# Object-Oriented Techniques in AI

## P. Ross

**Rapporteur:** **Maciej Koutny**

# Object-oriented techniques in AI

Peter Ross*
University of Edinburgh
Department of AI
80 South Bridge
Edinburgh EH1 1HN

September 1988

This brief paper summarises my views about object-oriented techniques in AI, and how this affects my views about the teaching of AI. Please bear in mind that at Edinburgh University there are separate departments for computer science and for AI, even if they do have many common interests.

## The context: about AI

First, there are some points to be made about the teaching of AI generally. There are perhaps three kinds of answer to the question "What is AI?":

- (the commercial answer, widespread) a new way to make more money faster than ever before; an interesting sub-culture in software engineering.
- (the cognitive psychological view, rare in its pure form) an approach to the study of human cognitive capacities, by trying to model them in some executable form. Clearly this depends on a good understanding of cognitive psychology. This answer is dogged by the possibility that there are no general cognitive principles underlying the functioning of human cognition, that the human brain is merely a monumental evolutionary hack-up. It is, however, unconstructive to adopt this possibility as a working premise.
- (the control theory answer, also rare) an investigation into the construction of artificial intelligence, perhaps with little or no direct parallels in terms of human intelligence. A good grounding in cognitive psychology is not required, but is still useful. This view treats AI as essentially an enormous

*e-mail: peter@uk.ac.edinburgh.aiva

> problem of control theory, in the broadest sense: how to make a system which is adaptive to a very wide range of circumstances, and which exhibits many desirable characteristics such as a sizeable but not infinite resistance to performance changes.

It seems important to try to keep all three in mind when teaching AI. However, it is very hard to train students in computer science, cognitive psychology and mathematics, and several other disciplines, and still expect them to be able to bring it all together as AI; therefore some blurring of the distinctions is inevitable.

The main paradigm used in AI is still that of symbol processing, although connectionism is attracting a lot of attention now too. The symbol processing approach rests on assumptions that are rarely discussed. One is that (in cognitive psychological terms) the amount of information which comes in through a person's senses is so vast that it seems self-evident that it is digested somehow into a very much more compact form, with massive loss of detail, before any of the mechanisms of consciousness get to work on it. If this is so then presumably we can work at the fairly abstract level of the digested data, trying to model it by data structures built using a meaningless vocabulary of symbols. It is easy to disagree with this, naturally; for example, can we safely assume that the basic vocabulary is 'meaningless', and is it reasonable to assume that the condensation of sensory data happens 'first of all'? (No). Object-oriented techniques in AI are still seen as part of this symbol processing paradigm, although they needn't be.

Because AI brings together many disparate subjects, it also seems useful to emphasise at least three different levels from which one can view any piece of work in AI (loosely based on the original categorisation in [Newell 81]):

- the *knowledge level*, concerned with what knowledge - as an abstract quantity - is represented within a system.
- the *symbol level*, concerned with implementational issues such as data structures and algorithms.
- the *engineering level*, concerned with design issues and bridging the gap between the other two levels.

It is all too easy to become engrossed in the last two of these and to hope that the first is taking care of itself. By custom, object-oriented techniques are presented as issues to do with the symbol and engineering levels. A short and very informal survey I made suggested that object-oriented techniques were not widely regarded as being central to AI's current research concerns.

The main theoretical bases of AI are knowledge representation, search and inference. Knowledge representation is not just a matter of databases, the larger

questions concern the nature, consistency and completeness of the represented information, and currently rest mainly on inference techniques. To hint at this distinction, the typical database approach to answering a question such as "how many students take Maths 301, Geology 429 and are mature students?" is some well-optimised enumeration technique. However, such technology is of no use at all in the following well-known scenario:

> A number of people in a room shake hands. Maybe some people don't shake hands with some others, and maybe some people are so antisocial that they don't shake hands at all. Nothing is known about the number of people in the room, or about who does or does not shake hands with whom.

Nevertheless, despite this minimal level of information, it is possible to say that the number of people who shook the hand of an odd number of people is even. Enumeration methods could offer empirical confirmation, but the general result is only available by some kind of deduction. The result is somehow 'knowable', given the scenario and common knowledge of arithmetic, even if the average man on the Clapham omnibus doesn't actually know it. The distinction between database technology and knowledge representation is further elaborated in [Levesque and Brachman 85]. That paper also points out some of the very serious limitations of the current technology.

All the philosophical difficulties surrounding knowledge and knowing still remain; a university course on 'knowledge representation' is, of necessity, a mixture of practical techniques for constructing certain limited kinds of system and an abstract charting of how little can be done with those techniques. Currently at the research frontier there is a strong emphasis on logics, starting with traditional first-order predicate logic but now embracing many kinds of modal logics and non-monotonic logics as well. There is still argument about the merits of logics as a basis for representing knowledge; see, for example, *Computational Intelligence* vol 3 no.4, 1987, for a recanting by Drew McDermott of his 'strong logicist' stance and commentary on it by many others. Although the use of many kinds of logic for representation raise the problem of logical omniscience - that such an agent would be compelled to believe all the consequences of what he knows - there are some logics which manage to avoid this and preserve a plausible semantics. See, for instance, [Hadley 88] for a brief account. Such work is heavily formalised, contrary to the popular view of AI as a hacker's paradise.

The practical expression of AI has tended to be much more informal, even though there are now a number of expensive commercial tools on the market such as ART, KEE and KnowledgeCraft. These offer a mixed bag of representational methods within a sophisticated programming environment, but tend to offer no

guidance about how to blend the methods to capture what you want to express. The mainstay of the declarative data structuring in these commercial systems, as in many research systems, has been the 'frame'. The notion of 'frames' was introduced by Minsky in [Minsky 75], and has been considerably modified since then. The basic notion is of a named collection of named fields ('slots'), each field having several subfields ('facets') with system-dependent contents and interpretations, arranged in an inheritance hierarchy of some kind. Typically, some facets will contain procedures to be run in predefined circumstances. The whole notion is, alas, drastically underspecified so that, for instance, some commercial systems will allow arbitrary over-riding in some frame deep in the hierarchy of anything inherited from above. This makes it impossible to distinguish between properties of a thing which are part of its defining characteristics and properties which are distinguishing ones. Thus, in such a system, there would be no way to express a proposition such as "All green vegetables are vegetables and not something else such as meat" and it would be possible for a programmer to express a proposition such as "a table is a kind of small, dead, flat-topped elephant without a trunk, ears, tail or tusks", possibly even unintentionally!

There is no consensus about what is and is not desirable in a frame system, indeed it is arguable and intuitively natural that no consensus would be universally adequate. At least, one can say that inheritance matters since the taxonomic aspects of knowledge representation are so prominent. Moreover, multiple inheritance is important; for example, in a software company there are times when it is important to regard an instance of 'project manager' as a human being and times when it is important to regard that same instance as a company resource instead. Also, in constructing a system, inheritance cannot all be resolved away at compile time. One might, for example, find a use for a learning system capable of discovering new links in an inheritance network, perhaps by abduction or induction rather than deduction.

From frame systems it is obviously a very short step to object-oriented systems, and so object-oriented programming has a natural place in an AI syllabus. But it is not the last word in AI programming, nothing is. For example, propositions involving quantification or negation are easily handled by some logic programming methods, and not so well handled by many object-oriented systems. Some systems try to blend these. LOOPS [Bobrow and Stefik 83], for example, is a synthesis of object-oriented, procedure-oriented and data-oriented programming, in and on top of a LISP system. The earliest release of LOOPS included rule-based programming too, but this feature was dropped because it never integrated well with the other styles. There are also logic-based languages which are based on object-oriented ideas, such as the Japanese MANDALA language and programming environment [Furukawa et al 84]. Some information about each of these is

given below.

## About object-oriented techniques in AI

A scan through the proceedings of the main AI conferences will show you that object-oriented techniques are very popular in AI. Why? The commonest reason is that object-oriented techniques help to solve some of the incidental software engineering problems, such as type overloading and the management of late binding. I don't need to repeat the arguments in favour of this; object-oriented techniques clearly help to take a considerable load off the programmer, and specific concepts such as *active values* make execution monitoring and debugging much easier. The term 'active value' perhaps needs explaining: it refers to an arrangement whereby some chosen procedures are run before and/or after every occasion on which some data item is accessed or modified. The procedures can be specific to a single data item. For example, a procedure might modify some graphic representation of the item's value, so that there appears to be some kind of meter attached to the item. However, it is clear that this kind of provision is rudimentary when it comes to trying to debug a really complicated system; not much real progress seems to have been made yet in building usable debugging aids for object-oriented systems.

Perhaps the second most popular reason is that many of the systems which one might wish to model in AI include some form of hierarchic or lattice structuring, and object-oriented systems provide a very neat way to represent this explicitly. Also, the ability to model a collection of entities, each with some local state and having a local program, is very useful. There is a growing community researching multi-agent architectures, in which each of the agents has some information about itself and can acquire some about the others, and the agents need to achieve co-operation without the intervention of some overall controller. For a good introduction to this area, with a formal basis, see [Rosenchein 85]. Object-oriented techniques are obviously useful here for creating simulations for experimental purposes.

There is also interest in the use of object-oriented languages for creating expert systems. This area has been dominated by rule-based systems, but there are considerable problems attached to using rules alone. For example, they suffer from the same kind of inadequacy ascribed to crude frame systems above: some of the preconditions in a rule exist to set a context, others exist to discriminate within that context, and there is no clear division between these (as well as considerable redundancy). Clearly such discriminating information is a necessity if good explanations are to be constructed of how the system reaches its conclusions. See [Buchanan and Shortliffe 84] for the definitive analysis. The notion of objects is useful here, because an object can represent a context, thus avoiding

the redundancy of respecifying the choice of context in each rule of a rule-based system. The general idea would be something like this (in a medical diagnostic system):

```
Send a message to the Patient object, requesting symptoms.
Send the list to a Diagnostic object, requesting a
    priority ordering of symptoms
Send messages to Symptom objects, requesting possible
    causes in the form of names of disease objects
Ask the disease objects for likely symptom patterns
Send these to another diagnostic object, asking for
    goodness-of-fit data
...
```

This is of course a very crude model, suffering from the same flaws as many of the early medical systems – for instance, there is no notion of a disease as a time-dependent process in this. However, it conveys the basic idea. It also suggests that such a use of object-oriented programming is likely to involve unusually large objects by the standards of today's applications. A small illustrative prototype system, strictly frame- rather than object-based, existed as early as 1980 [Aikins 80].

Object-oriented systems also relate naturally to blackboard systems in AI. These provide a model for co-operating small and specialised expert systems, communicating through a large global data structure (or perhaps several such structures) called 'the blackboard'. In this paradigm a component specialist system is termed a 'knowledge source'; these can access and modify entries on the blackboard, and a great deal of dependency information about which entries were used to support the inclusion of other entries is also maintained on the blackboard. Activation of the knowledge sources is under the control of a scheduler, and in recent systems the scheduler is itself implemented as a group of knowledge sources with scheduling queues being maintained on a part of the blackboard. Blackboard systems are useful for solving problems akin to a jigsaw, in which islands of an answer start to emerge and give clues as to how to proceed. In the more elaborate systems the analogue is of a box of jigsaw parts from several complete pictures, each with parts missing! The connection with object-oriented systems here is that the entries which instances of knowledge sources make on the blackboard are essentially anonymous messages (junk mail?) to any of the other knowledge sources which want to compete to make use of it. An introduction to blackboard systems can be found in [Engelmore and Morgan 88].

## About the o-o languages used in AI

Perhaps the best-known object-oriented programming system in AI is still the Flavors package, available as an extension to many kinds of LISP but best developed for Symbolics' ZetaLISP. Distinguishing features include:

- an impure approach: you can, should you wish, poke about inside objects using raw LISP. Objects are represented using a specially-added data type, **dtp-instance**. The first word of this points to a **defstruct** attached as a property of the flavor (that is, class) name which contains information such as a method dispatch table; other words of a **dtp-instance** contain the instance variable values.

- no meta-classes. Meta-classes in other o-o languages are useful if you find a need to send messages to a class as an object, or if you need to have a single variable shared amongst several classes. There are, of course, always other ways to achieve this effect in an unsafe manner.

- multiple inheritance, with a large number of options and even ways to define your own. For example, there are ways to gather the results of all the methods which could handle a message, not just the first. There are ways to add default methods which will be invoked if nothing else locatable in the flavor lattice will handle a message. There are ways to force the system to try several possible methods, looking for some 'desirable' result (such as non-**nil**).

- the 'daemon' style of method modification. In this style, you add before- and/or after-daemon methods nearer in the lattice than the basic method, and these do any necessary further work. This further reduces code redundancy. Sometimes this is inadequate, for instance if an object needs to run a method in some special environment; for this, Flavors provides a facility called 'whoppers' which can be used to wrap up the sending of a message in whatever environmental modifications you like.

The possibilities for making exciting mistakes are widespread. Probably the best introduction to Flavors is [Bromley and Lamson 87]. For an excellent introduction to using the Common LISP Object System (CLOS) see [Keene 88].

PARLOG [Gregory 87] and other parallel logic programming languages provide, through perpetual processes and streams, an object-oriented viewpoint as well as much of the representational power of logic programming. PARLOG provides and- and or-parallelism, with control by two means: first, a predicate has an associated mode declaration for its arguments, and a called predicate with a more general argument set suspends until its arguments become instantiated

by the running of some other goals; and second, the clauses for a predicate can have guards, a set of PARLOG goals which should be free of side effects. The clause whose guard succeeds first wins, the competitors die. Streams, in the form of partially instantiated lists, provide the communication channels. In such a paradigm, an object is represented by a perpetual process created by running a suitably recursive predicate. There are no inheritance mechanisms provided as base-level features, but a variety can be implemented in a trivial and natural way. MANDALA [Furukawa et al 84] also uses streams for message-passing, but is more explicitly object-oriented and does provide inheritance mechanisms as explicit base-level features. It has an accompanying window-based programming environment. Flavors, PARLOG and MANDALA are conceptually closer to the actor family of languages in which objects can only communicate with known friends, than they are to the traditional hierarchically-based languages such as SmallTalk.

LOOPS [Bobrow and Stefik 83] is an add-on to Xerox' InterLISP running on Xerox workstations. Distinguishing features include:

- meta-classes
- the more widespread 'send-super' style of method combination. A 'send-super' call within a method definition causes the same message which invoked the method, or even a modified version of it, to be sent to the same receiving object, but causes that method definition to be bypassed in the search for a method to answer it. The 'send-super' style necessitates run-time look up of methods, unlike the 'daemon' style of Flavors provided that the inheritance lattice is not modified at run-time; however, 'send-super' is probably the more generally useful mechanism of the two.
- a method cache for fast method access, done in the same kind of way as is used for working set management in a paged memory operating system. Xerox claim a hit rate of around 90% typically, and a cost of less than twice a typical InterLISP fuction call for methods which are in the cache.
- composite objects. These are defined by a grouping of classes, so that when an instance of the group is created all the necessary objects and constraints between them are created (or, if you wish, the parts are created when needed rather than when an instance of the composite itself is created). Thus, for example, you might define a car as a composite object consisting of a body object, door objects and so on, with the doors constrained to be the same colour as the body and vice-versa.

There are a large number of other, experimental object-oriented languages in AI, but very few have wide acceptance outside the lab (or room) where they

were created. Experiments continue with newer features such as multi-methods, in which a method is identified not only by the message selector but also by the types of the message's arguments. This is still slightly awkward in LISP, because the Common LISP type system is not a true hierarchy – **nil**, for example, is a subtype of all others.

## Teaching issues

It is very important to introduce object-oriented techniques into the teaching of AI, although it is equally important to retain a sense of balance about it. Discussions about the utility of the concept of metaclasses, of multi-methods, of multiple inheritance and so on are really peripheral to the main threads of a good AI course. Multiple inheritance is of course very important in knowledge representation terms, but no one system-provided implementation is usually adequate for all purposes.

Any programming technique or methodology, including the object-oriented approach, suits some kinds of task and not others. To take a trivial example: the 'missionaries and cannibals' puzzle can be solved in an object-oriented system by having instances to represent the missionaries, the cannibals, the boat and the banks. There is no special virtue in this, however; this does not take advantage of what an object-oriented system is offering. On the other hand, one could solve the puzzle by having an object-oriented system generate all the instances of the puzzle's state initially, and then getting them to send messages to each other to organise themselves into chains of successor states. This provides a good way to map out the problem space completely, but it is a less natural way, and in more complex problems is computationally intractable - so it doesn't provide even a good line of attack on questions of search management.

There are practical problems. At present, it is expensive to equip a laboratory with good facilities for object-oriented work in AI, and this limits the throughput of students. For example, to run LOOPS you need a Xerox workstation. The learning time is quite long too, so the ratio of students to workstations must be low. For students who have already paid the price of learning LISP, there are a variety of Flavors packages available, but running these in an adequate LISP environment also necessitates the use of individual workstations, or of a small number of students per VAX-like machine. The same applies to running systems such as PARLOG or Concurrent Prolog on SUNs or VAXes. Good and cheap implementations of simpler object-oriented systems such as C++ and Actor are now appearing, but these still don't cater too well for symbol processing tasks and so are not the ideal solution. Personally I am also reluctant to commit students to using a single large system (such as LOOPS), because to do so would tend to

obscure issues of representation and efficiency; students would only be seeing one provider's decisions in the trade-off between them. Therefore I still tend towards the 'low technology' route, using the public domain systems such as XLISP, which has only two predefined objects yet offers a lot of flexibility, and Little SmallTalk which offers over 2000 predefined objects but suffers from the expressive shortcomings of early SmallTalk such as having only hierarchical inheritance and no layered database. Such systems are adequate for simple explorations of the issues, and are not too difficult to learn in a reasonable time, they are cheap and they run on a wide variety of hardware. It is also easy to move on from such toys to serious object-oriented languages and environments.

# References

[**Aikins 80**] J.Aikins, "Prototypes and production rules: an approach to knowledge representation for hypothesis formation", Stanford University Dept of Computer Science Tech. Report CS-80-814 (PhD thesis), 1980 (see also "Prototypical knowledge for expert systems" *Artificial Intelligence*, 20(2), pp.163-210, 1983)

[**Bobrow and Stefik 83**] D.Bobrow and M.Stefik, *The LOOPS Manual*, Xerox Corporation, Xerox PARC, 123pp., 1983

[**Bromley and Lamson 87**] H.Bromley and R.Lamson, *LISP Lore: A Guide To Programming The LISP Machine*, 337pp., Kluwer Academic Publishers, New York, 1987

[**Buchanan and Shortliffe 84**] B.Buchanan and E.Shortliee, *Rule-based Expert Systems*, 748pp., Addison-Wesley: Reading, Massachusetts, 1984

[**Engelmore and Morgan 88**] R.Engelmore and T.Morgan, *Blackboard Systems*, 600pp, Addison-Wesley, Reading, Massachusetts, 1988

[**Furukawa et al 84**] K.Furukawa, A.Takeuchi, S.Kunifuji, H.Yasukawa, M.Ohki and K.Ueda, "MANDALA: A logic based knowledge programming system", in *proceedings of the International Conference on Fifth Generation Computer Systems* (ICOT, Japan), pp.613-622, North-Holland, Amsterdam, 1984

[**Gregory 87**] S.Gregory, *Parallel Logic Programming in PARLOG: the language and its implementation*, 228pp., Addison-Wesley, Reading, Massachusetts, 1987

[**Hadley 88**] R.F.Hadley, "Logical omniscience, semantics, and models of belief", *Computational Intelligence*, 4(1), pp.17-30, 1988

[**Keene 88**] S.Keene, *Object-oriented programming in Common LISP: A programmer's guide to CLOS*, Addison-Wesley, Reading, Massachusetts, 1988

[**Levesque and Brachman 85**] H.Levesque and R.Brachman, "A fundamental tradeoff in knowledge representation and reasoning", in (eds) Brachman and Levesque, *Readings in Knowledge Representation*, 571pp., Morgan Kaufmann, Los Altos, California, 1985

[**Minsky 85**] M.Minsky, "A framework for representing knowledge', in (ed) P.Winston, *The Psychology Of Computer Vision*, McGraw-Hill, New York, 1975

[**Newell 81**] A.Newell, "The knowledge level", *AI Magazine*, 2(2), pp.1-20, 1981

[**Rosenschein 85**] J.S.Rosenschein, "Rational Interaction: Cooperation Among Intgelligent Agents", Stanford University Dept of Computer Science Tech. Report CS-85-1081 (PhD thesis), 133pp., 1985

## DISCUSSION

Professor Nygaard asked about the extent to which different debugging methodologies are used in the kind of problems covered by the talk. Dr. Ross replied that although there are some interesting developments in the area of debugging methodologies, they are in general not extensively applied.

Referring to possible definitions of Artificial Intelligence mentioned by Dr. Ross, Professor Nygaard said that one characteristics of AI is that it is a field which comprises a number of interesting problems, and a number of interesting problem solving techniques, but has not yet developed a unified framework for dealing with such problems and techniques. A possible contribution of object-oriented methodology to the AI research might be such concepts as inheritance, locality, etc., facilitating the introduction of a disciplined way of the design of complex AI systems.

Prof. Bayer asked what are possible applications of metaclasses in AI. Dr. Ross replied that the concept of a metaclass turns out to be useful if, for example, one wants to send a message to some collection of classes.