I Know. -

23/2/88.

# Knowledge Representation and Inference Two

## Lecture Seven:

## Object-Oriented Programming

## 1 Introductory Comments

Object-oriented programming is one of the more recent and popular programming methodologies. There are now many so called object-oriented languages in use: Smalltalk, Actors, Strobe, LOOPS, Objective-C, C++, Objective Pascal, BLOBS, Common LOOPS, Common Objects to name just a few. Many commercial hardware and software systems now include some object-oriented features, such as Flavors on Symbolics machines, KEE and ART on various kinds of workstation, the GoldWorks package for Golden Common LISP on PC/ATs and so on.

Along with so called Expert Systems, Object-oriented Programming represents one of the contributions AI has made to the world of computer programming and the engineering of computer-based systems.

These notes aim to introduce the basic principles and elements of the object-oriented approach, and to illustrate their use. Some comments relating its development to work in knowledge representation and inference are also made. The sections on XLISP and FLAVORS are based upon a set of notes produced by Peter Ross.

## 2 Some History and Background

### 2.1 The History

The term *object-oriented programming* was first used to describe the Smalltalk programming environment developed at Xerox PARC in the 1970's, though the roots of the object-oriented approach go back to SIMULA [DH72] which was developed in the mid-1960s. It is related to, and influenced by, Minsky's work on *frames* [Min85] and Hewitt's work on *actors* [Hew77]. The first AI systems to incorporate it as a methodology were KRL [BW85] and UNITS [Ste78], both developed in the late 1970s.

During the 1980s object-oriented programming environments for Lisp machines were developed: Flavors at MIT, and LOOPS at Xerox PARC. It has also been included as a methodology supported by a number of AI Toolkits: KEE, ART, and CRL.

### 2.2 The Background

The background to the development of object-oriented programming can be briefly (and thus by no means completely) presented by considering the motivations behind Bobrow's



and Winograd's KRL (Knowledge Representation Language). These were:

- an attempt to integrate *procedural* and *declarative* knowledge representation formalisms,
- to use formalisms based upon *structured objects* with associated *descriptions* and attached procedures,
- to use procedural attachment to provide generic *operations* which depend upon the characteristics of the objects involved,
- that knowledge should be organized around conceptual entities with associated descriptions and procedures,
- that entities should be able to be partially described, and have multiple descriptors to capture different points of view of the same entity,
- to support reasoning in terms of comparisons between new objects and stored prototypes, with specialised reasoning strategies associated with prototypes,
- to support multiple active processes and appropriate control mechanisms,
- to provide a flexible set of basic tools which embody no commitment to specific representation and reasoning techniques.

KRL was therefore revolutionary, in that:

- the structuring of knowledge was object-centred,
- objects could exist at different levels of control in different perspectives,
- it was agnostic about epistemological issues and aimed to provide implementation tools.

## 3 Principles of Object-Oriented Programming

Most programming languages support the *data procedure* paradigm. Active procedures act on the passive data passed to them. For example, a square root function, `sqrt(x)`, takes a number and returns its square root.

In a strongly typed language such as Pascal, it would be typical to have a different version of `sqrt(x)` for each data type of x, usually returning a floating-point result. A *late-binding* language such as LISP detects x's type at run time and performs the appropriate operations for that type. Such generic operations are generally primitives restricted to a small class of data types such as numbers, or they are functions defined in terms of such primitives.

Object-oriented languages employ a data or *object-centered* approach to programming. Instead of passing data to procedures, objects perform operations on themselves, and each other. In the following examples the object name is followed by `:operation`, which is in turn followed by any further arguments, and terminated by a period (this

syntax is similar to that of Smalltalk-80, though simplified). For example, an expression to take the square root of x has the form:

```
x :sqrt.
```

The implication is that x is asked to perform the :sqrt operation on itself. We say that x is the *receiver* of the *message* :sqrt.

A more complicated example would be the dot-product operation. The dot-product of two vectors, x and y, is computed, producing a scalar result:

```
x :dot y.
```

Here, x is told to perform a dot-product operation with itself and the argument y. We could thus take the square root of the dot product of x and y and assign the result to the variable z in the following manner

```
z <-- (x :dot y) :sqrt.
```

using parentheses to indicate the order of evaluation, although the parentheses are really not needed here, assuming left-to-right evaluation.

### 3.1 Some Terminology

The object "x" referred to in the sqrt example is an *instance* of a *class*. This distinction between classes and instances in an important one in knowledge representation in general. The class provides all the information necessary to construct and use objects of a particular kind, its instances. Each instance has one class, and a class may have many instances.

The class also defines *methods* which apply to all instances of it. Methods are procedures invoked by sending *selectors*, or *messages*, to a class's instances. Methods may allocate temporary variables for use during the execution of the method. These temporary variables are like local variables in Pascal procedures in that their value is lost when you leave the method.

Each instance has storage allocated for maintaining its individual state. The state is referenced by *instance variables*. Instance variables may be primitive data types such as integers, other objects, or both, depending on the language. Each object has its own set of instance variables. Both temporary and instance variables may be freely referenced within the scope of an object's method, but unlike temporary variables the value of instance variables is not lost when you leave the object's method. Some object-oriented programming systems also provide *class variables*. These are used to store data about a particular class and their values are available to all instances of the class.

Computation is performed by sending messages to objects, which invokes a method defined in the object's class. Typically, a method sends messages to other objects, which invokes other methods, etc., until you reach the point where a *primitive method* is invoked. Here ends the chain of message-sends. Each message-send eventually returns a result to the sender (e.g., x :sqrt returns the square root of x). The final result of



all these message-sends is usually the changing of the state of one or more objects. Sometimes, however, a message is sent simply to invoke some primitive having a side effect external to the world of objects, for example, accessing an external file system or controlling hardware.

### 3.2 The Elements of Object-Oriented Programming

To fully support object-oriented programming a language must have four characteristics:

- information hiding,
- data abstraction,
- dynamic binding,
- inheritance.

Two languages, Ada and Modula-2, that have been mistakenly called object-oriented, will be used in order to illustrate why all four characteristics are necessary and why conventional procedure-oriented languages cannot adequately support object-oriented programming.

#### 3.2.1 Information Hiding

Information hiding is important for ensuring reliability and modifiability of software systems by reducing interdependencies between software components. The state of a software module is contained in private variables, visible only from within the scope of the module. Only a localized set of procedures directly manipulates the data. In addition, since the internal state variables of a module are not directly accessed from without, a carefully designed module interface may permit the internal data structures and procedures to be changed without affecting the implementation of other software modules.

Most modern languages, even FORTRAN, to some degree, support information hiding. ISO (standard) Pascal is one notable exception, since it provides no way to declare static variables within the scope of a procedure.

#### 3.2.2 Data Abstraction

Data abstraction can be thought of as a way of using information hiding. A programmer defines an abstract data type consisting of an internal representation plus a set of procedures used to access and manipulate the data. Modula-2 provides excellent data abstraction mechanisms. For example, it can easily be used to define a *stack* as an abstract data type, called Stack. Variables of type Stack may be declared and manipulated in other program units.

Modula-2's data abstraction mechanism provides a certain degree of protection since no direct access to the internal state of a stack is provided. The stack is manipulated through the module's processing and query procedures. But there are two problems with

the Modula-2 solution. First, the procedures used by a module must have either unique or qualified names. For example, if a module uses (imports) two different abstract data types, `Stack` and `Queue`, and variables of these types must be initialized, then the initialization procedures defined for these types must have different names, such as `InitializeStack` and `InitializeQueue`. This makes the resulting program less versatile. Secondly, and more importantly, Modula-2 abstract data types can operate on only one type of data. `Stack`, therefore, can store only integers, for example.

Ada partially solves both these problems through two language features: operator overloading and generic program units. Operator overloading permits a program to use multiple operators with the same name. The distinction between operators can be determined at compile time by examining the types and number of parameters, just as + can be used to add either integers or real numbers in most modern languages. Generic program units permit the definition of a module to be used with different data types. The generic program unit is a procedural template that can be parameterized with actual types during compilation of programs using its capabilities.

A problem still exists if you wish to use the stack to store heterogeneous elements. Neither compile-time solution, operator overloading or generic program units, is sufficient. A solution is dynamic binding.

### 3.2.3 Dynamic Binding

Dynamic binding is required to make flexible use of the `Stack` module. Consider the addition of a procedure, `Print`, to the Stack-Handler module that prints the contents of a stack. If we use the stack for storing integers, floating-point numbers, character strings, etc., a traditional procedure-oriented approach dictates that you include a case statement to check at run time that the correct printing procedure for an element's type is used. Trying to print an integer with a procedure designed to print character strings is potentially disastrous. The resulting problem is that every time you add a new data type to the system, you must modify all such case statements and recompile – a time-consuming and error-prone procedure. Ideally, additions should require only additions, not modifications.

The object-oriented approach pushes the responsibility for printing elements onto the objects themselves. Each object is sent the same message selector, `Print`, so that it will print itself in the proper way. This is known as *polymorphism*, since the same message can elicit a different response depending on the receiver. Operator overloading in Ada does not exhibit this form of dynamic polymorphism since the address of the procedure invoked is fixed at compile time.

This model of object-oriented programming can be improved. As presented thus far, the addition of a new type of object requires writing entirely new procedures for common operations such as `Print`. What's worse is that there will be a great deal of similarity between different print methods, requiring continual rewrites of methods that differ slightly or not at all. This burden is likely to be so great that programmers would avoid the creation of new object types, significantly reducing the practical usefulness of object-oriented programming systems. Inheritance is a mechanism that largely relieves programmers of this burden.



### 3.2.4 Inheritance

Inheritance enables programmers to create classes and, therefore, objects that are specializations of other objects. For example, you might create an object, `Trumpet`, that is a specialization of a `BrassInstrument`, which is a specialization of a `WindInstrument`, etc. A trumpet inherits properties that are associated with brass instruments, wind instruments, and musical instruments. Creating a specialization of an existing class is called *subclassing*. The new class is a *subclass* of the existing class, and the existing class is the *superclass* of the new class. The subclass inherits instance variables, class variables, and methods from its superclass. The subclass may add instance variables, class variables, and methods that are appropriate to more specialized objects. In addition, a subclass may override or provide additional behavior to methods of a superclass. Inherited methods are overridden when redefined in a subclass.

Inheritance enables programmers to create new classes of objects by specifying the differences between a new class and an existing class instead of starting from scratch each time. A large amount of code can be reused in this way.

## 3.3 A Prototypical Object-Oriented System

In a prototypical object-oriented system the basic elements described above are implemented in terms of the following features:

**Object** — This is the basic unit, a data structure with memory allocated for a set of attributes; these are called the object's *instance variables*. There will be a large number of objects in a typical application. Objects are organised into groups of similar objects. An object has access, in a way described below, to a set of procedures that define the operations that can be done on the object. The procedures that the object has access to are not stored in its own data structure; nevertheless, those procedures typically use the object's instance variables as data.

**Class** — A related group of objects is called a *class*. A class is also a data structure, and an object in its own right. When a class is defined, a set of instance variable names is specified to go with it. Any object which is a member of that class will have its own set of instance variables, but known by those names. This means that, given any instance variable name, there will be many variables known by that one name – one variable per object. A set of procedures is typically attached to a class, and these are some of the procedures that the objects in the class have access to.

**Inheritance** — The procedures attached to its class are not the only procedures that an object has access to, because classes are arranged in a hierarchy, by a superclass/subclass relationship. Any object in a class has access to the procedures stored with the class, and to the procedures stored with the superclass, and the superclass of that, and so on. Typically, the top-most class in the hierarchy is defined to be its own superclass. One says that an object *inherits* procedures from all the classes on the path up to the top of the hierarchy.

**Instance** — If an object is a member of a class it is said to be an *instance* of that class. It is *not*, however, an instance of the class's superclass or of other classes

up the hierarchy. An object is an instance of exactly one class. In some systems the word *instance* is used only for objects which are not themselves classes. In such systems, it is customary to talk of one class having another as its *superclass*, rather than saying it is an instance of that other class.

**Message** — An operation on some object, such as requesting the value of one of its instance variables or some more elaborate result, is performed by *sending a message* to the object. This is really just a certain kind of procedure call. The message consists of a *selector*, which is used to identify the procedure to be used, and arguments to be passed to the procedure wherever it is to be found. When an object receives a message, it looks for a suitable procedure accessible to it. First it searches the procedures defined for its class; if no procedure for that selector is attached to the class, the object then searches the superclass. The search proceeds up the class hierarchy until a procedure is found. It is typically an error for there to be no suitable procedure anywhere up the hierarchy. When a procedure is found, it is run. The procedure may utilise various kinds of data. First, it has access to the object's instance variables. Second, in some systems, there are also class variables available. These are variables which are part of a class's structure rather than being part of any member of the class. The procedure has access to the variables of the class of the object, and (by inheritance again) to class variables of any classes further up the hierarchy.

**Method** The procedures attached to any class are called its *methods*.

So, the essence of what goes on in an object-oriented programming system is the passing of messages. When an object receives a message, the following happens:

- there is a search for the appropriate method. This starts in the object's class (not in the object, even if the object is a class itself), and if necessary proceeds to the object's superclass and any super-super-...-classes until one is found.
- any references to variables in the procedure are tracked down. First, the object's instance variables are searched, then any class variables of the object's class, then the super-class of that, and so on. This is just a caricature, of course; in practice there might be no searching at all, because all the variable references have been resolved in some way at compile time.
- the procedure may cause more messages to be sent, sometimes to other objects and sometimes even to the same object. This leads to a wonderful economy of programming style, usually at little expense in execution speed. Normally, in a LISP-based object-oriented programming system, the symbol `self` is automatically bound to the object that received the message, so that your procedure code can then simply send messages to `self`, to identify which object is using the procedure or whatever.
- the result of executing the method is the result of the message.



# 4 Two Object-Oriented Systems

## 4.1 XLISP

XLISP is a toy Lisp system for experimenting with object-oriented programming. It was written by David Betz of Harvard University. Its main strengths are that it is free, available as source code, written entirely in C, runs on many different machines, is easy to extend and is reasonably efficient for an interpreted system. It most resembles the core of Common LISP. Its main features are the existence of class variables as well as instance variables, and a simple tree-structured class hierarchy as outlined above, rather than a class lattice. In other words, it does not allow multiple inheritance!

In XLISP, all the arguments of a message are evaluated, although the system follows the usual convention that keywords (symbols whose names begin with a colon) by default evaluate to their name. This makes it convenient to use such symbols as message selectors. The form of a message is

(*object-symbol message-selector arguments*)

There are initially two objects in existence. One is `Object`, the top of the whole class hierarchy. There are four predefined methods attached to it:

**:show** print out details of the instance variables of the receiving object (which will hardly ever be `Object` itself). This is mainly used for debugging purposes; other `:show` methods will usually be defined elsewhere in the hierarchy to do something more suitable for the specific application.

**:class** returns the class of the object which receives this message.

**:isnew** is a vacuous, default object initialisation method. It exists only to plug a gap (see `Class` below).

**:sendsuper selector arguments**
is useful for sending a message to the super-class of a class (that is, to the class one step up in the class hierarchy).

The object named `Object` is an instance of the class named `Class`. `Class` is the other predefined entity; it is a peculiar beast, which is an instance of itself and which has `Object` as its super-class. There are three predefined methods attached to it:

**:new** creates a new object which is an instance of the class which received this message. There may be further arguments given. By default, an `:isnew` message is sent to the new object, with the arguments (if any) given to the `:new` message. This is why a vacuous `:isnew` method has to exist somewhere, otherwise this default process would cause an error if the programmer had not yet defined his own version. The `:new` message can also be sent to `Class` itself, this creates a new class. By default the new class is a sub-class of `Object`, but an optional parameter naming some other class can be given, and then it will be a sub-class of that instead.

**:isnew ilist [clist [super]]**

This initialises a new class. The `ilist` is a list of instance variable names to be associated with the class – that is, a set of these variables is created for every instance of the class. The `clist` is an optional list of class variable names. The `super` is an optional parameter specifying the super-class of the new class – that is, where it is to be hung in the hierarchy.

**:answer msg arglist formlist**

This adds a new message to the receiving class. It is an error to send it to anything which is not a class. The `msg` is the message selector symbol. The `arglist` is a list of the formal arguments to the message; the usual keywords `&optional`, `&rest` and `&aux` are allowed and have their normal roles. The `formlist` is a list of executable expressions, the code comprising the body of the method. The result of the last expression will be the result of the message.

An example should make this all clearer. Here is a standard example used in many texts on object-oriented programming systems:

```
(setq Ship (Class :new '(x y xv yv mass name captain)))
```

This creates a new class called `Ship`; the list of instance variable names is not for use by the `:new` method itself, it is passed to the automatic `:isnew` message (see above). So, each instance will have a cartesian position and cartesian components of velocity, a mass, a ship's name and the name of its captain. Now we need messages to access this information about any given ship, and it makes sense to attach such methods to the class itself:

```
(Ship :answer :getx       '() '( x ))
(Ship :answer :getxv      '() '( xv ))
(Ship :answer :gety       '() '( y ))
(Ship :answer :getyv      '() '( yv ))
(Ship :answer :getmass    '() '( mass ))
(Ship :answer :getname    '() '( name ))
(Ship :answer :getcaptain '() '( captain ))
```

In each case the body of the method is trivial – just evaluate an instance variable, return its value. The following is a slightly more elaborate method; it takes one argument, a number of hours, and causes the information about a ship's position to be updated:

```
(Ship :answer :sail '(time)
  '( (self :setx
       (+ (self :getx) (* (self :getxv) time))
     )
     (self :sety
       (+ (self :gety) (* (self :getyv) time))
     )
   )
)
```

V. NICE!?!



This shows one of the common idioms, namely using messages as convenient subroutines. In this case the messages are sent to the ship itself, from the ship itself. Before the `:sail` method can be used, the `:setx` and `:sety` methods must be defined, of course:

```
(Ship :answer :setx       '(newx)     '( (setq x newx) ))
(Ship :answer :setxv      '(newxv)    '( (setq xv newxv) ))
(Ship :answer :sety       '(newy)     '( (setq y newy) ))
(Ship :answer :setyv      '(newyv)    '( (setq yv newyv) ))
(Ship :answer :setmass    '(newmass)  '( (setq mass newmass) ))
(Ship :answer :setname    '(newname)  '( (setq name newname) ))
(Ship :answer :setcaptain '(c)        '( (setq captain c) ))
```

It is, however, mildly inefficient to define the `:sail` method as above. It is better to do it in a more conventional way, and also to return something more usable than the 'y' component of velocity:

```
(Ship :answer :sail '(time)
  '( (setq x (+ x (* xv time)))
     (setq y (+ y (* yv time)))
     time   ; return the given sailing time
   )
)
```

It also makes sense to define a function which creates new ships and initialises them at the same time:

```
(defun newship (name captain mass &aux v)
    (setq v (ship :new))  ; set v to be the new ship
    (v :setx 0)   ; Now initialise it by sending messages
    (v :sety 0)
    (v :setxv 0)
    (v :setyv 0)
    (v :setname name)
    (v :setcaptain captain)
    (v :setmass mass)
    v             ; and finally return the object
)

; For instance:

(setq argo (newship 'Argo 'Jason 2307))
```

In XLISP, you need to create the methods for initialising instance variables and other such commonplace jobs. You could of course define your own interface to the `(Class :new ...)` message which did this for you automatically, and more elaborate object-oriented programming systems provide such things by default.

## 4.2 Flavors

The *Flavor* mechanism is part of ZetaLISP, which runs on Symbolics machines; the mechanism has been replicated in various other LISPs. It is described in the "Lisp Machine Manual" by Weinreb and Moon [1]. A more superficial description can be found in "LISP Lore: A Guide To Programming The Lisp Machine", by Hank Bromley (Morgan Kaufman).

The Flavor system is less purist about the object-oriented approach; you can poke inside objects using lower-level LISP functions, for example, although you shouldn't. It is much more powerful than XLISP. For example, inheritance can be across a true lattice, not just up a tree; this can, of course, result in very exciting bugs.

Here is the equivalent of the ship example:

```
(defflavor Ship (x xv y yv mass name captain) ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables
)
```

The keyword `:gettable-instance-variables` causes methods such as `:x` to be defined automatically; these return the value of the named instance variable. The keyword `:settable-instance-variables` requests the automatic definition of other methods such as `:set-x`, for setting the values. The keyword `:initable-instance-variables` creates keywords for use in the instance creation routine:

```
(setq argo
  (make-instance 'Ship
     :x 0 :xv 0 :y 0 :yv 0
     :name 'Argo
     :captain 'Jason
     :mass 2307)
)
```

DEFSTRUCT?

Methods are added like this:

```
(defmethod (Ship :sail) (time)
  (setq x (+ x (* xv time)))
  ... etc ...
)
```

and invoked by `send`, which is merely `funcall` in disguise:

```
(send argo :sail 7.5)
```

There are about a million special features available. One which is worth further discussion here is that extra argument to `defflavor`, just after the list of instance

[1] often called the 'Chine-ual' because of the way the name wraps around the cover



variable names. In the example above it was an empty list. More generally it is a list of other flavors (the *component flavors* of the composite, as it were), which instances of this flavor can search during the inheritance process. An object can not only search those flavors for a method or instance variable, but also all the flavors which they can inherit from too; the search is a straightforward depth-first search of the network of flavors. This listing of the flavors that instances of a flavor can search is, naturally enough, called *mixing flavors*. The customary ordering is to have the most specific flavors at the front of the list and the most general at the back. Usually a fundamental definition is called a *base flavor*, and flavors that just add features are called *mixins*.

Since the search of the network of flavors is purely depth-first, it sometimes happens that a component flavor of some other component flavor appears too soon in the search order – that is, a method appears there but it is not the method you wanted to be found first. There are several mechanisms to help you. For example, there are the `:required-flavor` or the `:included-flavor` options, which can appear as a separate list starting with the keyword and followed by one or more flavors. These options specify that the flavors they designate are not to be searched at that point, or just at some point. The `:included-flavor` form specifies that a flavor it designates must be added in last of all if it happens not to appear anywhere earlier in the search. The other form merely makes it be an error for that flavor not to appear anywhere in the search.

You can also define *before* and *after* daemons, like this:

```
(defflavor WarShip (country guns) (Ship)
  :gettable-instance-variables
  :initable-instance-variables
)
; Note: instance variables not settable - we suppose
; that navies do not sell their ships or change the
; number of guns on a ship!

(defmethod (Warship :before :sail)
  (confirm-sailing-orders)
)

(defmethod (WarShip :after :sail)
  (report-position-to-base)
)
```

By this means, warships do these extra actions whenever they receive a `:sail` message. Obviously, it would make no particular sense to attach these daemons to the same flavor that holds the method for `:sail`. Return values of such daemons are just ignored.

Sometimes, such daemons do not provide enough control; for example, you might want to cause the combined method to run in a special context (say, an `unwind-protect` or with some special variable temporarily given another value?). For such occasions the Flavors package provides *whoppers*.

For example:

```
(defwhopper (some-flavor :some-method) (arg1 arg2)
  (unwind-protect
      (progn (setup) (continue-whopper arg1 arg2))
      (cleanup)
  )
)
```

The job of the system function `continue-whopper` is just to call the normal (combined) method.

So far, it has been assumed that the top-down depth-first search for a method ends when the first appropriately-named method is found. Even this can be changed, by the `:method-combination` option! For instance, you can specify that the appropriately-named methods should be tried in turn, until one returns non-NIL. Consider the example of handling mouse button clicks. On a Symbolics, when you press a mouse button, the message `:mouse-click` is sent to the window under the mouse cursor, by the mouse handler. If you want that window to do something special, you can just define a `:mouse-click` method for it. One of the message arguments indicates what kind of button press it was (e.g. a double click), and your special method can just deal with the cases you want. However, one of the component flavours for all windows is `essential-mouse`:

```
(defflavor essential-mouse () ()
  (:included-flavors essential-window)
  (:method-combination
      (:or :base-flavor-last :mouse-click))
)

(defmethod (essential-mouse :mouse-click) (buttons x y)
  ... the code, default actions for all cases ...
  t
)
```

The effect of the `:or` keyword is to cause every `:mouse-click` method found in the search to be tried, until one returns non-NIL. The effect of the `:base-flavor-last` keyword is to cause `essential-mouse`'s `:mouse-click` method to be tried only at the very end of the search. There are several other ways of combining methods.

As all this should suggest to you, OOP systems such as Flavors offer a very powerful and economical way of programming, particularly suited to jobs such as process simulations and sophisticated interfaces in which there is a natural metaphor of a community of objects each with its own internal state and its own processing power.



# 5 Some Comments

## 5.1 Advantages

Object-oriented languages have many advantages over more traditional procedure-oriented languages. Information hiding and data abstraction increase reliability and help decouple procedural and representational specification from implementation. Dynamic binding increases flexibility by permitting the addition of new classes of objects (data types) without having to modify existing code. Inheritance coupled with dynamic binding permits code to be reused. This has the attendant advantage of reducing overall code size and increasing programmer productivity, since less original code has to be written, tested and debugged. Inheritance enhances code *factoring* Code factoring means that code to perform a particular task is found in only one place, and this eases the task of software maintenance.

## 5.2 Criticisms Defended

Object-oriented languages have a few characteristics that are considered disadvantages by some. The one most often debated is the run-time cost of the dynamic binding mechanism. A message-send takes more time than a straight function call. Some studies have shown that with a well-implemented messager this overhead is approximately 1.75 times a standard function call. Actual differences in execution speed between traditional languages and their object-oriented counterparts, however, do not prove to be very significant. This is most likely due to the fact that the overhead applies only to message-sends and that message-sends accomplish more than a function call. Often, some of the work done automatically by a message-send must be done by the programmer anyway using code surrounding function calls or even multiple function calls. In fact, a case can be made that in large applications the ability to standardize and fine-tune the functionality supplied with the message-sends can make the application run faster than a traditional counterpart. The primary reason is that messaging obviates much of the variability in function setup code that results from different programming styles and skill levels. Messaging also eliminates the complex code often needed when traditional programs have to simulate dynamic binding.

Another disadvantage often cited is that implementation of object-oriented languages is more complex than comparable procedure-oriented languages, since the semantic gap between these languages and typical hardware machines is greater. Therefore more software simulation is required. Fortunately, you pay the cost of implementation only once for a given machine.

Another potential problem is that a programmer must learn an often extensive class library before becoming proficient in an object-oriented language. As a result, object-oriented languages are more dependent on good documentation and development tools such as Smalltalk-80 browsers.

### 5.3 Problems Of Using Object-Oriented Programming

Problems can arise in using the more "powerful" object-oriented programming systems if an undisciplined approach is taken to the *overiding*, or *denial*, of default properties, and the use of multiple inheritance. Such practices make it effectively impossible to define anything.

Not all problems are easily programmed in terms of objects and message passing. For example, some numerical analysis and simulation problems are not well suited to an object-centred approach.

Two common problems arise in object-oriented programming. An *epistemological* problem about what do objects stand for, and an *implementational* problem about grain size, in other words, the structuring of the computation in terms of objects.

### 5.4 Objects, Production Rules and Logic Programming

Sets of production rules and Horne clauses used in logic programming lack explicit structure. Structured objects make it easier to encode other transitive relations apart from type-subtype, for example, part-whole, and cause-effect. Production rules are good for encoding direct stimulus-response patterns in an ill-structured domain. Whereas logic programming is good for describing complex relationships such as those involving quantifiers and n-ary predicates. Objects can be combined with rules and logic programming, as in the KEE system for example, and can act as a layer between these formalisms and an underlying implementation language like LISP.

### 5.5 System Engineering Level Again

Object-oriented programming can be seen as a powerful technique to be used to engineer knowledge-based systems, and other AI-based systems. Its effectiveness derives from its well structured yet flexible delivery of a means to effectively build compact easily testable and modifiable computer programs to implement the Symbol Level techniques selected to deliver the Knowledge Level functional requirements.

## 6 How To Get At XLISP

XLISP is available on edai, aipna, and itspna. To run XLISP login under your own user id and type `~tim/xlispThings/xlsipEx/xlisp.unix`. You should then see something like this:

```
edai% ~tim/xlispThings/xlispEx/xlisp.unix
XLISP version 1.6, Copyright (c) 1985, by David Betz
No Insert capability
>
```



You can find documentation in `~tim/xlispThings/xlispDoc/xlisp.doc`, and some example programs in the directory `~tim/xlispThings/xlispProgs`. A screen-based object-oriented turtle system is available in `~tim/xlispThings/xlispTurtle`, but you should `set term = VT100` if you don't normally do so.

## References


[BW85] D. G. Bobrow and Terry Winograd. An overview of KRL, a knowledge representation language. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, chapter 13, page 263, Morgan Kaufmann, 1985. Also appears in *Cognitive Science*, Vol. 1, 1977.

[DH72] O. J. Dahl and C. A. R. Hoare. Hierarchical program structures. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, Academic Press, New York, 1972.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3), 1977.

[Min85] Marvin Minsky. A framework for representing knowledge. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, chapter 12, page 245, Morgan Kaufmann, 1985. Also appears in *Mind Design*, J. Haugeland, Editor, MIT Press, 1981.

[Ste78] M. Stefik. *An Examination of a Frame-Structured Representation System.* Report HPP-78-13, Stanford University, Heuristic Programming Project, Computer Science Department, 1978.


## 7 Required Reading

The required reading for this week is Chapter 13 of the Big Red KR Book: Bobrow, D.G. and Winograd, T., *An Overview of KRL, a Knowledge Representation Language*, page 263.

## 8 Small Group Tutorial Preparation

Object-oriented programming is presented as a System Engineering Level technique. Could you use it to engineer a production rule system which has either a forward chaining interpreter, or a backward chaining interpreter, or an interpreter which mixes the two types of inferencing? If you think the object-oriented programming paradigm is suitable then you should prepare an argument in its defense. If you think it is not, you should prepare an argument as to why it is not. If you want to sit on the fence you should prepare an explanation as to why you can't decide and what you would need to do to find out.

Tim Smithers
February 1988