Artificial Intelligence, pp. 195-204, Elsevier Science Publishers B.V., North Holland, 1984. A short but very interesting paper on an object extension to LISP.

Ferber, J., "Object Languages: A Matter of Messages," Micro Syst. (France), vol. 52, pp. 152-159, April 1985. The integration of procedure and data in the MERING language.

Gabriel, R.P., "Massively Parallel Computers: The Connection Machine and Non-Von," Science, vol. ?, no. ?, pp. 980-985,

Goldberg, A., Smalltalk-80: The Interactive Programming Environment, Xerox Corporation? Palo AJto, California, 1983. A very useful reference describing how to use the interactive graphics tools provided by Smalltalk.

Goldberg, A. and D. Robson, Smalltalk-ftp: The Langrjag^e and its Implementation. Addison-Wesley,""Reading, MA, May 1983. The main reference for the Smalltalk language and programming environment; carefully presented and well-written with complete details of the implementation of the virtual machine and selected parts of the virtual image.

Goldstein, I.P. and D.G. Bobrow, "Extending Object Oriented Programming in Smalltalk,[1]* in Conference Record of the 1980 Lisp Conference, pp. 75-81,~1980.

Greif, I. and C. Hewitt, "Actor Semantics of Planner-73," Working Paper 81, MIT, Cambridge, MA, November 1974. Presents formal definitions and descriptions of Actor programs.

Griss, M.L., E. Benson, and G.Q. Maguire, "PSL: A Portable Lisp System," Conference Record of the 1982 Symposium on Lisp and Functional Programming, August 1982.

Griss, M.L. and E. Benson G.C. Maguire, "PSL: A Portable Lisp System," in Conference Record of J:he 198,2 ACM S^jn£o§liiD 2J3 Lisp and Functional. Programming, pp. 88-97, August 1982.

Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages," A. I. Memo 410, MIT AI Laboratory, December 1976. Presents the basis of Actor theory - communication between actors.

Hewitt, C. and H. Baker, "Laws for Communicating Parallel Processes," AI Working Paper 134a, MIT AI Laboratory, Cambridge MA, May 1977. Presents laws that must be satisfied by computations involving communicating parallel processes, stated in the context of the actor theory.

Hewitt, C, "Using Message Passing Instead of the GOTO Construct," Working Paper 164, MIT AI Laboratory, Cambridge MA, May 1978. Presents a programming methodology using message passing in an environment with many processors.

Hewitt, C, "Concurrent Systems Need both Sequences and Serializers," Working Paper 179, MIT AI Laboratory, Cambridge MA, February 1979. Distinguishes two classes of concurrent languages: those which support the notion of a sequence of values and some kind of pipelining operation over the sequence, and those which support the notion of transactions and some way to serialize transactions; in actor theory this amounts to the distinction between serialized and unserialized actors; this paper presents the utility of modeling both in a coherent formalism.

Hewitt, C., G. Attardi, and H. Lieberman, "Specifying and Proving Properties of Guardians and Distributed Systems," AI Memo 505, MIT AI Laboratory, Cambridge MA, June 1979. Discusses Guardians, abstractions that can be used to regulate the use of expensive resources in a distributed system.

Hewitt, C. and H. Lieberman, "Design Issues in Parallel Architectures for Artificial Intelligence," AI Memo 750, MIT AI Laboratory, Cambridge MA, November 1983. Describes design goals for architectures which hope to overcome the limitations of the von Neuman architecture, discusses actor theory as an approach to deal with these issues.

Hussman, M., "IS-A Isn't Enough: Towards a taxonomix framework for intensional concepts," in ECAI-84: Advances in Artificial Intelligence, pp. 350-351, Elsevier Science Publishers, North Holland, 1984.

Hutchinson, A., Inheritance in a Data Base of Frames, Dept. of Computing, King's College London. Decribes a simple mechanism for generalized inheritance of slot values.

Ingalls, D.H.H., "Design Principles Behind Smalltalk," BYTE, vol. 6, no. 8, pp. 286-299, August 1981. An enlightening discussion of how the design principles underlying the Smalltalk-80 language have contributed to its power and usability.

Ingalls, D.H.H., "The Smalltalk Graphics Kernel," BYTE, vol. 6, no. 8, pp. 168-199, August 1981. A description of how the Smalltalk Graphics Kernel provides the interface through which all text and graphics are displayed in the Smalltalk-80 system.

Janlert, L.E., Studies in Knowledge Representation, University of Umea, Institute of Information Processing, Umea, Sweden, October 1985. Two long papers: `Modeling change - the frame problem', and `Pictures and words'; the first paper has a lot to say about hierarchical inheritance models.

Kaehler, T., "Virtual Memory for an Object-Oriented Language," BYTE, vol. 6, no. 8, pp. 378-397, August 1981. Describes how virtual memory techniques can be used in an object oriented system when the active memory space needed by a language is much larger than the amount of available memory.

Kay, A., A. Goldberg, and Eds., "Smalltalk-72 Instruction Manual," SSL 72-6, Xerox PARC, Palo Alto CA, 1972. Reference manual for the first version of Smalltalk.

Keene, S.E. and D.A. Moon, "Flavors: Object-oriented Programming on Symbolics Computers," Proceedings Common Lisp Conference, Symbolics, Inc., Boston, MA, December 1985. A short but informative paper on the New Flavor System.

Krasner, G., "Machine Tongues VII: The Design of a Smalltalk Music System," Computer Music Journal, vol. 4, no. 4, pp. 4-14, 1980.

Krasner, G., "The Smalltalk-80 Virtual Machine," BYTE, vol. 6, no. 8, pp. 300-321, August 1981. Describes how the use of a Smalltalk-80 Virtual Machine allows the system to be transported easily amongst different 16-bit microprocessors.

Krasner, G., Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, Reading MA, 1983. A useful collection of articles describing the experiences of several implementors of the Smalltalk-80 system.

Ledbetter, L. and B. Cox, "Software-ICs: A Plan for building reusable software components," BYTE, pp. 307-316, McGraw Hill, June 1985. Presents an approach to preserving reusability of software modules (eg, class definitions) borrowed from electrical engineering.

Lieberman, H. and C. Hewitt, "A Real Time Garbage Collector that can Recover Temporary Storage Quickly," AI Memo No. 569, MIT AI Laboratory, Cambridge MA, April 1980. An algorithm which makes short term object storage cheaper than long term, operates in real time; object creation and access times are bounded, the method works well with multiple processors and a large address space.

Lieberman, H., "A Preview of Act-1," AIM-625, MIT AI Laboratory, Cambridge MA, June 1981.

Lieberman, H., "Machine Tongues IX: Object Oriented Programming," Computer Music Journal, vol. 5, no. 1, pp. 34-50, 1982.

Lieberman, H., "Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object-Oriented Systems," Proceedings of AFCET Langages Orientes Objet, pp. 79-89, January 1986. A good introduction to some of the concepts used in Actor languages; contrasts alternative mechanisms for sharing information.

Liskov, B. et al, "CLU Reference Manu[illegible]," TR-225, MIT LCS, Cambridge MA, October 1979.

Marrin, K., "Software Tools, Tailored Architectures extend Lisp and Smalltalk capabilities. II," EDN (USA), vol. 29, no. 20, pp. 53-62, October 1984.

Meehan, J.R., The New UCI Lisp Manual. Lawrence Erlbaum Associates, Hillsdale NJ, 1979.

Meyrowitz, N., ObjectC Report, Brown University, June 1985. Discusses work on adding OOP to the C language, in a similar vein with to Apple's Object Pascal.

Middleton, S. and R. Zanconato, "BLOBS: An Object-Oriented Language for Simulation and Reasoning," Working Paper, Cambridge Consultants Ltd., Cambridge, February 1985. A rather glossy overview of an object blackboard package written in Popl1.

Miyoshi, H. and K. Furukawa, "Object-Oriented Parser in the Logic Programming Language ESP," TM-0053, ICOT, Tokyo, April 1984. Presents an interesting application written in ESP, illustrating the use of partitive inheritance.

Nilsson, N.J., "Structured Object Representation," in Principles of Artificial Intelligence, pp. 361-415, Tioga, Palo Alto CA, 1980. A more formal presentation of structured objects; emphasis on relationships between structured object representations (including associaitive nets) and deductive inference techniques, such as predicate calculus.

Novak, G.S. Jr., GLISP User's Manual, ??.

Pascoe, G.A., "Elements of Object-Oriented Programming," BYTE, vol. 11, no. 8, pp. 139-144, McGraw-Hill, August, 1986.

Pepper, J. and G. Kahn, "Knowledge Craft: An Environment for Rapid Prototyping of Expert Systems," Proceedings of the SME Conference on Artificial Intelligence for the Automotive Industry, Detroit, Michigan, March 12-13, 1986., Carnegie Group, Inc., Pittsburgh, PA, March 1986.

Rees, J.A. and N.I. Adams, "T: A Dialect of Lisp, or, LAMBDA: The Ultimate Software Tool," in Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, pp. 114-122, August 1982. Has aroused interest from the Common Loops community; message sending and function call have the same syntax; classical method discrimination.

Rich, E., "Structured Representations of Knowledge," in Artificial Intelligence, pp. 201-244, McGraw-Hill, Singapore, 1983.

Robson, D., "Object-Oriented Software Systems," BYTE, vol. 6, no. 8, pp. 74-89, August 1981. Discusses how object-oriented software systems provide the underlying design of Smalltalk.

Serlet, B., "Object Oriented Programming in Cedar," Journees Langages Orientes Objet, pp. 64-68, AFCET-Informatique, Paris, January 1986. A brief description of Cedar, and of whether the title begs the question.

Shapiro, E. and A. Takeuchi, "Object-Oriented Programming in Concurrent Prolog," TR-004, ICOT, Tokyo, April 1983. Discusses an interesting approach to integrating object oriented and logic programming, using parallel distributed processing

techniques.

Shapiro, E. Y., "A Subset of Concurrent Prolog and its Interpreter," TR-003, ICOT, Tokyo, February 1983. A description of an interpreter for Concurrent Prolog [Shapiro 83a].

SIoman, A. , The Computer KevoJ.uJii.on An Philosophy: Philosoghy, S£ ^Hd Models of Mind, Harvester, Hassocks, Sussex, 1978.

Smith, R.G., "Strobe: Support for Structured Object Knowledge * Representation," in Proc. IJCAX-83, pp. 855-858, 1983. Describes an interesting object extension to Interlisp, derived from the UNITS package.

Snyder, A., "Object-Oriented Programming for Common Lisp,![1] ATC-85-1 (Rev. 1), Hewlett-Packard Company, Palo Alto, CA, July 1985. Proposed object oriented programming facilities for Common Lisp; describes some interesting alternative views on the operational semantics of inheritance.

Steele, G.L., Common LISP: The Language, Digital Press, Hanover MA, 1984. The main reference on Common LISP.

Stefik, M., D. G. Bobrow, S. Mittal, and L. Conway, "Knowledge Engineering in LOOPS: Report on an Experimental Course," AAAI, pp. 3-13, Fall 1983. Report on a 3-day course teaching knowledge programming in Loops.

Stefik, M. and D.G. Bobrow, "Object-Oriented Programming: Themes and Variations," The A£ Maga^zijie, vol. ??, no. ??, pp. 40-62, January 1986. A highly recommended discussion and survey of basic concepts and recent trends.

Stefik, M.J., "An Examination of a Frame-Structured Representation System," in Procjs of the Sixth International Joint Conference on Art2f icJ[a2 Intelligence, pp. 845-852, August 1979. The reference for the UNITS package: the precursor to Loops, KEE and Strobe.

Stefik, M.J., D.G. Bobrow, and K.M. Kahn, "Integrating Access-Oriented Programming into a Multiparadigm Environment," IEEE Software, vol. ??, no. ??, pp. 10-18, January 1986. Describes how Loops provides access-oriented programming, and how it is used.

Stroustrup, B.t "Data Abstraction in C," Computing Science TR-109, AT&T Bell Labs, Murray Hill, NJ, January 1984. A brief overview of how C++ adds classes, data hiding, etc to C*

Stroustrup, B., A C++ Juj:orij|i, AT&T Bell Laboratories, Murray Hill, NJ, September 1984? Presents examples of data abstraction and encapsulation in C++.

Stroustrup, B., in The C++ *ZL9^L^MiLllZ L<*HZUQ£*L>* Addison-Wesley, Reading MA, 1986. ~The main reference for C++.

Tesler, L., "The Smalltalk Environment,[11] BYTE, vol. 6, no. 8, pp. 90-147, August 1981. Demonstrates how programming and debugging in Smalltalk are inherantly interactive in nature.

Tesler, L., "Object-Oriented User Interfaces and Object-Oriented Languages," 1983 ACM Conference on Personal and Smal1 Computers, vol. 6, no. 2, pp. 3-5, 1983.

Tesier, L., Object Pascal Report, Apple Computer, Inc., February 1985. Describes progress on OOP extensions to Pascal; descended from earlier work on Clascal with design help from Niklaus Wirth.

Tesler, L., "Programming Experiences," BYTE, vol. 11, no. 8, pp. 195-210, McGraw-Hill, August, 1986.

Theriault, D. , "A Primer for the Act-1 Language," AIM--672, MIT AI Laboratory, Cambridge MA, June 1982.

Tyugu, E. H., "NUT - An Object-Oriented Language," Proc. of Third ICAI and information-control l^s^ems of robots, North-Holland, Amsterdam, Netherlands, June 1984.

Ungar, D., R. Blau, P. Foley, D. Samples, and D. Patterson, "Architecture of SOAR: Smalltalk on a RISC," nth Annual Intl. Symposium on Computer Architecture Pr^oceedijig^, no. CAT. NO. 84CH2051-1, pp. 188-197, IEEE Computer Society Press, Silver Spring, MD., USA, June 1984. Implementing Smalltalk on Reduced Instruction Set Computers.

Wall, R.S., "Industrial strength knowledge representation," Third Intl. Ph2£Sl[x] Conf• on Computers and Communications Proceed-rng£, no. CAT. N0~ 84CH2010-7, pp. 6-10, IEEE Computer Society Press, March 1984.

Weinreb, D. and D. Moon, "Objects, Message Passing, and Flavors," in Li.sj2 Machine Manual,, Chapter 20, pp. 401-447, Symbolics, Inc., June 1984. The older reference for the Symbolics Flavors System.

Weinreb, D. and D. Moon, "Introduction to the Flavor System," in Reference Gujjde t£ Symbolics-Lisp , pp. 417-476, Symbolics, Inc., March 1985. The main reference for the Symbolics Flavors System.

Wertz, H. , "bVLISP: An Example of a Programming Environment," ^J2H£H££5 kiiILS2i*£l§ 2£ï£Qi£jS Ob^et, pp. 55-63, AFCET-Informatique, Paris, January 1986.

Wilks, Y., "Good and Bad Arguments about Semantic Primitives," D.A.I. Research Report No. 42, Department of Artificial In-

telligence, University of Edinburgh, Edinburgh, Scotland, May 1977.

Williams, C., ART: Conceptual Overview, Inference Corporation, Los Angeles, CA, 1985.

Winston, P.H. and B.K.P. Horn, "Object-Centered Programming: Message Passing and Flavors," in LISP, pp. 239-251, Addison Wesley, Reading MA, 1984. A somewhat limited but useful introduction to object oriented programming for beginning AI students and programmers.

Wood, R.J., "Franz Flavors: An Implementation of Abstract Data Types in an Applicative Language," TR-1174, Maryland Artificial Intelligence Group, University of Maryland, College Park MD, 1982. A description of a Flavors system based on Franz Lisp [cf, Allen 83].

Yonezawa, A., H. Matsuda, and E. Shibayama, "An Approach to Object Oriented Concurrent Programming: A Language ABCL," Journees Langages Orientes Objet, pp. 125-135, AFCET-Informatique, Paris, January 1986. A framework of parallel computation and a scheme of distributed problem solving; very interesting work based in part on Actor theory.

Zaniolo, C., "Object Oriented Programming in Prolog," in 1984 International Symposium on Logic Programming, pp. 265-270, IEEE, February 1984. A simple approach to providing some object oriented capabilities in Prolog.

Zippel, R., "Capsules," Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems, vol. 18, no. 6, pp. 166-169, June 1983.