Object-oriented and frame-based programming in CommonORBIT

Koenraad De Smedt
Leiden University, Dept. of Experimental and Theoretical Psychology
P.O. Box 9555, 2300 RB Leiden, The Netherlands
desmedt@rulfsw.leidenuniv.nl

Abstract

Objects are representations of entities in a domain which is modeled in a computer. Each object encapsulates the knowledge relevant to one abstract concept or physical object in the real world. This knowledge may consist of data but also of procedures which are applicable to the object. Using the metaphor of a society of communicating entities, these procedures are activated by sending messages to objects. In an applicative view of object-oriented programming, however, procedures are called as generic functions rather than by sending messages. Object-oriented languages allow knowledge to be shared between several objects by a mechanism called *inheritance*.

The object-oriented language which is discussed below is CommonORBIT, an extension to Common LISP. It is an easy but powerful language, which offers a representation based on structured objects but is also inspired by frame-based systems.

Friday, March 10, 2006

Contents

1 The object-oriented paradigm

When we are faced with the task of analyzing and representing a cognitive domain, we will often start by identifying the kinds of entities in the domain. These *objects* may represent situations, concrete things, or abstract concepts. In the domain of medical diagnosis, some of the objects involved are patients, diseases, bacteria, and symptoms. In natural language processing, some of the objects are linguistic entities such as sentences, phrases, words, and speech sounds. In a man-computer interface, some of the objects are icons, windows, the user, and the computer.

Objects are structured: each object has a number of *aspects* which contain knowledge relevant to the object. E.g., a *patient* has aspects *current-symptoms, medical-history, blood-pressure,* etc. In addition, a patient has the same aspects which are relevant for each normal person, i.e., *name, sex, weight, age,* etc. Clearly, some of these aspects, e.g. *name,* contain static knowledge and could be stored as in a data base. But other aspects, e.g. *blood-pressure,* are dynamic and may continuously change during a computer simulation. Some aspects can be computed from other knowledge, e.g., the *age* of a person at a certain time can be computed from the person's *date-of-birth*.

Object-oriented languages provide a direct means for the representation and manipulation of such objects. Computational objects can be created and changed in the course of the computation. They are separate units, each one containing all the knowledge—procedures as well as data—which is relevant to an entity in the real world. This knowledge forms the internal state of an object, which can be modified during the computation. Thus, objects may represent entities which change over time and which interact in various ways.

Moon (1986) views the object-oriented programming paradigm as a technique to organize very large programs. Distribution of knowledge into manageable units makes it practical to deal with programs that would otherwise be impossibly complex. Objects provide a uniform way of structuring knowledge which supports programming by extending or reusing existing objects. Organizing knowledge in objects closely resembles the way in which scientists think about the knowledge in their domain. In a grammar book, for instance, the grammarian will often devote a separate chapter to the verb, another chapter to the noun, yet another to the sentence, etc. We replicate this organization by defining an object *noun*, an object *verb*, etc., thus establishing a rather direct knowledge transfer from a theoretical framework to the computer.

Object-oriented programming belongs to a family of knowledge representation languages including *frames* and *semantic networks*. The formalisms in this family all attempt to group knowledge in somewhat larger units than, for example, production rules. Semantic networks represent entities in memory, represented graphically as nodes, and associative relations between entities, represented graphically as arcs which link nodes. Arcs labeled *'is a'* or *'a kind of'* represent specialization relations which allow inheritance of properties (Brachman, 1979). *Frames* have somewhat lost the original meaning intended by Minsky (1975). In many frame-based languages, frames correspond closely to structured objects. The different aspects of knowledge about an object are usually called *slots* in frame terminology. In some frame systems, as in most semantic networks, slots contain only declarative knowledge (slot *fillers*). The inclusion of procedural knowledge in frame slots (which is standard in object-oriented languages) is called *procedural attachment* (Bobrow & Winograd, 1977).

The knowledge representation language which is discussed below is CommonORBIT (De Smedt, 1987). It is written as an extension of Common LISP (Steele, 1984; Winston & Horn, 1988). CommonORBIT, sometimes abbreviated as CORBIT, is the successor to ORBIT (Steels, 1983, 1985; De Smedt, 1984), which was written in Franz LISP. The ORBIT and CommonORBIT formalisms have been used in several applications, including natural language processing systems, a window system for a CRT terminal, and VLSI design.

This document introduces basic concepts of object-oriented programming and explains how they are realized in CommonORBIT. First it is explained how LISP can be extended in an object-oriented way by means of generic functions. Then two contrasting views on knowledge sharing are compared. Next, the primitives for creating and activating objects in CommonORBIT are introduced with some examples. The remainder deals with more advanced issues in object-oriented programming: multiple inheritance, roles, structured inheritance, memoization, aspect types, function paths, and merging of objects.

2 Extending LISP

As a programming paradigm, object-orientedness cuts across other classifications, such as imperative, functional, or logic based languages. Hence it will not be surprising that many object-oriented languages are extensions of existing non-object-oriented languages. Some of the first object-oriented languages were implemented as extensions of an *imperative language*, such as SIMULA (Dahl & Nygaard, 1966; Dahl, Myhrhaug & Nygaard, 1971). Later, extensions of LISP and logic languages became popular.

2.1 Message passing

Some object-oriented languages are *message passing* systems, e.g. SMALLTALK (Ingalls, 1978; Goldberg & Robson, 1983) or the ACT-I formalism (Hewitt, 1979). A message passing system is a computational architecture where every object is viewed as a processor, sometimes called an *actor*, which can send and receive messages. Each object has a number of *methods* which it uses to react to incoming messages. In general, objects may react to messages by creating more objects, sending more messages, and/or updating their local state.

A message passing system can very well be implemented in hardware by constructing a network of separate processors. The metaphor of message passing between objects is very powerful because it is simple and general. It has become so dominant that it is often considered to be virtually identical to object-oriented programming. Message passing allows the programmer to empathize with the concepts he defines. Such programs therefore have an anthropomorphic feel, e.g. the following *method* written in ‘procedural English’:

> “If I'm a Fever object, and I get a message asking for my Most-Likely-Cause, I ask my Temperature whether it is greater than 100, and I ask my duration whether it is greater than 3 days. If so, I reply with a Serious-Infection. Otherwise, I reply with the Flu.” (Lieberman, 1987)

Some extensions of LISP, e.g. the original Flavors system (Weinreb & Moon, 1980) use a message passing architecture. Message passing languages use a special message passing operator, e.g. `SEND` in the following example:

```
(SEND FEVER-1 MOST-LIKELY-CAUSE)
```

However, message passing introduces into LISP a radically new feature which does not fit into its original *applicative* style. In an applicative language, the basic operation is the application of a function, the primitive element for constructing new functions is abstraction, and in principle, the only side-effect is the definition of a new function (McCarthy, 1960). Therefore, Steels (1983) suggests *generic functions* as a way of extending LISP elegantly without incurring the overhead of message passing. Using generic functions in LISP rather than messages allows a normal functional LISP syntax to be used for communicating with objects:

```
(MOST-LIKELY-CAUSE FEVER-1)
```

2.2 Generic functions

A generic function is a function whose definition is not a single body of code but is distributed among objects. Like a normal LISP function, it is called on arguments, performs certain operations, and returns values. Unlike an ordinary function, the actual operation to be invoked is not stored in the function definition itself, but in the arguments that the function is applied to. Conceptually, generic functions are useful because they allow one function name to be used for a high-level operation which requires different work for different kinds of objects. For the caller, generic functions provide a simpler interface, because the differentiation is carried out automatically by the generic function rather than explicitly by the caller.

Imagine that you are writing a text editor and a high-level function is needed for deleting entities in the editor domain. Of course, different operations are required depending on whether a character, word, sentence, paragraph, etc. is to be deleted. The interface may be simplified by having one name for all these actions. So it is appropriate to write a generic function DELETE which performs the correct task regardless of the arguments it is applied to. The conventional way to write such a function is to make a procedure with a number of cases for each kind of object involved, for example:

> To DELETE:
> If the object to be deleted is a *character*, use procedure *A*,
> If the object to be deleted is a *word*, use procedure *B*, etc.

Suppose that we want to add another function for a different operation, then again we have to consider the different cases, for example:

> To MOVE:
> If the object to be moved is a *character*, use procedure *C*,
> If the object to be moved is a *word*, use procedure *D*, etc.

This approach is typical for an *action-oriented* style. We think in terms of what actions need to be taken, analyze the possible cases and then define actions for each case. Consequently, the knowledge about each action is grouped in one place, but the knowledge about each object is distributed. Action-oriented programming languages stimulate this approach by providing primitives for defining subroutines, conditional statements, etc.

A different, *object-oriented* way is to write the function as a generic function. The function is defined within the context of the objects it will be applied to, e.g.:

For a CHARACTER:
To *delete* the object, use procedure *A*,
To *move* the object, use procedure *C*, etc.

For a WORD:
To *delete* the object, use procedure *B*,
To *move* the object, use procedure *D*, etc.

If we take this approach, we think in terms of what kinds of objects are in the domain, and then group the possible actions for each kind. As a result, e.g., the definition of the function DELETE is *distributed* among several objects. Object-oriented languages support this approach by providing primitives to create objects and generic functions, add or change knowledge associated with objects, make objects share knowledge, etc. Generic functions in LISP gave rise to the design and implementation of ORBIT and CommonORBIT. Other extensions of LISP along the same lines are the New Flavors system on the LISP Machine (Moon, 1986), CommonLoops (Bobrow et al., 1985), and CLOS, which has been accepted as part of the Common LISP standard (Keene, 1989).

2.3 Mixing styles

CommonORBIT is designed as a *minimal* extension of LISP, i.e., it aims at an integration of object-oriented programming constructs with applicative programming while avoiding new syntax as much as possible. The invocation of a CommonORBIT function on an object is therefore implemented as a normal LISP function call. As an alternative, the metaphor of message passing can still be realized in CommonORBIT. The function is then viewed as the message and the first argument as the object which receives the message. It is straightforward to implement a message passing syntax with the following LISP macro:

```
(DEFMACRO SEND (OBJECT MESSAGE &REST ARGS)
  `(FUNCALL ,MESSAGE ,OBJECT ,@ARGS))
```

However, generic functions offer a smoother extension of LISP than the incorporation of message passing would. Generic functions in CommonORBIT can be traced, used as functional arguments, and treated as functions in every respect. This allows object-oriented code to be mixed easily with conventional LISP code, and even to write down a complete function call in a program *before* deciding whether the function will be defined as a generic or normal function. This is important because CommonORBIT is intended as an extension of LISP, not as a replacement.

In addition, there is a special provision in CommonORBIT which allows an object-oriented function to also have a *global* definition. This allows a generic function to be applicable to objects as well as to other kinds of arguments such as numbers or lists. The object-oriented definition, if present for the given arguments, is the preferred one. If no object-oriented definition can be found, then a global definition associated with the function is applied. If neither an object-oriented nor a global definition is found for a generic function, the symbol `UNDEFINED` is returned.

3 Knowledge sharing

An important feature of object-oriented languages is that they provide some kind of mechanism for objects to share their structure and behavior with other ones. Sharing is

usually one-way (to avoid circularity): an object can inherit knowledge associated with another one.

3.1 Why knowledge sharing?

The goals of knowledge sharing in object-oriented programming can be seen from different points of view:

1 *Specialization*. From a conceptual point of view, knowledge sharing mechanisms allow subtyping in the form of specializations of a general object. In this way a *specialization hierarchy* is produced, which corresponds closely to the hierarchy of *'is a'* relations in a semantic network. Sometimes the hierarchy will contain concepts representing concrete entities. E.g., *sedan* and *RV* might be subtypes of *car*. In other cases, the programmer will use hierarchies to model intangible concepts that are nevertheless external to the program. E.g., a hierarchy of grammar concepts might contain objects for *word* and its subtypes *noun, verb,* etc.

2 *Combination*. Another use of knowledge sharing is the representation of an object as a combination of two or more other objects. This kind of knowledge sharing is often called *multiple inheritance*. This is often done if an object needs to integrate knowledge from different sources or perspectives. E.g., *John* may be a male patient; thus the behavior of *man* and that of *patient* is combined. Composition will often consist of the addition of a few special features (e.g. *patient*) to a more general-purpose category (e.g. *man*); the more special proxy is sometimes called a *mixin*. A *transitive compound strong verb* combines the behavior of *transitive, compound,* and *strong verb*. There are some conceptual pitfalls here. It cannot be expected that an object *little galaxy* can be simply composed of an object *little* and an object *galaxy;* there is some kind of interdependence. Other well-known examples are *toy truck, past president,* etc.

3 *Stepwise refinement*. A program can be constructed by first modeling the most general concepts in the application domain, and then dealing with special cases through more specialized objects. The programmer is not so much concerned here with the construction of a taxonomy but rather with refinement as a programming methodology. Stepwise refinement *by specialization* can be compared with the well-known methodology of stepwise refinement *by decomposition* (Wirth, 1971). It is significant that the effort of defining an object is proportional to the extent in which it *differs* from other objects. Thus refinement is not only useful as a programming methodology, but it can also be thought of as a general cognitive mechanism, for it reflects a principle of least effort.

4 *Avoiding redundancy*. From the point of view of data storage, knowledge sharing provides efficiency by avoiding redundancy. A piece of information which is necessary in many objects needs to be stored in only one object. This not only reduces the memory needed to store a piece of knowledge, but also improves modularity because the shared knowledge needs to be updated only in one place. Again, this can be thought of as a general cognitive principle and not just a software engineering strategy.

3.2 Proxies (stereotypes) versus classes

Two architectures for specialization are distinguished. The simplest one is to let objects freely act as *proxies* for other objects. When an object cannot satisfy a request to exhibit a certain behavior, it may *delegate* the request to another object, which then acts as its proxy. The delegating object is said to be a *client* of the proxy. The intuitive account in a message passing system might be: "I don't know how to handle this message, can you respond for me?" For generic functions, it could be worded as "I don't have a definition for this generic

function, do you have one for me?" A client may have several proxies, each handling specific requests.

An specialization network (or inheritance network) is a labeled directed graph whose nodes represent objects and whose arcs denote specialization relations between the objects. The relation *"x is a client of y"* is written as $x \rightarrow y$. To avoid circularity, the graph must be acyclic. An example of a specialization network is shown in Figure 1. Note that objects may delegate to more than one proxy.

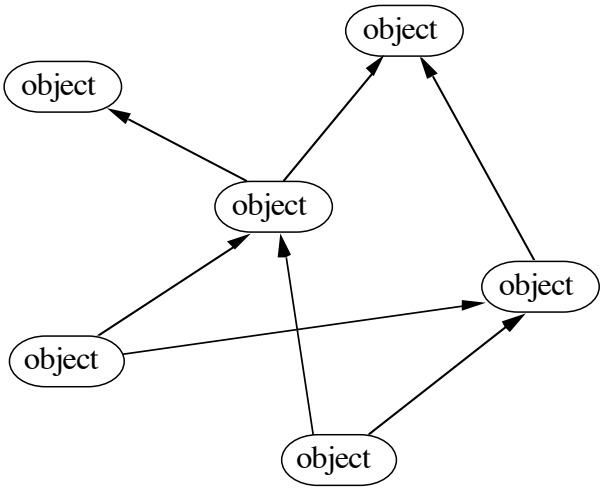


Figure 1
Proxy/client relationships in a specialization network

A proxy can thus be viewed as a *stereotype* (sometimes called a *prototype*). It may refer to a typical situation or to an ideal object. By comparing other objects to the stereotype, similarities and differences emerge. It is possible that no other object has *all* the properties of the stereotype. The effort involved in the definition of a new object is proportional to the extent in which it differs from another, prototypical object. These differences may be exceptions; e.g., a bird can fly, but an ostrich, which is a bird, cannot fly. The differences may also consist of details which are not represented in the stereotype at all; e.g., a particular ostrich may match the ostrich stereotype, but in addition, it may have a black spot on the left foot, which is irrelevant for the stereotype.

A completely different approach to knowledge sharing is to define the common structure and behavior of a set of objects in a *class* (also called a *type* or a *flavor*). Each object must then be an *instance* of a class. Classes represent abstract *categories*; they are themselves *not* objects in this approach. While instances maintain their own internal state in the course of computation, they may *inherit* default information and behavior from their classes. Classes may be built on other classes (*super*classes), from which they inherit the knowledge common to all constituent classes. An example configuration of classes and instances is given in Figure 2.

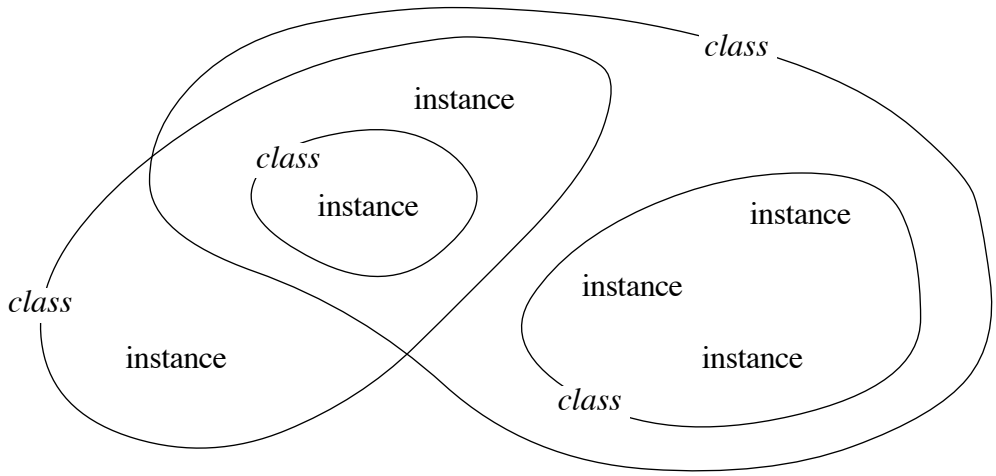


Figure 2
Classes and instances

The proxy/client architecture, which is adopted in CommonORBIT, is by virtue of its uniformity more general than the class/instance architecture. Class/instance systems are restricted to a representation of *'is a'* relations, because they do not allow objects to inherit information from other objects; proxy/client systems allow this and thus they can represent the more general *'is like'* relation (also called the *conformance* relation). In a proxy/client system, classes can easily be modeled by designating certain prototypical objects as proxies for a set of clients. The reverse—using classes as stereotypes—is more difficult, because the class/instance distinction imposes restrictions on inheritance. In particular, class definitions apply only to the instances of the class, not to the class itself; hence, a class is not an object and cannot itself perform any operations; classes have no use except for creating instances. Some languages, e.g. SIMULA, do not even allow the creation of classes while the program is running; classes are defined at compile time and remain fixed thereafter.

3.3 Using the LISP datatype hierarchy

Several programming languages allow the definition of abstract structured datatypes. By allowing the encapsulation of procedures as well as data in datatypes, these datatypes can be used as object classes. The encapsulated procedures thus become typed generic functions. Such extensions of LISP, which include New Flavors, CommonLoops, and CLOS, are well integrated into the LISP core in the sense that the specialization hierarchy of object classes forms a part of the LISP datatype hierarchy. CLOS and KRS (Van Marcke, 1987) even provide classes for most of the standard Common LISP datatypes. A consequence of the datatype approach is the necessity of a class/instance distinction in these languages.

Using the host language's type hierarchy is not very suitable for implementing a language like CommonORBIT, where objects are not rigidly classified. Therefore, CommonORBIT adds just one new datatype to LISP, the structured type OBJECT (and one subtype, NAMED-OBJECT). All objects are of this type, and proxy-client relations are stored *independently* of the LISP datatype hierarchy. In a similar way, Objective-C (Cox, 1986) adds just one datatype to C.

4 Objects in CommonORBIT

A basic CommonORBIT object is implemented as a Common LISP structure of type OBJECT. Objects have a number of *aspects (slots)* which contain information about the object. Aspects can be activated by calling the aspect name as a generic function with the object as its first argument.

4.1 Named and anonymous objects

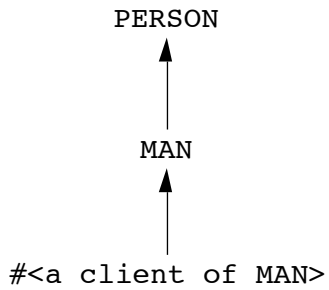
Objects can be either named or anonymous. A named object has a name, which is a symbol that can be used to access it[1]. Basic objects, which do not have a name, are usually called *anonymous* objects. Anonymous objects are created with the `AN` or `A` macros while named objects are created with the `DEFOBJECT` macro. Here are some examples. The result of evaluation is given below each form.

```
(DEFOBJECT MAN PERSON
  (SEX :VALUE 'MALE))
#<object MAN>

(SETQ THIS-MAN
  (A MAN
     (FIRST-NAME :VALUE 'JOHN)
     (BIRTHYEAR :VALUE 1954)))
#<a client of MAN>
```

The code above—the order of the expressions does not matter—defines three objects. The named object `MAN` is a client of the named object `PERSON` (which is automatically created). The object bound to the variable `THIS-MAN` is an anonymous object which is a client of `MAN`. The specialization hierarchy at this point is graphically represented as follows:

(1)



4.1.1 The printed representation of objects

The printed representation of objects is normally signalled by angle brackets preceded by a sharp sign, for example `#<...>`. The printed representation of named objects is based on the name, preceded by the word "`object`"[2], for example `#<object MAN>`. By default, the representation of anonymous objects is `#<an object>`. However, when the object has roles

---

[1] An object with name `N` can be accessed by means of the form `(OBJECT 'N)`. However, this is often unnecessary because CommonORBIT generic functions automatically coerce names to their corresponding objects.

[2] The switch ONLY-NAME controls whether the full printed representation of a named object is given, or only the symbol which is its name.

(see below) or proxies, the printed representation is based on the roles or proxies, for example `#<the HUSBAND of <a client of WOMAN>>.`

4.2 Declarative and procedural knowledge in aspects

All knowledge contained in CommonORBIT objects is retrieved by means of generic functions. Local definitions of generic functions are called *aspects*. For programming convenience, several aspect types are provided. Aspects of type `:VALUE`, such as `BIRTHYEAR` and `SEX` in the above examples, express declarative knowledge. They simply return a value (the aspect *filler*) when the generic function is called, e.g.:

```
(BIRTHYEAR THIS-MAN)
1954

(SEX THIS-MAN)
MALE
```

The value returned by the generic function `BIRTHYEAR` is directly retrieved from the aspect definition in the argument. The value returned for the generic function `SEX` is obtained by delegation to the object `MAN`.

Another aspect type is `:FUNCTION`, which expresses procedural knowledge. The aspect filler is a function which is applied to the given arguments. An example of a `:FUNCTION` aspect is given below. It defines a function to compute the age a person will reach in a certain year.

```
(DEFOBJECT PERSON
  (AGE :FUNCTION #'(LAMBDA (SELF YEAR)
                     (- YEAR (BIRTHYEAR SELF)))))
#<object PERSON>

(AGE THIS-MAN 1992)
38
```

The usual *lambda binding* in LISP is used when the function is applied to its arguments. With the appropriate bindings, the body of the function is executed and returns a value. Note that it is not required for any arguments except the first one to be an object. Only the first argument acts as a selector for the definition[3]. A generic function must therefore take at least one argument. While in some object-oriented languages, `SELF` is a reserved word denoting the object itself, this is not the case in CommonORBIT. It is a normal parameter of the function and could have any other name.

In addition to `:VALUE` and `:FUNCTION`, which are the most basic aspect types, CommonORBIT offers a few more aspect types. They will be discussed below. All aspect types, however, are activated uniformly as generic functions. Some object-oriented languages, e.g. the New Flavors system, have a different syntax for different kinds of aspects: aspects of type `:FUNCTION` are defined and called as generic functions (*methods*) whereas aspects of type `:VALUE` are accessed as variables (*instance variables*).

---

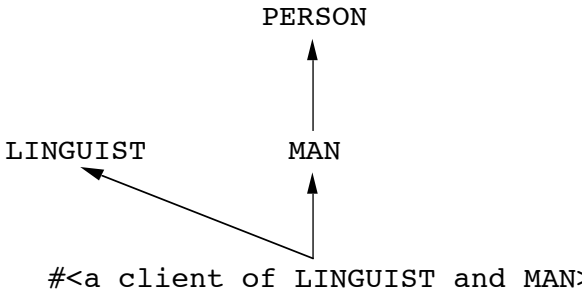
[3] CommonLoops and CLOS have taken the concept of a generic function one step further in that the selection of a definition depends on all arguments, not just the first one; thus there are no privileged argument positions for objects. Generic functions may be defined for standard LISP types as well as for user-defined object types, and for any combination of arguments to the function. Thus, the concept of generic function is an extension and a generalization of the concept of ordinary Common Lisp functions (cf. DeMichiel & Gabriel, 1987).

4.3 Adding knowledge to existing objects

After an object is defined, more proxies and aspects can be dynamically added. The following example adds a proxy by means of an `IS-CLIENT` operation[4]. The effect on the specialization hierarchy is graphically represented in (2).

```
(IS-CLIENT THIS-MAN 'LINGUIST)
#<a client of LINGUIST and MAN>
```

(2)



Aspects can be added to objects by means of `DEFASPECT`. This operation requires arguments for the aspect name, the object, the aspect type, and the *filler* (the actual contents of the definition)[5]. Here are some examples.

```
(DEFASPECT LAST-NAME THIS-MAN :VALUE 'DOE)
LAST-NAME

(DEFASPECT PRINT-NAME 'PERSON :FUNCTION
  #'(LAMBDA (SELF)
      (PRINC (FIRST-NAME SELF))
      (PRINC " ")
      (PRINC (LAST-NAME SELF))
      T))
PRINT-NAME

(PRINT-NAME THIS-MAN)
JOHN DOE
T
```

Proxies and aspect can be removed again from objects for which they are defined. The functions `IS-NO-PROXY` and `UNDEFASPECT` can be used for these purposes. The capacity to undo definitions provides flexibility which is often needed during the prototyping phase of a program.

5 Knowledge sharing mechanisms

Knowledge sharing in object-oriented systems is based on *defaults* rather than absolute and irretractable statements. If we say “birds fly” then we mean “birds typically fly” and we have no problems to accept and handle exceptions—e.g. ostriches and penguins—adequately. Exceptions cannot be handled well in first order logic, because to derive for a particular bird that it can fly, it would be necessary to first prove that it is not exceptional, i.e., that it is *not* an ostrich or a penguin, etc. Rather, we want to be able to derive that a bird can fly if it

---

[4] Unlike `DEFOBJECT`, `IS-CLIENT` evaluates its arguments (the objects).

[5] Unlike `DEFOBJECT`, `DEFASPECT` evaluates the argument specifying the object.

cannot be shown that it is exceptional (cf. Brewka, 1989). In object-oriented languages, inheritance is based on this kind of reasoning—default reasoning. All inherited knowledge only holds in so far as it is not overruled by knowledge in the inheriting object.

5.1 Inheritance by delegation vs. inheritance by copying

In any system which represents a part of the real world, changes in the real world are the enemy, because they require updating of stored information. When various knowledge bases depend on one another, updating one piece of information may require a substantial amount of updating in other parts of the system as well. Therefore it is important that knowledge-based systems which are used in rapidly changing situations, such as in office systems, or during the development stage of a large program, allow efficient updating. Because object-oriented languages may be used to build very large specialization hierarchies, the architecture for sharing common knowledge must allow such updating: changes to an object must be immediately accessible to other objects which share its knowledge. This is basically an issue of modularity: a change in one module of a system should not disturb other modules. In how far such modularity can be realized, depends on how dynamic the mechanism for sharing common behavior is.

There are basically two ways in which the specialization relation can be used for sharing definitions. I will use the term *inheritance by copying* for the technique of copying default information from a more general object to a more specialized one, or from a class to a subclass or instance. This kind of inheritance is typical for many class/instance systems. A class is like a mould which shapes each instance. For example, when a CLOS object is created, a structure is created which contains all slots with defaults as specified in the object class. Obviously, copying involves a great deal of updating when defaults change[6].

In contrast, *inheritance by delegation* (or *dynamic inheritance*) is a technique for accessing knowledge in other objects only when it is needed. This is typical for systems such as CommonORBIT, which use the proxy/client metaphor. When a specialization relation is established between two objects, no defaults are copied. Thus, objects remain largely empty. When defaults are changed in an object, the changes are automatically accessible to all its clients. A system based on delegation is therefore more modular.

5.2 Multiple inheritance

In systems which allow an object to have only one immediate proxy, the specialization hierarchy forms a tree. Other systems, such as CommonORBIT, allow objects to have multiple proxies. In systems allowing multiplicity, the hierarchy is no tree, because it may branch upward as well as downward from a given node; therefore it is a directed graph[7] and is often called a *tangled* hierarchy. A mechanism for sharing common behavior which allows multiplicity can be very powerful, because it allows the creation of new objects as a combination of other ones. However, there is always the possibility that the knowledge to be combined is conflicting. It is a thorny issue how such conflicts can be resolved. In any case

---

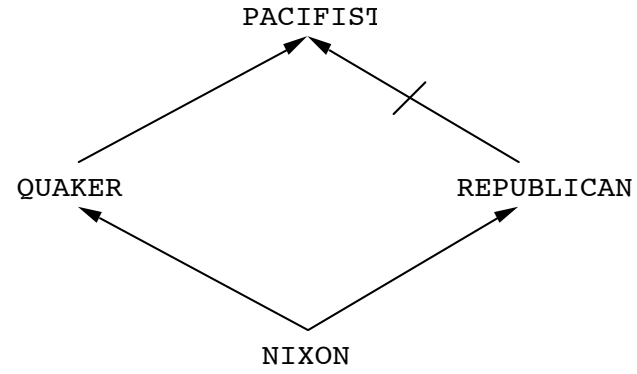
[6] CLOS provides an option of sharing slots. Thus copying is avoided, but it is not a proper delegation technique, because changing the slot in one instance will affect all other instances and even the class itself. This is contrary to the idea of specialization, because it allows information to spread from more specific objects to more general ones.

[7] The graph representing the specialization relations in a multiple delegation system is usually a-cyclic, meaning that objects may not delegate to themselves (directly or indirectly). CommonORBIT checks for cyclic proxy/client relationships and issues a warning if necessary.

it is better to avoid conflicts and compose new objects out of prototypes which are either complementary or so different that they do not interfere.

The canonical example of conflicting information in an inheritance network is the so-called 'Nixon diamond', which is depicted in Figure 3. Nixon is both a Quaker and a Republican. Quakers are typically pacifists while Republicans typically are not. In a *bipolar* system (Touretzky, 1986; Touretzky, Horty & Thomason, 1987), which allows both positive (→) and negative (∅,/) inheritance links, the network can be represented as Figure 3a; in a *unipolar* system, such as CommonORBIT, it can be represented as conflicting defaults, e.g. as shown in Figure 3b.

(a)



(b)

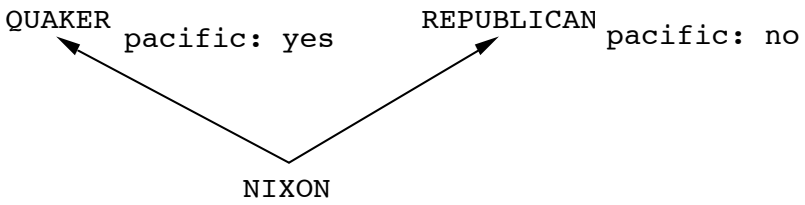


Figure 3
The Nixon diamond

What should we conclude from these conflicting informations? If we take a skeptical view, we could remain agnostic about Nixon's pacifism in this case. If we take a more credulous attitude, we might try to conclude as much as possible, producing multiple 'acceptable' belief sets. A different solution, which is adopted in many object-oriented languages, consists of establishing explicit priorities between proxies. Thus, the programmer has to decide whether it is more important for Nixon to be a Republican or a Quaker. If by convention we decide to represent this ordering graphically from left to right, then in Figure 3 it is represented that Nixon is more a Quaker than a Republican. If we accept that the priority of proxies is extended to *their* proxies, then Nixon is also more a pacifist than a Republican. Thus the hierarchy is searched in a depth-first manner, and we conclude that Nixon is anti-military. Multiplicity in CommonORBIT is formulated in the principles (3). Clearly, the arbitrary ordering imposed by (3b) is necessary only in a system like CommonORBIT, where proxies can be dynamically added.

(3) *Multiplicity:*

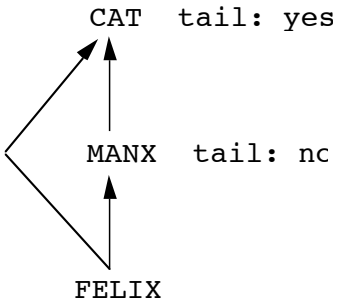
a. *(Local ordering)* A proxy occurring earlier in the `DEFOBJECT` form has a higher local priority than a proxy occurring later in that form.

b. *(Recency)* A proxy added later during execution of the program has a higher local priority than a proxy added earlier.

c. *(Depth-first)* If an object *a* has direct proxies *b* and *c*, and *b* has a higher priority than *c*, then this priority is extended upward so that all proxies of *b* have a higher priority than *c* and all proxies of *c*.

Multiplicity is thus a partial order on the recursive proxies of an object. Since specialization also defines a partial order on the recursive proxies, the question is whether these two relations can be integrated to determine the inheritance ordering. As long as specialization and multiplicity do not contradict each other, this is straightforward. But unfortunately, it is all too easy to find contradictions, and it will turn out we have really opened Pandora's box. The specialization networks in Figure 4 are some examples of contradictions between specialization and multiplicity. Figure 4a shows a *redundant* arc which might cause inheritance to skip over a specialization; Figure 4b shows an example without redundant arcs but where a strict depth-first search according to multiplicity would nevertheless violate the specialization partial order.

(a)



(b)

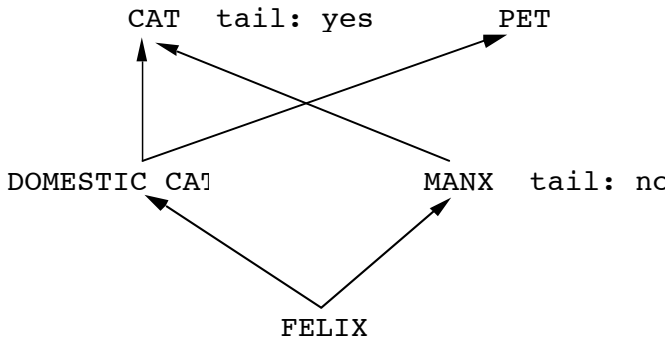


Figure 4
Specialization vs. multiplicity

Should we restrict the importance of the local ordering for the sake of specialization? It is indeed generally agreed upon (e.g. Touretzky, 1986) that inheritance must always follow the specialization partial order. This principle, formulated in (4), prevents 'shortcuts' in the hierarchy.

(4) *Specialization vs. Multiplicity:*

> Inheritance must follow the specialization partial order; therefore, in any case the specialization relation excels the multiplicity relation.

It must be pointed out that this principle is completely arbitrary. We could as well have chosen the opposite and so we could have used redundancy arcs as a programming control tool enabling us, e.g., to take deliberate shortcuts in the hierarchy[8]. Moreover, principle (4) may yield counterintuitive results in some cases. It is not difficult to imagine several examples which are isomorphic to Figure 4b but which suggest different ordering principles (this is left as an exercise to the reader).

Several algorithms which obey principle (4) can be constructed. An efficient and straightforward algorithm could climb depth-first as high as possible but would skip objects which have clients which have not been considered. This algorithm, which is used in CommonORBIT, is discussed in more detail below. Unfortunately, the hierarchy may exhibit conflicting multiplicity relations, e.g. Figure 5.

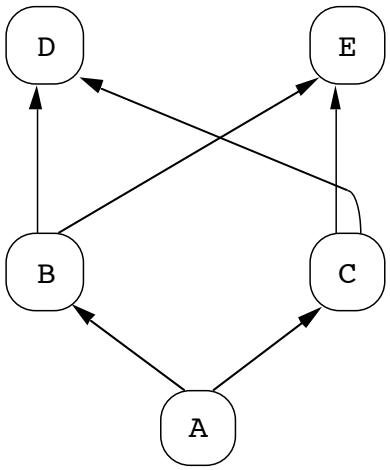


Figure 5
Conflicting multiplicity relations

In this case, the algorithm will choose the ordering (A B C E D), rather than (A B C D E). It can be argued that this is not correct, since the former ordering does not reflect the local precedence in the leftmost branch. It is possible to obtain the latter ordering with a different algorithm, but such an algorithm would be more time consuming (cf. Ducournau & Habib, 1987). Since hierarchies with conflicting multiplicity relations are atypical and counterintuitive anyway, the simpler algorithm is the default in CommonORBIT.

Moreover, computing a total linear ordering of all recursive proxies of an object is not always absolutely necessary in systems which search at most one definition in the proxy hierarchy. Since CommonORBIT is such a system, it searches the proxy tree in a modified depth first manner—skipping proxies if they still have clients waiting in the hierarchy—and the search is halted as soon as a proxy can provide a definition for the aspect in question.

We conclude that the inheritance mechanism in CommonORBIT is a practical one which is in accordance with commonly accepted principles, but does not resolve the clash of intuitions about ambiguous networks (cf. also Touretzky, Horty & Thomason, 1987).

---

[8] CommonORBIT, on the contrary, offers an option to automatically prune redundant specialization links. It is clear that this is a dangerous option in a system where proxies can be dynamically removed.

5.3 Combining operations

The simplest kind of inheritance consists of choosing only one definition—the most specific—and ignoring all others. This most basic form of default inheritance is used in CommonORBIT. Simply ignoring all other possible definitions higher up in the hierarchy is useful if one wants the most specific kind of knowledge to simply override any defaults provided by more general knowledge.

However, there are occasions when retrieving an aspect definition from just one object in the hierarchy and simply applying it does not meet the needs of an application. We may want to combine operations retrieved from different objects, or modify a result obtained by applying an inherited definition. Suppose, for example, that that the name of a person is represented as a list of the first and last names. Suppose, furthermore, that the name of a medical doctor is represented like that of a normal person, except that the letters M.D. should be added at the end. Thus, the aspect `NAME` is inherited, but the result is to be modified afterward. In CommonORBIT, such 'custom-made' delegation can be achieved by using the `DELEGATE` macro. Consider the following definition for `PERSON`:

```
(DEFOBJECT PERSON
   (NAME :FUNCTION #'(LAMBDA (SELF)
                       (LIST (FIRST-NAME SELF)
                             (LAST-NAME SELF)))))
#<object PERSON>

(NAME (A PERSON
         (FIRST-NAME 'JOHN)
         (LAST-NAME 'DOE)))
(JOHN DOE)
```

Now we define an object `MEDICAL-DOCTOR` with an aspect `NAME` that uses the definition of `NAME` in `PERSON` but adds "`M.D.`" at the end. The `DELEGATE` macro is used to delegate the aspect `NAME` to `PERSON` and apply the obtained definition locally to the delegating object by giving it the argument `SELF`[9]. The result is then modified by `APPEND`:

```
(DEFOBJECT MEDICAL-DOCTOR
  (NAME :FUNCTION #'(LAMBDA (SELF)
                      (APPEND (DELEGATE (NAME 'PERSON) SELF)
                              '(M.D.)))))
#<object MEDICAL-DOCTOR>

(NAME (A MEDICAL-DOCTOR
         (FIRST-NAME 'JOHN)
         (LAST-NAME 'DOE)))
(JOHN DOE M.D.)
```

Using `DELEGATE` instead of rewriting the complete definition for `MEDICAL-DOCTOR` not only simplifies the program by storing the knowledge in only one place, but it also has the advantage that the program is more modular: if the definition of `NAME` for `PERSON` is changed, the change will automatically apply to `MEDICAL-DOCTOR`.

When it is desirable to delegate to more than one object in sequence, `DELEGATE` can be used as a control tool to specify an ordering. Suppose that we have a window system with

---

[9] In fact, *any* arguments may be given to the function obtained by `DELEGATE`, thus allowing localized as well as non-localized delegation. Lieberman (1986) calls non-localized delegation 'true' delegation, because the client leaves the proxy to process the message on his own.

definitions for framed windows and labeled windows. Suppose then that we would like to add a *framed labeled* window which is drawn by first drawing it as a framed window and then adding the label. A definition of such a window could be as follows:

```
(DEFOBJECT FRAMED-LABELED-WINDOW
  WINDOW
  (DRAW :FUNCTION
    #'(LAMBDA (SELF)
        (DELEGATE (DRAW 'FRAMED-WINDOW) SELF)
        (DELEGATE (DRAW-LABEL 'LABELED-WINDOW) SELF))))
#<object FRAMED-LABELED-WINDOW>
```

In contrast with CommonORBIT, which provides only `DELEGATE` as a general and flexible tool for customizing delegation, CLOS provides a set of predefined tools for the automatic combination of values generated by the application of methods. Predefined *method combination types* include the operators `LIST`, `+`, `MAX`, etc. In addition to a *primary method* provided by one class, additional methods may be provided which are called either *before* or *after* the execution of the primary method. Thus the order in which methods are executed can be controlled. Finally, arbitrary LISP code can be wrapped around the other methods to set up an environment in which the other methods are executed.

6 Roles

6.1 Reverse evaluation and backpointers

In an object-oriented system, it is sometimes useful to be able to perform *reverse evaluation*. Reverse evaluation is the ability to list the objects for which a given function will return a given value. E.g. we may want to know whose brother Peter is, in other words, all objects for which the function `BROTHER` will return `PETER`. In CommonORBIT, reverse evaluation is possible when the value of an aspect is itself also an object (e.g. the named object `PETER`). In that case we use the aspect type `:OBJECT` rather than `:VALUE`. When such an aspect is defined, a *backpointer* is kept; the aspect filler is said to have a *role* in the object for which the aspect is defined. For example, consider the following definitions of the named objects `PETER`, `MARY` and `KATHY`:

```
(DEFOBJECT PETER PERSON)
#<object PETER>

(DEFOBJECT MARY
  PERSON
  (BROTHER :OBJECT 'PETER))
#<object MARY>

(DEFOBJECT KATHY
  PERSON
  (BROTHER :OBJECT 'PETER))
#<object KATHY>
```

Normal (forward) evaluation of the generic function `BROTHER` will produce the following results:

```
(BROTHER 'MARY)
#<object PETER>
```

```
(BROTHER 'KATHY)
#<object PETER>
```

Reverse evaluation is performed by means of WHOSE and always returns a list:

```
(WHOSE 'BROTHER 'PETER)
(#<object KATHY> #<object MARY>)
```

We say that PETER has the role of BROTHER (or is the *role filler* of BROTHER) with respect to MARY and KATHY. A backpointer is stored in PETER for each role it has in another object. These relations are depicted in Figure 6, where labeled arrows represent aspects and dashed arrows represent roles.

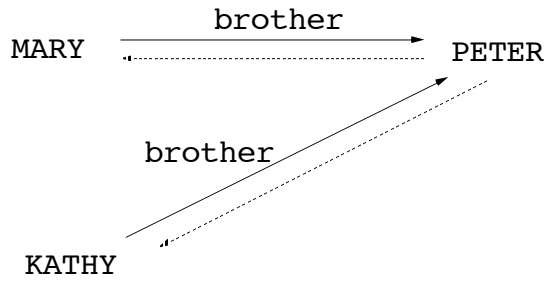


Figure 6
Roles

The backpointers allow us to answer the question, whose brother PETER is, without having to access *every* object in the system. The information is directly available in the object PETER itself. Reverse evaluation follows the backpointers and returns a list of objects which satisfy the requirement that PETER fills their BROTHER aspect. Reverse evaluation provides a richer access to knowledge. In the event of changes, backpointers are automatically updated. In this way, consistency between (forward) evaluation and reverse evaluation is maintained in the CommonORBIT environment.

6.2 Plural aspects

When an aspect filler is a list of objects, it is sometimes desirable to give individual roles to each object in the list. For example, if Mary has three brothers—represented as a list of objects—, then for each of them we would like to establish a BROTHER role. In analogy to the aspect type :OBJECT, CommonORBIT offers the aspect type :OBJECTS for this purpose. The following example is schematically presented in Figure 7.

```
(DEFOBJECT BRIGITTE
  PERSON
  (BROTHER :OBJECTS (LIST 'OTTO 'IVAN 'MICHAEL)))
#<object BRIGITTE>

(WHOSE 'BROTHER 'MICHAEL)
(#<object BRIGITTE>)
```

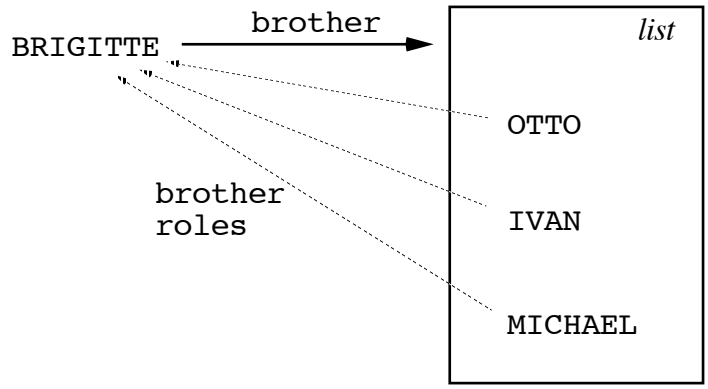


Figure 7
Roles in an aspect of type *:OBJECTS*

To make the representation language more natural, CommonORBIT allows the use of *plural* aspect names which implicitly define aspects of type `:OBJECTS`. Plural aspect names end on "`-S`", which is stripped from the aspect name when it is used as a role. For example:

```
(DEFOBJECT CAROLINE
  PERSON
  (BROTHER-S (BROTHER 'BRIGITTE)))
#<object CAROLINE>

(WHOSE 'BROTHER 'MICHAEL)
(#<object CAROLINE> #<object BRIGITTE>)
```

7 Structured inheritance

Structured inheritance has been described as the ability to "… preserve a complex set of relations between description parts as one moves down the specialization hierarchy" (Brachman & Schmolze, 1985:177). In other words, inheritance is not limited to simply sharing a value, but it models an object and the network of all its associated objects after one higher up in the hierarchy. This mechanism, which is central in KL-ONE, is also present in CommonORBIT. Consider, e.g., the CommonORBIT definitions for `WOMAN` and `PERSON` below.

```
(DEFOBJECT WOMAN
  "A woman is a person of the female sex."
  PERSON
  (SEX :VALUE 'FEMALE))
#<object WOMAN>

(DEFOBJECT PERSON
  "The mother of a person is a woman who is,
   by default, not a virgin."
  (MOTHER :OBJECT (A WOMAN
                     (VIRGIN? NIL))))
#<object PERSON>
```

Delegation of the `MOTHER` aspect to `PERSON` involves more than simple retrieval of the aspect filler, otherwise every person would share the same mother. Therefore, CommonORBIT makes a client of the mother object for every client of `PERSON`, so that every person has a

unique mother[10]. This mother object will be created when needed and permanently stored, for example in the following situation[11]. The specialization hierarchy is graphically represented in Figure 8.

```
(DEFOBJECT OLIVIA PERSON)
#<object OLIVIA>

(MOTHER 'OLIVIA)
#<the MOTHER of OLIVIA>

(SEX (MOTHER 'OLIVIA))
FEMALE

(VIRGIN? (MOTHER 'OLIVIA))
NIL
```

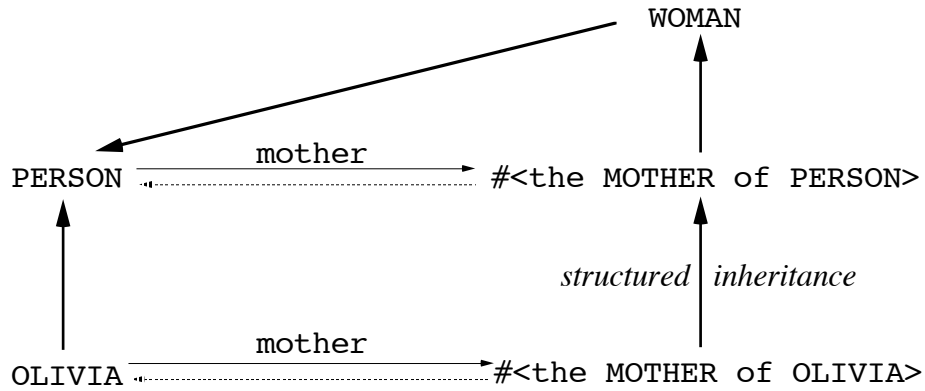


Figure 8
Structured inheritance

Structured inheritance can be viewed as the enforcement of a default on aspect fillers via their roles. This default is also active when the mother aspect of a client of `PERSON` is explicitly defined. In that case, CommonORBIT tries to establish a specialization relation between the mother of the client of `PERSON` and the mother in `PERSON` itself. This is illustrated by the following CommonORBIT sequence.

```
(DEFOBJECT HELGA
  PERSON
  (MOTHER :OBJECT (A MEDICAL-DOCTOR))))
#<object HELGA>

(SEX (MOTHER 'HELGA))
FEMALE

(VIRGIN? (MOTHER 'HELGA))
NIL
```

---

[10] This only works for *anonymous* objects. When an aspect of type `:OBJECT` is filled by a *named* object, that object itself is always returned. This has proved to be in line with the programmer's intuitions, although some counter-examples could be contrived. A cleaner solution would be, to dedicate a special aspect type to the structured delegation mechanism, for example, `:INSTANTIATE`.

[11] The creation of clients by structured delegation happens only when they are needed in order to prevent infinite recursion.

Of course, defaults obtained via structured inheritance relations can still be overridden, because they are just like other client/proxy relations in this respect. The following example speaks for itself.

```
(DEFOBJECT JESUS
  GOD PERSON
  (MOTHER :OBJECT (DEFOBJECT MARY
                    (VIRGIN? T))))
#<object JESUS>

(VIRGIN? (MOTHER 'JESUS))
T

(SEX 'MARY)
FEMALE
```

Summing up, structured inheritance means that when a specialization relation holds between two objects, then specialization relations are also established between their aspect fillers. By consequence, the inheritance mechanism effectively operates on a whole structure or network of objects rather than on isolated entities[12].

8 Memoization

Memoization is a general method to save time which would be needed for repeated computations at the expense of a little extra storage. When it is likely the result of a computation will be needed again later, it can be saved as a *memo*[13]. Rather than wasting precious time on recomputation, the memo can simply be fetched when needed. In CommonORBIT, memos are part of the internal state of an object. Memoization is enabled in various circumstances, which will be described in this section.

8.1 Computing a value when needed

Sometimes an aspect is defined as a form to be computed only once for each object. The resulting value can then be stored and should not be recomputed on subsequent access. The aspect type `:IF-NEEDED` achieves precisely this goal by storing the value as a memo. It is like `:FUNCTION` but memoizes the result of the computation by redefining the aspect as one of type `:VALUE`. Memoization is *lazy*, i.e., the value is not computed until it is needed—when the generic function is first called on the object. For example:

---

[12] Although the current implementation of structured delegation is dynamic in the sense that an anonymous instance is created only when needed, it is also static in the sense that a delegation link between a filler in the client and a filler in the proxies is established at *definition* time, not during each retrieval. Obviously, this is done for efficiency reasons. But since there is no updating mechanism for this, structured delegation links are not undone when the state of affairs changes such that the links would not be warranted anymore, for example when the role filler in the proxy loses its role or when the role object loses its client-proxy relationship.

It must also be mentioned that aspects of type `:OBJECTS` do not offer structured delegation. This is mainly because it is not easy to define what kind of delegation would apply. Need there be an aspect of type `:OBJECTS` in the proxies, or one of type `:OBJECT`? Of course, it is always possible to program some form of inheritance explicitly by means of an `:IF-NEEDED` aspect.

[13] Or it can be saved in a *cache*—this is simply another metaphor for the same thing (e.g. as used in KRS).

```
(DEFOBJECT PERSON
   (NAME :IF-NEEDED #'(LAMBDA (SELF)
                        (LIST (FIRST-NAME SELF)
                              (LAST-NAME SELF)))))
#<object PERSON>

(SETQ J (A PERSON
           (FIRST-NAME 'JOHN)
           (LAST-NAME 'DOE)))
#<a client of PERSON>

(NAME J)
(JOHN DOE)
```

The result of calling NAME is now memoized, i.e. J now has an aspect NAME of type :VALUE which contains the name as it is computed by the :IF-NEEDED form. Thus, the name is immediately available for retrieval when it is needed again.

8.2 Saving inherited values

Retrieving knowledge dynamically from a specialization hierarchy reduces the burden of updating, but it is often costly in terms of search time, because the hierarchy must be searched every time knowledge is needed. Thus, inheritance by copying may increase the speed of the system. On the other hand, the extra storage increases overall paging time in virtual memory systems, which slows them down again. Lieberman (1986) elaborates this prestorage vs. computation trade-off.

CommonORBIT solves the choice between delegation and copying by providing an option to make a memo of results obtained by delegation. The switch function MEMOIZE-ASPECTS enables memoization of an aspect definition of type :VALUE, :OBJECT or :OBJECTS to clients after they have first delegated the aspect[14]. Subsequent accesses will be faster because the hierarchy does not have to be searched for this aspect anymore. The option leaves the choice to tune the system for a certain application. In systems which undergo many changes, modularity will be important and delegation is probably preferred; stable systems for which speed is critical and storage is available can rely on memoization.

8.3 Updating

Memoization is contrary to the idea of inheritance by delegation. Changes to objects which have provided definitions may invalidate the information memoized in other objects. As long as inherited knowledge is not stored, it can be recomputed at any time to reflect the current state of affairs. E.g., suppose that a client of BIRD inherits the information that it can fly. If later it becomes known that the bird is also a client of the object PENGUIN (which is a specialization of BIRD), then the question, whether our bird can fly, normally gets a different answer due to a new search of the specialization hierarchy. Thus, inheritance reflects non-monotonicity: additional information may show that the case at hand is exceptional and the default conclusion has to be retracted (cf. Brewka, 1989). This is illustrated in Figure 9.

[14] As already mentioned, objects created by structured inheritance are *always* memoized.

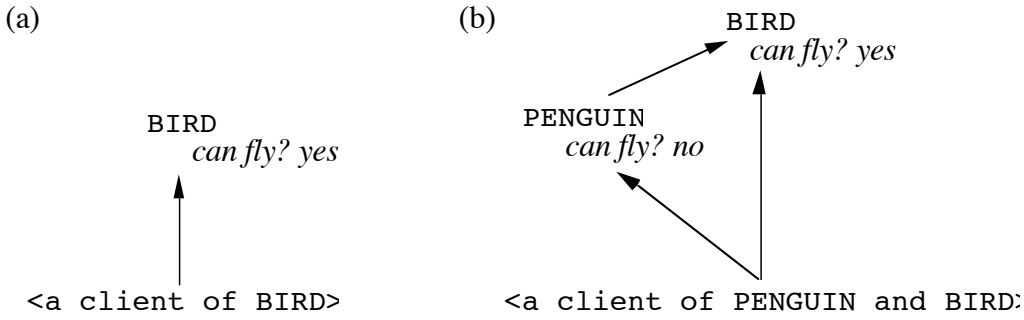


Figure 9
A change to the specialization hierarchy invalidates previous inferences.

In a dynamic inheritance system, knowledge is normally recomputed rather than retracted. However, when knowledge is memoized, it is normally computed just once and stored; hence the memoized value must be removed. CommonORBIT maintains an *updating* network which allows memoizations to be automatically undone when they are no longer valid. This applies to stored results of `:IF-NEEDED`[15] as well as to inherited values and objects[16]. Of course, new values may be memoized again when they are recomputed in the new state of affairs.

Memoization *without* updating may be a conscious choice for a programmer who wants objects to be 'frozen', no matter which changes apply to related objects. This can be done for efficiency or for safety—when correctness of the program depends on knowledge *not* changing. CommonORBIT provides this option in the form of the switch `UPDATE-MEMOS`.

9 Aspect types and paths

In principle, there could be just one kind of generic function which applies a function definition to the arguments and returns a result. However, for efficiency as well as for programming convenience, there are a number of different aspect types with different kinds of behavior. A CommonORBIT aspect definition consists of a type and a filler. The type determines how the filler is interpreted when activated. The following overview presents the five CommonORBIT aspect types which have already been introduced above.

- `:FUNCTION` is the basic aspect type. The aspect filler is a function which is applied to all arguments.
- `:IF-NEEDED` is like `:FUNCTION` but memoizes the result of the application when the function is called, so that the next time the function is called, the computation is not redone but the memoized value is simply returned.
- `:VALUE` simply returns a value. This aspect filler is computed and stored at the time of definition.
- `:OBJECT` is like `:VALUE` but tries to coerce the filler to a CommonORBIT object. A backpointer (or *role*) is automatically stored in the filler.

---

[15] Although the update mechanism effectively propagates changes of aspect definitions throughout the specialization hierarchy, it does not safeguard against changes of functions (generic or not) called *within* `:IF-NEEDED` forms.

[16] In the case of an object created by structured inheritance, no local (non-inherited) information is transferred to the new one. Although this may sometimes appear unnecessary, it warrants against possible inconsistencies which might arise when old local information is associated with a newly inherited object .

- `:OBJECTS` is like `:VALUE` but tries to coerce the filler to a list of CommonORBIT objects; for each object in the list, a role is stored.

9.1 Special aspect types

To enhance the expressive power of the language, there are a number of additional aspect types in CommonORBIT. Although these aspects have special semantics, the aspects are eventually defined internally as aspects of type `:FUNCTION` or `:IF-NEEDED`. Additional aspect types are indicated by keywords in the `DEFASPECT` form; these keywords may occur either after the `:FUNCTION` or `:IF-NEEDED` keywords, or by themselves.

- `:DELEGATE` delegates the aspect to another object, which may or may not exist yet. (Default `:FUNCTION`).
- `:ADJOIN` adds a value (with `ADJOIN` or `UNION`, depending on whether it is an atom or not) to the result of delegating this aspect to the proxies. (Default `:IF-NEEDED`).
- `:REMOVE` removes a value (with `REMOVE` or `SET-DIFFERENCE`, depending on whether it is an atom or not) to the result of delegating this aspect to the proxies. (Default `:IF-NEEDED`).
- `:ASK` queries the user for a value. The filler (optional) is a format-string which is used to query the user. This format string will be called with two arguments: the aspect name and the object. (Default `:IF-NEEDED`).

Some examples of special aspect types are given below. The number of these aspect types is purposely kept small so as not to overwhelm the programmer with a baroque set of tools. Rather than creating a different aspect type for each different behavior, the programmer is encouraged to define this behavior as a LISP form within an existing aspect type.

```
(DEFOBJECT PERSON
  (BIRTHYEAR :ASK))
  (AGE :FUNCTION #'(LAMBDA (SELF YEAR)
                     (- YEAR (BIRTHYEAR SELF)))))
#<object PERSON>

(DEFOBJECT PET ANIMAL
  (BIRTHYEAR :IF-NEEDED :DELEGATE 'PERSON)
  (AGE :DELEGATE 'PERSON))

(DEFOBJECT FIDO PET DOG)
#<object FIDO>

(AGE 'FIDO 1992)
What is the BIRTHYEAR of #<object FIDO>? 1980
12

(DEFOBJECT COMPUTER
  (PART-S (LIST 'MEMORY 'CPU)))
#<object COMPUTER>

(SETQ MY-COMPUTER
      (A COMPUTER (PART-S :ADJOIN 'HARD-DISK)))
#<a client of COMPUTER>

(PART-S MY-COMPUTER)
(HARD-DISK MEMORY CPU)
```

9.2 Paths of functions

Sometimes one would like to specify a *path* of functions for an object, whether or not all objects in the path are actually created. Suppose we are writing a program for a word processor, with objects for windows and buffers. Buffers are associated with windows, windows have corners, and corners have *x* and *y* coordinates. Suppose one would like express the statement that the x-coordinate of the upper left corner of the window associated with the object `BUFFER` has the value `1`. The following CommonORBIT definition would be one possibility, using `DEFASPECT` to change the value of the aspect `X-COORDINATE`:

```
(DEFASPECT X-COORDINATE (UPPER-LEFT-CORNER (WINDOW 'BUFFER))
  :VALUE 1)
X-COORDINATE
```

However, this will only work if at this point in the computation, the object representing the window and the object representing its upper left corner are already defined. But if they don't exist yet, there is no object for which we attempt to define the function `X-COORDINATE`. CommonORBIT will signal an error in such a case[17]. In a problem domain where such situations often occur, it is a tedious task to check whether objects have been created before defining aspects which presuppose their existence.

To solve this problem, CommonORBIT has a provision for the automatic creation of objects in a path in case they do not exist yet. The path is defined by specifying a *list* of functions instead of a single function in the `DEFASPECT` syntax:

```
(DEFASPECT (WINDOW UPPER-LEFT-CORNER X-COORDINATE) 'BUFFER
  :VALUE 1)
X-COORDINATE

(X-COORDINATE (UPPER-LEFT-CORNER (WINDOW 'BUFFER)))
1
```

When the above code is processed, the path is scanned from left to right[18]. If the objects in the path already exist, no action is taken. If they do not exist, an anonymous object is created for every step which is undefined. In this case, both the window and its upper left corner are automatically created.

10 Referents and merging of objects

Sometimes it is useful to merge two objects into one, or to give two names to one object, e.g., to denote different viewpoints on the object. Such representations are possible in CommonORBIT because an object is actually a two-level structure. The first level, which is called the *reference*, contains a pointer to the second level, which is called the *referent* of the object. All information about an object (except the name of a named object) is stored in its referent. Normally, each reference has its own referent which denotes an individual external entity. Figure 10 is a schematic overview of this representation.

---

[17] Moreover, if the functions `UPPER-LEFT-CORNER` or `WINDOW` are not yet defined, the LISP interpreter will signal an error as well

[18] Note that the order of functions in the path is the reverse of the order in the function application. This is because the left-to-right order seems more logical here than the function application order.

Anonymous object:

Named object:

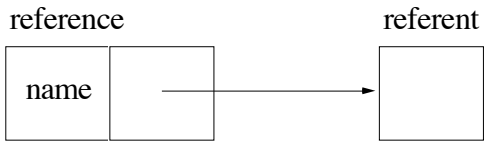


Figure 10
Objects and referents

By creating a second reference to an existing referent, we can create two *coreferential* objects. Due to the two-level structure, any subsequent changes to one object will also affect the other. This has proved useful, e.g., in a model of the student administration of the University of Nijmegen, which for a while assigned two distinct student numbers each student.

Also, we can merge two existing objects by transferring all information from one referent to the other[19] and subsequently redirecting the pointer from the 'emptied' referent to the 'filled' one. The two objects then share a referent, which remains accessible via both references. Merging objects is useful in situations when two objects with different names turn out to be the same entity in the real world. A famous example is that of the morning star and the evening star, which were initially thought of as separate objects but turned out to be the same planet. The effect of merging objects is exemplified in Figure 11.

```
(MERGE-OBJECTS 'MORNING-STAR 'EVENING-STAR)
#<object MORNING-STAR>
```

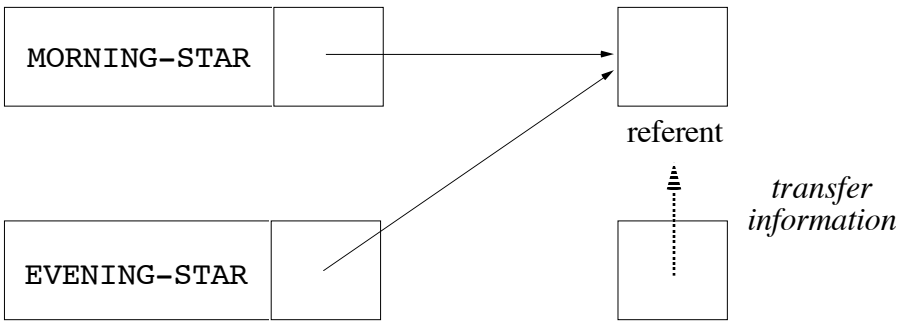


Figure 11
Merging objects

Decoupling the representation of an object and its referent raises the question, how equality of objects must be defined. Two distinct equality tests are provided. The predicate

[19] In case of conflict, the knowledge in the first object overrules that in the second. This, unfornunately, makes the operation non-symmetric, but it always succeeds. Another operation, UNIFY-OBJECTS, succeeds only if there is no conflict.

OQL tests whether two CommonORBIT objects are denotationally equal, i.e., whether they share the same referent. The normal EQ test in Common LISP can be used to test whether two objects are identical references.

Using coreference in object-oriented representation is further developed by Ferber and Volle (1988) who have implemented a logic for coreferential reasoning. Their system also allows references to be accessed from their referents, which is not possible in CommonORBIT. Coreferentiality is essential for an object-oriented implementation of *unification* in natural language processing.

11 Concluding remarks

An object-oriented programming language differs from non-object-oriented languages in the way knowledge is organized. Careful comparison of existing object-oriented languages reveals that this programming paradigm actually consists of a set of related, subtly different metaphors. This work has presented in detail one such system, CommonORBIT, and has pointed out some similarities and differences with other approaches.

CommonORBIT emerges as a system which is easy to use because its syntax fits well in the normal LISP syntax. It is an expressive language because its mechanism for behavior sharing is allows the most general kind of delegation. It is a flexible language because delegation relations can be specified on the level of single aspects. It provides both modularity and efficiency because it offers memoization in addition to delegation. It is compact because there are reasonable defaults for common operations. On the other hand, it is admitted that CommonORBIT has its limitations. It is a system where many concepts are realized to the extent that they are practical, but are not realized to the fullest. First, generic functions are not generalized to cover all arguments of the function, just the first one. Second, structured inheritance does not cover multiple roles, is static (established at definition time) and does not offer multiple inheritance. Finally, the updating mechanism works fine for memoized aspects as such but cannot retract arbitrary computations based on the value of an aspect.

Object-oriented programming embodies a different cognitive architecture than non-object-oriented systems. Simulations of intelligent processes may benefit from inheritance if the cognitive domain exhibits knowledge sharing. In addition, there are some issues involving modularity and efficiency of large programs. It offers a different kind of *modularity* than an action-oriented style. In an object-oriented style, it is easy to include new objects (or classes) in the domain of a generic function, because we only need to add code, not modify existing code. An action-oriented style is more modular with respect to adding a function. One simply adds the function definition, whereas in the object-oriented style, all relevant object types have to be modified to include the new operation.

Grouping knowledge either in one place or the other will also have an influence on *efficiency*, because storage and retrieval of knowledge requires time. In an action-oriented style, the number of object types will increase the average search time for a specific operation in the conditional statement, because more cases will have to be considered. In the object-oriented approach, such an increase has no effect because the knowledge is associated directly with each type. If, on the other hand, the number of possible actions for each type increases, then, in the object-oriented approach, searching this action among the other actions associated with the type will consume more time. When considering these differences, one should be aware of the fact that there are two levels on which a knowledge representation environment can support object-oriented programming: that of the internal representation and

that of the user interface. It is quite possible for a knowledge representation environment to store knowledge in an action-oriented way, while presenting it to the user in an object-oriented way, and vice versa.

A growing number of knowledge representation environments supporting an object-oriented or frame-based programming style is available today, either commercially or within the academic community. Several object-oriented languages have been successfully used to write impressive application programs, especially in Artificial Intelligence research, but also in more general areas of symbolic computing, including graphics and systems programming. Object-oriented programming has created new metaphors for the manipulation of symbols in a computer (such as the *message passing* or the *actors* metaphor). These metaphors allow the programmer to reason about a program in a different way. They have also encouraged the exploration of different computer hardware architectures.

At present, some trends can clearly be discerned in object-oriented programming. On the one hand, object-oriented programming languages are moving away from being just extensions or variants of conventional languages. KRS, e.g., is a powerful knowledge representation framework with many unconventional capacities. It maintains an representation of every concept at an *intensional* as well as an *extensional* level. It is outfitted with a full-fledged truth-maintenance mechanism (Van Marcke, 1986, 1987) and is programmable on a meta-level so it can modify itself by introspection (Maes, 1986)[20]. On the other hand, the object-oriented representation paradigm is being merged with other knowledge representation paradigms commonly used in AI. The language FORK (Beckstein, Görz & Tielemann, 1987) is one of many systems where object-oriented and rule-oriented programming are integrated. Also worth mentioning are object-oriented variants of logic languages (e.g. Kahn, Tribble, Miller & Bobrow, 1987). Object-oriented representation is also merged with conventional database management systems (for example, the Statice system by Symbolics). This trend shows that an object-oriented organization of knowledge is beginning to have an impact on other programming styles. What I envision for the future is an open-ended knowledge representation paradigm where many different control structures and organizations of knowledge can coexist.

References

Beckstein, C., Görz, G. & Tielemann, M. 1987. FORK: a system for object- and rule-oriented programming. *Proceedings of ECOOP'78 (Bigre+Globule 54),* pp. 303-314.

---

[20] Unfortunately, the price paid by KRS programmers is a syntax which is much more complicated than that of CommonORBIT. For example, the definition of the AGE aspect in CommonORBIT which was given above is rewritten in KRS as follows:

```
(DEFCONCEPT PERSON
  (BIRTH-YEAR)
  ((AGE
     (A SUBJECT
       (ARGUMENTS [ARGS ?CURRENT-YEAR])
       (DEFINITION [FORM (A NUMBER
                           (DEFINITION
                             [FORM
                              (- (>> REFERENT OF ?CURRENT-YEAR)
                                 (>> REFERENT BIRTH-YEAR))
                             ]))])))))
```

Bobrow D.G. & Winograd, T. 1977. An overview of KRL, a Knowledge Representation Language. *Cognitive Science,* 1, 3-46.

Bobrow D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M. & Zdybel, F. 1985. *CommonLoops: Merging Common LISP and Object-Oriented Programming.* Report ISL-85-8, Xerox Palo Alto Research Center, Palo Alto (CA).

Brachman, R.J. 1979. On the Epistemological Status of Semantic Networks. In: Findler, N.V. (ed.) *Associative Networks: Representation and Use of Knowledge by Computers* (pp. 3-50). New York: Academic Press.

Brachman, R.J. & Schmolze, J.G. 1985. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science,* 9, 171-216.

Brewka, G. 1989. Nonmonotonic logics—a brief overview. *AI Communications* 2, 88-97.

Cox, B. 1986. *Object-oriented programming: an evolutionary approach.* Reading: Addison-Wesley.

Dahl, O.J. & Nygaard, K. 1966. SIMULA — an ALGOL-based simulation language. *CACM,* 9, 671-678.

Dahl, O.J., Myhrhaug, B. & Nygaard, K. 1971. *SIMULA 67 Common Base Language.* Pub. S-22, Norwegian Computing Center, Oslo.

De Smedt, K. 1984. *ORBIT, an object-oriented extension of Lisp.* Internal Report 84-FU-13. Psychologisch Laboratorium, University of Nijmegen (NL).

De Smedt, K. 1987. Object-oriented programming in Flavors and CommonORBIT. In: Hawley, R. (ed.) *Artificial Intelligence programming environments* (pp. 157-176). Chichester: Ellis Horwood.

Ducournau, R. & Habib, M. 1987. On some algorithms for multiple inheritance in object-oriented programming. In: *Proceedings of ECOOP'78 (Bigre+Globule 54),* 291-300. Paris: AFCET.

Fikes, R. & Kehler, T. 1985. The role of frame-based representation in reasoning. *CACM,* 28, No. 9, 904-920.

DeMichiel, L.G. & Gabriel, R.P. 1987. The Common Lisp Object System: an overview. In: *Proceedings of ECOOP'78 (Bigre+Globule 54)*, pp. 201-220. Paris: AFCET.

Ferber, J. & Volle, Ph. 1988. Using coreference in object-oriented representations. In: Kodratoff, Y. (ed.) *Proceedings of the 8th European Conference on Artificial Intelligence* (pp. 238-240). London: Pitman.

Gallaire, H. 1986. Merging Objects and Logic Programming: Relational Semantics. In: *Proceedings of the 5th National Conference on AI. AAAI-86* (pp. 754-758). Morgan Kauffmann Publishers, Inc. Los Altos (Calif.).

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

Hewitt, C. 1979. Viewing Control Structures as Patterns of Passing Messages. In: Winston, P. & Brown, R. (eds.) *Artificial Intelligence: an MIT Perspective.* Cambridge (MA): MIT Press.

Ingalls, D. H. 1976. The Smalltalk-76 Programming System: Design and Implementation. In: *Conference record of the Fifth Annual ACM Symposium on Principles of Programming Languages* (pp. 9-16). Tucson (AZ).

Kahn, K., Tribble, E.D., Miller, M.S. & Bobrow, D.G. 1987. VULCAN: logical concurrent objects. In: Shriver, B. & Wegner, P. (eds.) *Research directions in object-oriented programming.* Cambridge, MA: MIT Press.

Keene, S. E. 1988. *Object-oriented programming in Common Lisp.* Reading (MA): Addison-Wesley.

Lieberman, H. 1986. Using prototypical objects to represent shared behavior in object-oriented systems. In: *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* Portland (Oregon). *SigPlan Notices* 21, 214-223.

Lieberman, H. 1987. Languages, object-oriented. In: Shapiro, S. (ed.), *Encyclopedia of Artificial Intelligence,* Vol. 1, (pp. 452-456). New York: Wiley.

Maes, P. 1986. Introspection in knowledge representation. In: *ECAI-86. Proceedings of the 7th European Conference on Artificial Intelligence* (pp. 256-269).

McCarthy, J. 1960. Recursive functions of symbolic expressions. *CACM,* 3, 184-195.

Minsky, M. 1975. A framework for representing knowledge. In: Winston, P. (ed.) *The psychology of computer vision* (pp. 211-277). New York: McGraw-Hill.

Moon, D. A. 1986. Object-Oriented Programming with FLAVORS. *Proceedings of OOPSLA '86.* ACM.

Shapiro, E., & Takeuchi, A. 1983. *Object Oriented Programming in Concurrent Prolog.* Technical Report TR-004, Institute for New Generation Computer Technology, Tokyo.

Steele, G.L. 1984. *Common LISP: the language.* Digital Press.

Steels, L. 1983. ORBIT: An applicative view of object-oriented programming. In: Degano & Sandewall (eds.), *Proceedings of the European Conference On Integrated Interactive Computing Systems, Stresa, Italy, 1-3 Sept. 1982.* North-Holland, Amsterdam.

Steels, L. 1985. Constraints as consultants. In: Steels, L. & Campbell, J.A. (eds.) *Progress in Artificial Intelligence* (pp. 146-165). Chichester: Ellis Horwood.

Touretzky, D.S. 1986. *The mathematics of inheritance systems.* Los Altos: Morgan Kaufmann.

Touretzky, D.S., Horty, J.F. & Thomason, R.H. 1987. A clash of intuitions: The current state of nonmonotonic multiple inheritance systems. In: *Proceedings of the 10th IJCAI, Milan*, pp. 476-482. Los Altos: Morgan Kaufmann.

Van Marcke, K. 1986. FPPD: a consistency maintenance system based on forward propagation of proposition denials. *Proceedings of The 7th European Conference on Artificial Intelligence, Brighton,* pp. 278-290.

Van Marcke, K. 1987. KRS: An Object-Oriented Representation Language., *Revue d'Intelligence Artificielle,* 1, No. 4.

Winston, P.H. & Horn, B.K.P. 1988. *Lisp* (third edition). Reading (MA): Addison-Wesley.

Weinreb, D. & Moon, D. 1980. *Flavors: message passing in the Lisp Machine.* Memo AIM-602, MIT, Cambridge, MA.

Wirth, N. 1971. Program development by stepwise refinement. *Communications of the ACM* 14, 221-227.