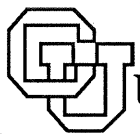
The OPUS User Manual

Alex Repenning

CU-CS-556-91 October 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

The OPUS User Manual

Alex Repenning

CU-CS-556-91 October 1991

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

(303) 492-7514
(303) 492-2844 Fax

The OPUS User Manual

Alex Repenning

October 28, 1991

Department of Computer Science
Campus Box 430
University of Colorado, Boulder CO 80309
492-1218, ralex@cs.colorado.edu
&
Asea Brown Boveri Research Center
Artificial Intelligence Group
Baden, AG 5405
Switzerland

Make it as simple as possible, but not simpler
- Albert Einstein

1. Abstract

Object-oriented programming techniques have proven to be a powerful paradigm to overcome many problems addressed by software engineering. Issues like fast prototyping, fast implementation, ease of maintenance and ease of modification are well supported in systems providing highly interactive user interfaces. The programming language Common Lisp is an ideal platform on which to build object-oriented tools because its environment supports the incremental development of software.

OPUS is an object-oriented system based on Common Lisp influenced by Smalltalk. It consists of a very compact kernel being easily portable to different Common Lisp environments. In contrast to most object-oriented systems based on Lisp, OPUS treats classes as real objects, i.e., classes have their own methods and variables. Rather than exploiting many different concepts in a hybrid fashion, OPUS employs few concepts and applies them consistently.

2. Introduction

This report is intended as a user's manual . It is intended to help programmers to exploit the OPUS system efficiently. All important concepts and features introduced in this report are augmented with sample source code in order to ease the process of understanding. However, general object-oriented programming issues are only covered as required. This report is by no means a replacement for a theoretical introduction to object-oriented programming.

A good preparation before reading this report is the paper of Stephik [1] and a basic understanding of Common Lisp.

The concepts of object-oriented programming are highly intertwined and non orthogonal. Object variables, methods, inheritance, etc. cannot be explained easily without referencing each other. This report reflects this property intrinsic to object-oriented programming. Rather than providing a large set of shallow explanations this report is organized into a few sections describing properties of OPUS, or object-oriented programming in general, in depth. To prevent too much redundancy, references between sections in form of footnotes are included. The reader is encouraged to follow these pointers and to read this report in a non sequential fashion.

During the early design and implementation phase of the KEN expert system shell [2] in 1985 the need for a compact, highly portable and efficient object-oriented subsystem based on Common Lisp was expressed. The very ambitious perception of portability embodied a hard constraint (in terms of memory) on the evaluation process to choose an existing object-oriented system. We had to cope with the weakest element of the chain - in our case an IBM PC. The need of large flexibility in the graphics domain led us to the implementation of an object-oriented system.

Not surprisingly from today's view, our system very quickly migrated from a specific graphic tool to a full-size generic object-oriented system. Still, we believe that the development of OPUS was of crucial importance and helped us to overcome significant software engineering problems involved in large projects.

3. Philosophy of OPUS

It is the philosophy of OPUS to provide a small set of concepts sufficient to provide a powerful tool for software design and implementation. Furthermore, OPUS is intended as a very efficient implementation platform for many portable tools. The portability aspect is emphasized by the explicit distinction between the specification and implementation parts of a system and by furnishing a mechanism to enforce compatibility between the two parts.

3.1. Objects are Data plus Behavior

On a low level of abstraction, programs can be viewed as an aggregation of bits and bytes. Slightly more interesting, they could be perceived in terms of Turing machines. Neither perception is indeed very helpful in coping with the complexity of today's software projects. Wirth [3] points out that programs are the sum of algorithms and data structures. This model exhibits the very important separation of programs into two separate views. However, the notion of programs is quite abstract. What is a program? How can it be

composed from smaller pieces? What kind of pieces? The object-oriented paradigm provides a partial answer [4]. In the designer's model of functionality to be achieved, the basic terms are objects and the relationships among them.

To represent objects from the real world, e.g., physical objects or conceptual objects, with software objects on a computers appears to be quite "natural". Each software object consists of data and behavior (Figure 1). The data representing the state of an object can be further separated into variables (also called slots) while the behavior is given as a set of methods (also called procedures or functions).

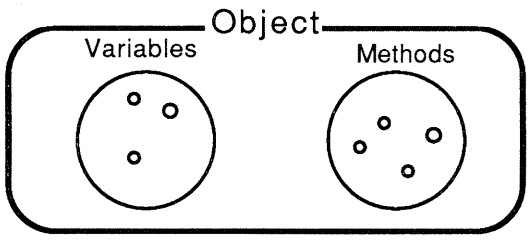


Figure 1. Objects = Variables + Methods

Depending on the context, object-oriented design either focus on the data aspect or the behavior aspect. We speak of procedural abstraction or data abstraction. Despite the fact that there is a grey zone between the two aspects we believe that the distinction is a very important one.

3.1.1. Procedural Abstraction

Frame systems, semantic nets and databases belong to the procedural abstraction family. Slots and their associated values are the primary interest to a user. These values represent declarative knowledge. The procedural part is kept in the background. Accessing and manipulating slots may or may not invoke procedures attached to slots. These types of procedures are often called triggers or demon functions. Typically they are used to maintain the consistence of a system. It is very unlikely that they are invoked by the application directly.

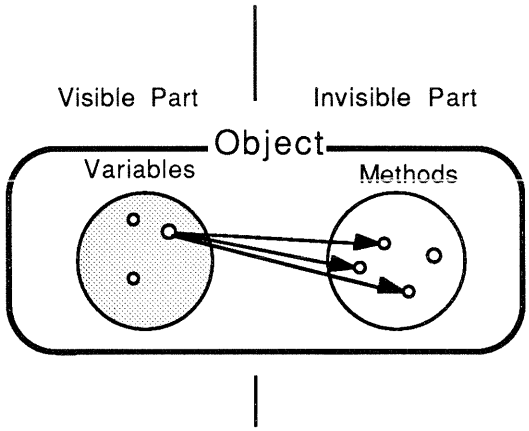


Figure 2. Procedural Abstraction

Figure 2 shows a possible mapping from an object variable to a set of methods. The object variables are known to the application whereas the methods are only invoked indirectly via access to the variables. In this situation methods are often called demon functions.

3.1.2. Data Abstraction

OPUS as well as programming languages like Smalltalk concentrate on the data abstraction aspect. Applications communicate with objects by means of sending messages to them. Each message is matched against a set of so called methods representing the procedural aspect of the object. The data abstraction concept decouples behavior from implementation specific issues regarding internal data structure. In other words, the specific representation of the state of an object should not affect the behavior of the object.

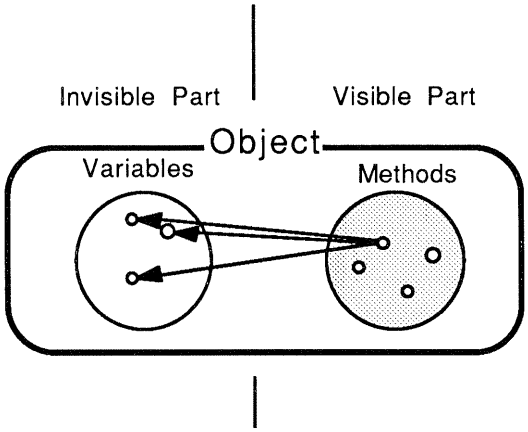


Figure 3. Data Abstraction

In Figure 3 the methods belong to the visible part of the object. Abstraction of data is accomplished by hiding details of the data structure by providing an interface to the object consisting of a set of methods.

3.1.3. Merging the two Aspects

Because the mapping between variables and methods is not generally a bijection, i.e., a variable might invoke several methods and a method might alter several variables, the two aspects cannot be easily transformed to each other. However, both aspects can coexist in one system, e.g., the Units concept of KEE [5]. We believe that these hybrids do not necessarily represent the best of both worlds, instead they exemplify a compromise. Many existing systems implement the alternate aspect on top of the already existing one. This often leads either to performance decreases or the lack of flexibility.

In the KEN expert system shell both aspects are furnished as separate sub-systems. An initial attempt to implement a graphic interface for KEN using the existing frame system exhibited undesirable overhead in terms of execution time cased by unneeded consistency checks. Furthermore, the fact that graphic-objects require to be drawn by modifying a frame slot such that a "draw" demon function gets triggered instead of simply sending a "draw" message to the object, appeared rather absurd to us.

3.2. The Model of OPUS

The OPUS model captures syntactic and semantic issues involved in the definition of object variables and methods. Furthermore, the model reveals a means to express relationships among objects, to handle errors and to debug OPUS programs.

3.2.1. Classes and Instances

A class specifies the procedural aspect as well as the declarative aspect of a set of objects - the instances of the class. Classes are expressed in terms of other classes, i.e., instead of defining functionality in an absolute fashion, object-oriented programming furnishes a general framework to reuse existing building blocks, to compose them together, and to extend them.

OPUS Classes are Real Objects

In contrast to many class-oriented languages including C++ and Flavors, OPUS treats classes as *real objects* [6]. Messages can not only be sent to instances but also to classes. Classes own class variables which can be viewed as shared variables of all instances of this class, and they have class methods. Methods and variables of classes and instances respectively, are completely disjoint, i.e., they can have the same name without disturbing each other.

Creating Named Instances

The create-instance function creates a named instance of a class:

```
create-instance Class-Name &optional Instance-Name                    [Function]
```

where *Class-Name* and *Instance-Name* are symbols. If *Instance-Name* is not provided, then a unique symbol is generated. However, the direct use of create-instance is not recommended. The section "The Built-in Class OBJECT" will show how instances are created using the NEW method. That technique has the advantage of supporting incremental refinement of instances creation. For example, the instantiation of sub

classes of the class called Object frequently includes the initialization of object variables based on additional parameters defined to the extended NEW method.

Creating Anonymous Instances

It is often necessary to create composite objects containing other objects. Especially when the number of these objects is large, then the overhead associated with naming can become significant. The creation of an anonymous instance helps to minimize this overhead. Anonymous instances are not attached to a symbol and the creation of an anonymous instance will not create a send function.

`create-anonymous-instance` *Class-Name* [*Function*]

Create-anonymous-instance will return a handle to the object different to a symbol

```
(create-anonymous-instance 'object) => #<OPUS instance, anonymous instance, a OBJECT>
```

3.2.2. Class Definition

Procedural as well as data structure aspects of objects are covered with the class definition. The following object components are defined in the class:

- class documentation
- super classes
- class/instance variables
- class/instance methods

The class definition in OPUS is expressed with the `create-class` macro:

```
(create-class Class-Name
   [Doc-String]
    [(sub-class-of {Super-Class-Name}*)]
    [(class-variables {Variable-Definition}*)]
    [(class-methods {Method-Definition}*)]
    [(instance-variables {Variable-Definition}*)]
    [(instance-methods {Method-Definition}*)] )
```

where *Class-Name* and *Super-Class-Name* are symbols. The *Doc-String* is used by browsing tools in order to provide a brief description for each class typically explaining the basic semantics of the class. All `create-class` entries are described in detail in the following sections.

A class is created by either evaluating a class definition or loading a file containing uncompiled or compiled class definitions.

3.2.3. Variables

The aggregation of all variables of an object is also called the *state* of the object. For the sake of abstraction, the state of an object is completely hidden from applications. The only way to access object variables is via methods. Any method of a class or sub classes may access the object variables freely. From the viewpoint of a method, all object variables are defined locally to the method.

Defining Variables

Class variables as well as instance variables in OPUS are defined in the `create-class` form. An object *variable definition* has the following syntax:

```
(Variable-Name [Initform [Doc-String]])
```

Names

The variable name is a general Common Lisp symbol.

Initforms

Initforms are required by OPUS in order to determine the initial value of object variables. Note that initforms **are evaluated**, i.e., constant expressions (e.g. a list of symbols) which are not self evaluating need to be quoted. The resulting values of the evaluation are bound to the object variables. Conceptually, initforms are evaluated in parallel; they may not refer to other object variables because they are not bound yet at the time of initform evaluation.

References to global variables within initforms are detected by OPUS; they generate warnings without affecting the functionality of the code[1].

Object variables of object definitions not containing an initform are bound to nil.

Instance Initforms

The instance initforms are evaluated at instance creation time. They **are** allowed to *refer to class variables*. This property can be used to initialize instance variables depending on the state of the variables of its class. An example will illuminate this concept:

```
(create-class instance-variables-refering-to-class-variables
    (class-variables
        (Instance-List nil
            "List of all created instances of this class")))
    (instance-variables
        (My-Index (length Instance-List)
            "A unique number for an instance.")))
```

At the instance creation time `(length Instance-List)` is evaluated. `Instance-List` is a class variable which can be freely accessed in instance initforms and in class methods or instance methods, respectively.

OPUS furnishes a function called `name-of-instance` returning the name of the instance owning the variable. The value returned by this function is only defined during the instance creation. For example the built-in object class Object makes use of this function by assigning its value to a variable called "self".

Class Initforms

Class variables are *persistent* in the sense that their initforms are only evaluated **once**. Re-evaluating the class definition does *not re-evaluate the initforms*. This behavior is similar to the `defvar` concept of Common Lisp.

---

[1] This warnings can be suppressed by assigning a nil value to the variable *Warn-if-Reference-to-free-Variable* exported in the code-walker package.

Similar to instance initforms, class initforms provide a function to determine the name of the created object called `name-of-class`.

Doc-String

It is very important to ease the maintenance of an object system by providing a documentation string for each variable. The intentions, warnings and underlying ideas concerning an object variable are ideally captured in a brief documentation string. The documentation string is utilized by OPUS' inspector and browsing tools.

Accessing and Modifying Variables

Object variables are lexically scoped, i.e., functions being called (not lexically nested) within the body of a method may not access the object variables of the class defining them. However, the scope of instance variables is nested in the scope of their class variables. All instance methods perceive the variables of their class as global. The modification of class variables by one instance is visible to all other instances of the same class. On the other hand, the only way to change instance variables within a class method is to send messages to the corresponding instance object.

Example: A taxi is a thing able to move. Each individual taxi has a meter to measure its mileage. Furthermore, a class defining taxis keeps track of the total mileage of all taxi instances.

```
(create-class Taxi
    "Taxis are vehicles keeping track of their traveled distance."
    (sub-class-of Object)
    (class-variables
        (Total-Mileage 0 "The mileage of all taxis."))
    (class-methods
        (MILEAGE (&optional New-Mileage)"
            in:  &optional New-Mileage
            out: Current-Mileage
            Set and return the total mileage."
            (if New-Mileage
                (setq Total-Mileage New-Mileage)
                Total-Mileage)))
    (instance-variables
        (Mileage 0 "Individual mileage of a taxi."))
    (instance-methods
        (MOVE (Number-of-Miles)"
            in:  Number-of-Miles
            out: New-Number-of-Miles
            Move the taxi a certain distance."
            (incf Total-Mileage Number-of-Miles)
            (incf Mileage Number-of-Miles))
        (MILEAGE (&optional New-Mileage)"
            in:  &optional New-Mileage
            out: Current-Mileage
            Set and return the individual mileage."
            (if New-Mileage
                (setq Mileage New-Mileage)
                Mileage))))
=> TAXI
```

The Taxi class is defined as a sub class of Object in order to inherit all system methods (e.g. the method NEW to create an instance of the class). We create two instances:

```
(taxi 'new 'taxi1) => TAXI1
(taxi 'new 'taxi2) => TAXI2
```

move the taxis around,

```
(taxi1 'move 10) => 10
(taxi2 'move 20) => 20
(taxi1 'move 25) => 35
```

check their individual mileage,

```
(taxi1 'mileage) => 35
(taxi2 'mileage) => 20
```

and the total mileage

```
(taxi 'mileage) => 55
```

Having identical names for instance methods and class methods does not lead to ambiguity. The selection of the appropriate Millage method is determined by the type of the receiver object (class object or instance object). The desired functionality is achieved by incrementing not only the instance variable describing the mileage of each individual taxi, but also by incrementing the class variable denoting the total mileage of all taxi instances being moved.

Adding Variables

In some situations it is helpful to define variables in addition to those defined in the create-class form. As we will see in the Methods section, the so-called unique methods are used to represent implementation specific behavior. They often require additional variables to refine the state representation of an object. We highly recommend not using this kind of variable definition within a specification part. Two variable definition forms are furnished by OPUS:

`definstancevar` *Class-Name* `&rest` *Variable-Definition* [*Macro*]

`defclassvar` *Class-Name* `&rest` *Variable-Definition* [*Macro*]

Evaluation of either of these forms causes the replacement of the own variables V_o with the result of the extended union[1] of the current own variables and the variables defined with the definstancevar form or the defclassvar form V_{new}:

$$V_o := V_o \tilde{\cup} V_{new}$$

Furthermore, the set of total variables V_{tot} of the modified class and all subclasses of it will be updated:

$$V_{tot} = V_o \tilde{\cup} V_{in}$$

where V_{in} is the set of inherited variables. Shadowing existing object variables by (re)evaluating create-class, definstancevar, and defclassvar forms will overwrite initforms and documentation strings of existing variables. However, the value of existing class variables will not be modified by these operations. Object variables, and methods as well for that matter, get never never deleted unless they get explicitly removed from a class by the user.

---

[1]The extended union is described in detail in the Inheritance section. It is represented by the symbol $\tilde{\cup}$

3.2.4. Methods

Methods often operate on the state of an object by exerting side effects to the variables of the object. They also may be of more functional character computing new values based on the current state of the object and simply returning them to the sender of a message.

In the following sections we use the distinction of *method specification* and *method implementation*. The method specification consists of the parameter (lambda) list and the documentation string of the method, whereas the method implementation consists of the method body.

Each method is either defined as part of the create-class form utilizing the class-methods and instance-methods slots,

```
(Name Lambda-List {Declaration | Doc-String}* {Form}*)
```

or in a separate defmethod form:

```
(defmethod Name Lambda-List
    (class-method-of | instance-method-of Class-Name)
    {Declaration | Doc-String}*
    {Form}*  )

=> Type-and-Name-of-Method
```

where `Name` and `Class-Name` are symbols and `Lambda-List` is conforming to the lambda list definition of Common Lisp.

The OPUS method classification scheme breaks methods up into three types depending on the degree of implementation dependence: *generic*, *specific*, and *unique*.

Generic Methods

All methods defined within a create-class form *providing a body*[1] are of type *generic*. Generic methods are supposed to be completely implementation invariant. The example below of a Number class provides a generic method called '+'.

```
(create-class Number
    (sub-class-of Object)
    (instance-variables
        (X 0 "A scalar number."))
    (instance-methods
        (+ (Delta-X) "
            in: Delta-X. out: New-X.
            Increment the number."
            (incf X Delta-X))))
```

The documentation string exhibits the input and output parameter and explains the semantics of the method.

A generic method can be *shadowed* by redefining the method using the defmethod form. The lambda list of the defmethod form needs not by any means to match the one defined in the create-class form. The method documentation string is overwritten with the new one, i.e., all inspectors and browsing tools exploit the documentation string provided in the defmethod form. In our Number example we shadow the '+' method:

---

[1]The documentation string is not considered part of the body.

```
(defmethod + (Dx)
    "Slightly changed method implementation."
    (setq X (+ X Dx)))

=> "GENERIC method: +"
```

Specific Methods

Specific methods consist of the *method specification* (a bodiless method definition) in the create-class form and a separate *method implementation* defined with a defmethod form. In contrast to generic methods, the evaluation of a defmethod form defining a specific method does not overwrite the documentation string and the lambda list definition. The documentation string employed by inspectors and browsers always remains the one defined in the create-class form.

The parameter list of the defmethod form is matched against the parameter list of the method definition within the `create-class` form. This OPUS feature is especially suited to maintain large object-oriented systems.

Method definitions being part of the create-class form of specific methods are slightly restricted to:

```
(Name Lambda-List [Doc-String])
```

where *Name* is a symbol and *Lambda-List* conforms to the lambda list definition of Common Lisp.

In order to introduce a specific '*' method in our Number example we first have to extend our class definition with a specification for the '*' method:

```
(create-class Number
    (sub-class-of Object)
    (instance-variables
        (X 0 "A scalar number."))
    (instance-methods
        (+ (Delta-X)"
            in: Delta-X. out: New-X.
            Increment the number."
            (incf X Delta-X))
        (* (Times)"
            in: Times. out: New-X
            Multiply the current number with Times."
            )))
```

Furthermore, we state the method implementation:

```
(defmethod * (Multiplier)
    (instance-method-of Number)
    (setq X (* Multiplier X)))

=> ":SPECIFIC method: *"
```

Despite the fact that different symbols have been used for the parameter in the lambda list of the '*' method in the defmethod form and the create-class form respectively, the two lambda lists still match.

The *semantics of matching* create-class and defmethod lambda lists in general is defined as follows:

- both lambda lists must have the same number of elements.
- the lexicographic orders of the lists have to be identical, e.g., all lambda list keywords must match exactly.
- the names of parameters may vary.
- if a parameter of a lambda list is a symbol then the corresponding parameter must also be a symbol.
- if a parameter of a lambda list is given as a list (e.g. &optional parameters having an initform) then the corresponding parameter must also be a list having the same number of elements.

Examples of valid create-class defmethod lambda list pairs:

```
(X Y &optional Z),                          (Parameter-1 Parameter-2 &optional Parameter-3)
(A &rest B &key (C (f Q P) Bound-C)),                  (U &rest V &key (W (g O) B-W))
```

invalid examples:

```
(X &rest Y &key Z),     (P1 &rest P2 P3) different lambda list keyword structure
(U &optional (V (f Q) S)), (I &optional (J (f Q))) wrong number of elements of the
&optional argument.
```

Unique Methods

Some implementations require methods in addition to the ones defined in the create-class form. These methods are only used internally and do not have any impact on the definition of a class. A typical use of these methods is to provide extra features for certain implementation platforms. Still and all, users are warned not to use unique methods too frequently. Large unique/specific methods ratios are often good indicators of a design flaw. In order to gain some control over unique methods, OPUS provides a switch to enable and disable their creation:

```
allow-unique-methods &optional (Disable t Suppliedp)                    [Function]
```

Calling allow-unique-methods without the Disable argument simply returns the current state of the enable/disable unique methods flag. Otherwise, the flag is set. The Disable argument is of boolean type (nil or non-nil).

In our Number example we subjoin a unique method called '/'

```
(defmethod / (Divisor)
    (instance-method-of Number)
    "A method not mentioned in the create-class form."
    (setq X (/ X Divisor)))

=> "UNIQUE method: /"
```

Specification and Implementation Part

OPUS supports the model of having a separate specification and implementation part. The interface between both parts is embodied with the lambda list (argument list) of the method definition. These parts are typically kept in separate files and are also created by separate programming teams. OPUS matches each implementation part with its specification. Similar concepts can be found in languages like Ada and Modula-2 [7]. Per contra, these languages provide this concepts as a means of abstraction but they do not support inheritance.

The separation of a method into a specification part and an implementation part provides a powerful abstraction. The specification part furnishes a means to capture invariant, constant and organic properties of the behavior of objects. On the other hand, the implementation part is the place to cope with changing, temporary and accidental object properties.

We use an example of the graphics domain to illuminate this abstraction mechanism. The KEN expert system shell provides a simple graphic editor similar to the MacDraw package available for Macintoshes. KEN is delivered for many different hardware platforms such as VaxStations, Apollos, IBM-PCs, and Macintoshes. Unfortunately the Common Lisp standard does not include a generally accepted windowing package. Each hardware provides its own windowing software and unique interfaces between the windowing

software and Common Lisp. For the sake of simplicity we focus our attention on one graphic object class called Circle. From the data structural aspect, we require for our design that a circle is described with a coordinate of its mid point, and a radius. Furthermore, because graphic objects have to be attached to a window we also need a link from the circle object to a window. The behavior of the object is limited to attaching a circle to a window and drawing the circle. Consider the following class definition belonging to the **specification part**:

```
(create-class Circle
    (sub-class-of Object)
    (instance-variables
        (X 0 "The horizontal coordinate of the circle.")
        (Y 0 "The vertical coordinate of the circle.")
        (Radius 0 "Circle radius.")
        (Window nil "Link from circle to a window."))
    (instance-methods
        (ATTACH-TO-WINDOW (A-Window)"
            in: Window.
            Attach the circle instance to a window."
            (setq Window A-Window))
        (DRAW ()"
            Draw the circle in the window.")))
```

Note that within this specification part no assumptions are made concerning what windows really are, and how a circle is physically drawn. Because of its generality, the attach-to-window method is defined as a generic method providing a body.

In the worst case, one has to provide as many different implementations as hardware platforms are supported. In the real world the worst case is actually very likely. One way to cope with different implementations is to apply Common LISP's read macros[1] [8]. However, because of the substantial differences in windowing packages the granularity of necessary read macros is typically very large. Many times, entire method bodies have to be encapsulated for each implementation with a read macro. This intensive use of read macros leads to severe problems:

- Source files grow by a factor equal to the number of underlying packages supported.
- The maintainability suffers; it is impossible to work with multiple implementation teams on the same piece of software in parallel.

Our experience manifested that there is a high risk that an implementation team accidentally (or sometimes even deliberately) changes parts of the implementation not belonging to them. After a small number of modifications by independent teams the implementation tends to be in complete disorder.

It is of great advantage to keep implementation invariant elements in a separate file, the specification part, and to spread each implementation part to its own file. In our example the **implementation part** specific to the Macintosh could look like:

```
(defmethod DRAW ()
    (instance-method-of Circle)
    "Draw the circle."
    (mac-toolbox:draw-circle Window X Y Radius))

=> "SPECIFIC method: DRAW"
```

---

[1] #+ and #- read time conditionals

Even for this simple application the DRAW method could be much more complex, e.g., it could request to transform the coordinates first before calling an appropriate function to draw circles, or it could request to initialize the window first or to refresh it after the circle has been drawn. All these details, however, are very implementation specific and therefore do not belong to the specification part.

The table below shows what the different types of methods define, and whether they belong to the specification or the implementation part:

	create-class	**defmethod**
generic	specification + implementation *specification part*	shadowed create-class implementation *specification part*
specific	specification *specification part*	implementation *implementation part*
unique	- -	specification + implementation *implementation part*

For example, a generic method defined using the defmethod form is shadowing the one defined in the create-class form.

Evaluation Order and Consistency

The effects of evaluating and reevaluating create-class forms and defmethod forms is analogous to those of evaluating a defun form. If, for example, a method is defined which is identical in name, class and class type to an existing one, then the old one will be overwritten with the new one.

Reevaluating a create-class form will not only overwrite all methods previously defined in the create-class form, but it will also overwrite the generic methods previously defined using defmethod for this. Specific methods behave differently: method-specification and method-implementation do not overwrite each other, i.e., reevaluating a create-class form containing the method specification of a specific method will not delete the method implementation defined in a defmethod form.

There are basically six types of nonexclusive changes to a create-class form:

- **adding variables**: the variables are added to the class and all its sub classes. The initforms of added[1] class variables are evaluated. Existing instances are not updated.
- **removing variables**: no effect. Variables defined before continue to exist.
- **adding methods**: the methods are added to the method dictionary of the class, they are inherited by all sub classes.
- **removing methods**: no effect. Methods defined before continue to exist.

---

[1]Initforms of already existing variables do not need to be evaluated.

- **adding super classes**: the class precedence list[1] is redetermined. Existing instances are not updated with newly inherited variables.
- **removing super classes**: the class precedence list is redetermined. Existing instances are not updated; they keep possessing variables which are possibly no longer inherited.

3.2.5. Sending Messages to Objects

Defining variables and methods is of little use as long as there is no way to invoke methods. The technique exploited to invoke methods is called *message sending*. The actors involved in the sending of a message are the *sender* object and the *receiver* object. Each message consists of a *selector* and *arguments*. The selector is matched against the *method dictionary* of the receiver. Is there is no match but the receiver has super classes, then the dictionaries of the super classes will be searched for a method matching the selector. The search starts at the most specific super class and works towards the least specific one. In case that none of the super classes provides a matching method an error is signaled.

In OPUS class objects and named instance objects are function objects, i.e., the symbol function of symbols representing the name of objects are bound to lambda expressions. Sending a message has the syntax of an ordinary function call:

```
(Receiver Selector {Argument}*)
```

In cases where the receiver needs to be an evaluated entity apply or funcall can be bestowed:

```
(funcall Receiver Selector {Argument}*)
(apply Receiver Selector {Argument}* More-Arguments-List)
```

The fact that objects act like functions can be brought into play to trace messages using the Common Lisp trace macro.

A symbol can only be attached to either a class object or an instance object. Depending on the type of object to which a receiver symbol is attached, a message will either invoke a class method or an instance method. Consider the earlier mentioned Number class. Assume that the class Object provides a message called NEW to create instances. Because Number is defined as a sub class of Object it will inherit[2] the NEW class method. The following script makes use some of the introduced concepts:

```
(number 'new 'number1) => number1
  Create an instance called number1. Note that the receiver called
  number is a class. Therefore, the selector, new, is matched
  against the class methods of the Number class. The initform for x
  has been evaluated and has bound x to zero.

(number1 '+ 13) => 13
  Because number1 is an instance the '+' selector of this message
  is matched against the instance methods of the Number class. x
  assumes a new value 13 which is also returned by the method

(number1 '+ 17) => 30
   x is set to the new total of 30.
```

---

[1]The class precedence list is discussed in detail in the "Multiple Inheritance" section.

[2]For information concerning inheritance see the section on inheritance.

Messages are sent to **anonymous** instances using either the function apply-instance-message, or the macro aim:

`apply-instance-message` *`Instance Selector Arguments`* [Function]

`aim` *`Instance Selector`* `&rest` *`Arguments`* [Macro]

To create a taxi from our taxi example and make it move we do the following:

```
(setq A-Taxi (create-anonymous-instance 'taxi))
=> #<OPUS instance, anonymous instance, a TAXI>

(apply-instance-message A-Taxi 'move '(10)) => 10

(aim A-Taxi 'move 10) => 20
```

3.2.6. Relationships among Objects

In order to support concepts known from semantic networks like generalization, aggregation and grouping relationships among objects [9], these relationships need to be expressed explicitly. In OPUS we have three types of relationships:

- relationships among classes
- relationships among instances
- relationships between instances and classes.

SUB-CLASS-OF

The specializing concept of semantic networks, which is a very important relationship among classes, is embraced by the *sub-class-of* link in OPUS. A class may have multiple super classes. The primary purpose of the sub-class-link is to support inheritance. The immediate super classes of a class are defined in the sub-class-of slot of the create-class form.

INSTANCE-OF

Each instance is an exemplar of its class. The link between an instance and its class is called the *instance-of* link.

User Defined Relationships

Any kind of user defined relationship between objects can be specified with instance variables or class variables respectively. A sample of a relationship between instances can be found in the section "An Example: Grouping of Graphic Objects".

3.2.7. Inheritance

Instead of defining a set of classes from scratch, classes may be defined incrementally by reusing existing classes and refining them. A new class C2 may be defined by saying something like "..C2 is essentially like C1 except that..". Inheritance provides a means to express to incrementally define new software artifacts.

A class specializing another class refining any aspect of it is a sub class. In terms of semantic nets, a sub class **is-a** class. The is-a relation is transitive, i.e., if A is-a B and B is-a C then A is-a C.

A subclass inherits all properties[1] of the data and the behavioral aspects of its super class(es). Sub classes basically have three ways to render the functionality of their super classes:

- introduce new variables and/or methods
- shadow existing variables and/or methods
- extend existing method in their functionality

Shadowing is achieved by defining a variable or a method in a sub class that has the same name as one of its super classes. Shadowing does overwrite a definition of a property completely rather than merge the property being shadowed with the new one. If we create a sub class called Number defining an own variable x, then the initform and the doc-string of the super class are replaced with the ones from the sub class.

The concept of extending properties is limited to methods. A method of class is extended by reusing methods with identical names from super classes.

Single Inheritance

In a single inheritance scheme a class is allowed to have zero or one *immediate super class*, which in turn may have one immediate super class again. A class inherits all properties of its super classes, i.e., properties defined in any super class of a class also belong to the class. The total set of properties of a class is the union of its own properties and the ones inherited from its super classes. The union operation required is not symmetrical. That is, if the two to-be unified property sets share properties with the same name, then the ones from the more specific class have precedence. In cases like this we say that the properties of the more specific class shadow the properties from the less specific class. We call this union operation of own properties and inherited properties the *extended union*:

$$P_{tot} = P_o \tilde{\cup} P_{in}$$

where P_{tot} is the set of all properties of a class, P_{in} is the set of inherited properties and P_o is the set of own properties defined in the class. The set of inherited properties of a class is identical with the set of total properties of its immediate super class. We define the extended union operator as:

$$U \tilde{\cup} V = U \cup \{v \mid (v \in V) \wedge (\forall u)[(u \in U) \wedge (\mathrm{name}(u) \neq \mathrm{name}(v))]\}$$

In other words, the extended unification of a set U with a set V is equal to the unification of U with the subset of V, consisting only of elements having different names than all members of U. Note that this extended notion of a union operation is only symmetric in case where U and V are disjoint sets.

Consider the following set of class definitions:

```
(create-class C1
    (instance-variables
        (X 0 "X of c1")
        (Y 1 "Y of c1")
        (Z 2 "Z of c1"))
    (instance-methods
        (+X () "+X of C1" (incf X))
```

---

[1]In this context we use the term properties as a short form for class/instance variables and class/instance methods of a class.

```
        (+Y () "+Y of C1" (incf Y))))

(create-class C2
    (sub-class-of C1)
    (instance-variable
        (Z 3 "Z of C2"))
    (instance-methods
        (+Z (DX) "+Z of C2" (incf Z DZ))))

(create-class C3
    (sub-class-of C2)
    (instance-variables
        (Y 4 "Y of C3"))
    (instance-methods
        (+X (DX) "+X of C3" (incf X DX))))
```

After applying unification to all classes we get the following schemes:

Class: C1	Variables	Methods
$P_{inherited}$	{}	{}
P_{own}	{(X 0 "X of C1"), (Y 1 "Y of C1"), (Z 2 "Z of C1")}	{(+X () "+X of C1" ...), (+Y () "+Y of C1" ...)}
P_{total}	{(X 0 "X of C1"), (Y 1 "Y of C1"), (Z 2 "Z of C1")}	{(+X () "+X of C1" ...), (+Y () "+Y of C1" ...)}

Class: C2	Variables	Methods
$P_{inherited}$	{(X 0 "X of C1"), (Y 1 "Y of C1"), (Z 2 "Z of C1")}	{(+X () "+X of C1" ...), (+Y () "+Y of C1" ...)}
P_{own}	{(Z 3 "Z of C2")}	{(+Z (DX) "+Z of C2" ...)}
P_{total}	{(X 0 "X of C1"), (Y 1 "Y of C1"), (Z 3 "Z of C2")}	{(+X () "+X of C1" ...), (+Y () "+Y of C1" ...), (+Z (DX) "+Z of C2" ...)}

Class: C3	Variables	Methods
$P_{inherited}$	{(X 0 "X of C1"), (Y 1 "Y of C1"), (Z 3 "Z of C2")}	{(+X () "+X of C1" ...), (+Y () "+Y of C1" ...), (+Z (DX) "+Z of C2" ...)}
P_{own}	{(Y 4 "Y of C3")}	{(+X (DX) "+X of C3" ...)}
P_{total}	{(X 0 "X of C1"), (Y 4 "Y of C3"), (Z 3 "Z of C2")}	{(+X (DX) "+X of C3" ...), (+Y () "+Y of C1" ...), (+Z (DX) "+Z of C2" ...)}

It is not possible for any system design to furnish the enormous flexibility necessary to cope with all possible future system enhancements. Systems need to be easily extendible. The need to modify existing code with conventional programming techniques in order to extend the functionality of a system leads to

enormous complexity. Object-oriented programming provides a means to represent the structure of a system design explicitly. Intermediate design stages do not get lost, they are provided as so called *mixin classes*. These classes are exploited to provide basic functionality in order to create system extensions later on by mixing these classes together to more concrete classes. In contrast to concrete classes mixin classes are merely provided for the sake of extendibility; no instances are created of these classes.

Multiple Inheritance

Multiple inheritance is the generalization of single inheritance allowing each class to have more than just one immediate super class. This technique is often employed to combine the functionality of classes by merging their properties.

Class Graphs

A class can be viewed as a vertex in a graph of all classes. The edges of the graph are given by the directed sub-class-of links. The *super class graph* is a sub graph of this graph containing only super classes of a class including the class itself. This super class graph is specific to each class. OPUS linearizes the super class graph while maintaining topological order. This linearized super class graph, called the *class precedence list* is associated with each class. Two criteria implying a partial order are applied to arrange the class precedence list:

i) The lexicographic order of classes within the sub-class-of definition slot of the create-class form is preserved as long as it does not contradict with ii).

ii) Every class always precedes it super classes.

Example class graph:

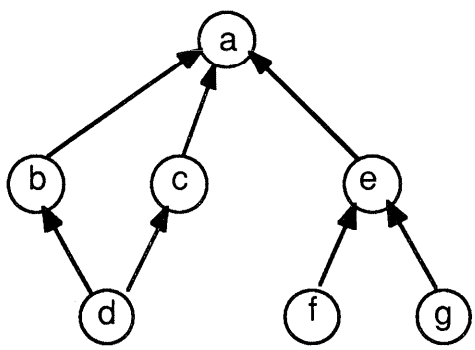


Figure 5. Class Graph

The lexicographic order within sub-class-of slot of the class definition is reflected in the depiction of the graph, e.g., class b precedes class c. Then the super class graph for class d is a sub graph of the class graph:

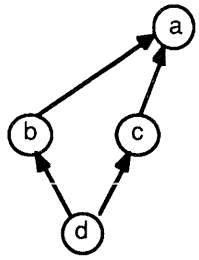


Figure 6. Super Class Graph of Class d

The super class graph of d is linearized by applying a topological sort starting at class d. We get the class precedence list **(d, b, c, a)** for class d. In general, applying i) and ii) to a class graph is guaranteed to lead to an unique total order.

Another example elaborates the relationship of the class definition and the class precedence list:

```
(create-class A)

(create-class B
    (sub-class-of A))

(create-class C
    (sub-class-of B A))

(create-class D)
    (sub-class-of B))

(create-class E
    (sub-class-of D C))
```

Figure 7 shows the set of classes as a graph including the class precedence list for each class:

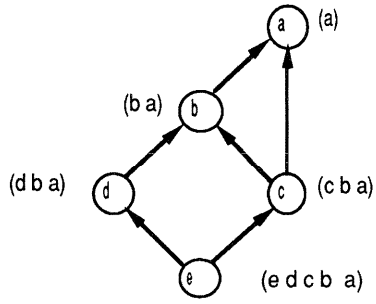


Figure 7. Class Precedence Lists

If we denote the class precedence list as (C1, C2, ..., Cn) then the set of total properties of a class C1 is given as:

$$P_{tot,C_1} = P_{o,C_1} \tilde{\cup} P_{o,C_2} \tilde{\cup} \ldots \tilde{\cup} P_{o,C_n}$$

The criterion mentioned might contradict each other. Criteria ii) outweighs criteria i) to resolve cases like the one illustrated in Figure 8.

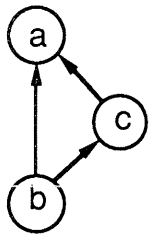


Figure 8. Class Graph leading to Contradictions

The lexicographic order implyed by Figure 8 suggest the class precedence list (b, a, c). However, class a is a super class of class c. To satisfy ii) c has to precede a, and b has to precede c and a. Therefore, the only valid order is (b, c, a).

An Example: Grouping of Graphic Objects

Consider building a simple MacDraw-like graphic editor. A picture consists of a set of objects. Each object shall be an instance of a sub class of a class called Graphic-Object. In order to select, drag and update a graphic object, we need a means to describe the area they cover. For simplicity we chose a rectangular shape completely containing the two dimensional graphic object. We call this rectangular area the *foundation area.* First, we define the Graphic-Object class which is a mixin class, i.e., no instances are created of this class, it is only employed to furnish a basis for refined classes. Obviously we cannot define a method to draw a generic Graphic-Object as we have no associated depiction for it. However, knowing about the foundation area of the object we can define a UNDRAW method.

```
(create-class Graphic-Object
    "The root class of all graphic object classes."
    (sub-class-of Object)
    (instance-variables
        (X-Pos 0 "Horizontal position.")
        (Y-Pos 0 "Vertical position.")
        (Foundation-Area (list 0 0 0 0)
            "List of <x, y> bottom-left, <x, y> top-right."))
    (instance-methods
        (FOUNDATION-AREA () "Return the foundation area."
            Foundation-Area)
        (UNDRAW () "
            Clear rectangular area defined by foundation-area."
            (format t "undraw ~S: ~S~%" Self Foundation-Area))))
```

Every Graphic-Object has a reference point (X-Pos, Y-Pos) which is independent of the foundation area. Instead of physically clearing the area on the screen defined with the foundation area, the UNDRAW method simply prints out the foundation area.

Two sub classes of Graphic-Object, Circle and Rectangle, are defined below. Each of them defines a DRAW method and a method to set-up the properties of the object:

```
(create-class Circle
    "A graphic object class defining draftable circle objects."
    (sub-class-of Graphic-Object)
    (instance-variables
        (Radius 0 "Circle radius"))
    (instance-methods
        (DEFINE (X Y R)"
            in: X, Y, R
            Define the circle parameters."
            (setq X-Pos X Y-Pos Y Radius R)
            (setq Foundation-Area
```

```
                (list (- X R) (- Y R) (+ X R) (+ Y R))))
        (DRAW ()"
            Draw the circle."
            (format t "draw circle: X ~D, Y ~D, radius ~D~%"
                X-Pos
                Y-Pos Radius))))
=> CIRCLE


(create-class Rectangle
    "A graphic object class defining draftable rectangle objects."
    (sub-class-of Graphic-Object)
    (instance-methods
        (DEFINE (X1 Y1 X2 Y2)"
            in: X1, Y1, X2, Y2
            Define the rectangle parameters."
            (setq X-Pos X1 Y-Pos Y1)
            (setq Foundation-Area (list X1 Y1 X2 Y2)))
        (DRAW ()"
            Draw the rectangle."
            (apply #'format t
                "draw rectangle: X1 ~D, Y1 ~D, X2 ~D, Y2 ~D~%"
                Foundation-Area))))
=> RECTANGLE
```

Note that the DEFINE methods are also setting up the foundation areas. Now we define a Group class to deal with agglomerations of objects. A group is capable of broadcasting messages to its members.

```
(create-class Group
    "A group consists of arbitrary many elements."
    (sub-class-of Object)
    (instance-variables
        (Elements nil "List of elements."))
    (instance-methods
        (ADD-ELEMENT (Element)"
            in: Element. out: Element-List.
            Add a new element."
            (push Element Elements))
        (BROADCAST (Selector &rest Parameters)"
            in: Selector + Parameters = Message.
            The message is propagated to all elements of the group."
            (dolist (Element Elements)
                (apply Element Selector Parameters)))))
=> GROUP
```

Finally, we orchestrate the Graphic-Object class and the Group class by merging them together. Graphic-Object-Group instances also have a foundation area which is given as the geometrical union of the foundation area of its elements. The DRAW and the UNDRAW message are forwarded to the elements of the group by making use of the BROADCAST method:

```
(create-class Graphic-Object-Group
    (sub-class-of Graphic-Object Group)
    (instance-methods
        (ADD-ELEMENT (Element)"
            in: Element. out: Element-List
            Extension of Group ADD-ELEMENT: set-up foundation area."
            (setq Foundation-Area
                (if Elements
                    (area-union
                        Foundation-Area
                        (funcall Element 'Foundation-Area))
                    (funcall Element 'Foundation-Area)))
            (super 'add-element Element))
        (DRAW () "Broadcast to elements."
            (self 'broadcast 'draw))
        (UNDRAW () "Broadcast to elements."
            (self 'broadcast 'undraw))))
=> GRAPHIC-OBJECT-GROUP
```

The union of the foundation areas is determined incrementally, i.e., adding a new element to the group also updates its foundation area. We define the union of two rectangular shapes to be the smallest rectangle that can entirely enclose the rectangular shapes. Within the body of the ADD-ELEMENT method we call the area-union function defined below, returning the union of the current foundation area and the one from the new element.

```
(defun AREA-UNION (Area1 Area2)
  (list
   (min (first Area1) (first Area2))
   (min (second Area1) (second Area2))
   (max (third Area1) (third Area2))
   (max (fourth Area1) (fourth Area2))))
```

Having defined all classes, we can now create instances and initialize them:

```
(rectangle 'new 'rectangle1) => RECTANGLE1
(rectangle1 'define 6 7 8 9) => (6 7 8 9)
```

In our example the DEFINE method returns the foundation area.

```
(circle 'new 'circle1) => CIRCLE1
(circle1 'define 2 3 1) => (1 2 3 4)
```

A circle with radius 1 with its center at position 2,3 has a foundation area of (1, 2, 3, 4).

```
(circle1 'draw)
draw circle: X 2, Y 3, radius 1
=> NIL
```

The DRAW method simply prints out the graphic-object parameters.

```
(rectangle1 'draw)
draw rectangle: X1 6, Y1 7, X2 8, Y2 9
=> NIL
```

The foundation area is queried with the FOUNDATION-AREA message:

```
(circle1 'foundation-area) => (1 2 3 4)
```

We create a first Graphic-Object-Group instance called gop1,

```
(graphic-object-group 'new 'gop1) => GOP1
```

and add the circle1 as an element to it.

```
(gop1 'add-element 'circle1) => (CIRCLE1)
```

The Graphic-Object-Group instance assumes the foundation area of circle1.

```
(gop1 'foundation-area) => (1 2 3 4)
```

The DRAW message sent to gop1 is forwarded to circle1.

```
(gop1 'draw)
draw circle: X 2, Y 3, radius 1
=> NIL
```

We also annex the rectangle1 as an element to gop1.

```
(gop1 'add-element 'rectangle1) => (RECTANGLE1 CIRCLE1)
```

Gop1's foundation area is replaced with the union of its old foundation area and the one from the rectangle.

```
(gop1 'foundation-area) => (1 2 8 8)
```

If we now send the DRAW message again, then rectangle1 and circle1 are printed out.

```
(gop1 'draw)
draw rectangle: X1 6, Y1 7, X2 8, Y2 9
draw circle: X 2, Y 3, radius 1
=> NIL
```

Also the UNDRAW message is forwarded.

```
(gop1 'undraw)
undraw RECTANGLE1: (6 7 8 9)
undraw CIRCLE1: (1 2 3 4)
=> NIL
```

Group elements are not restricted to be rectangles or circles, another group can also be an element of a group. A second group (gop2) is created containing rectangle2 as an element.

```
(graphic-object-group 'new 'gop2) => GOP2
(rectangle 'new 'rectangle2) => RECTANGLE2
(rectangle3 'define 10 10 11 11) => (10 10 11 11)
(gop2 'add-element 'rectangle2) => (RECTANGLE2)
```

Finally, we add gop2 as the new element to gop1 and send the DRAW message to gop1.

```
(gop1 'add-element 'gop2) => (GOP2 RECTANGLE1 CIRCLE1)
(gop1 'draw)
draw rectangle: X1 10, Y1 10, X2 11, Y2 11
draw rectangle: X1 6, Y1 7, X2 8, Y2 9
draw circle: X 2, Y 3, radius 1
=> NIL
```

The Graphic-Object-Group class is a simple example of how to exploit objects to define recursive schemes. Multiple inheritance is a powerful technique allowing the definition of disjoint concepts on a general level in order to combine them arbitrarily later on to sub classes.

Semantic Issues

Classes merged to a new sub class are typically not disjoint, i.e, certain properties get shadowed depending on the lexicographic order of the class names within the sub-class-of slot of the create-class form. The property name remains, but other elements of a property like initform or parameter list get changed. In practice these effects are not a primary problem. A much more severe problem is the *semantic overloading* of properties.

Consider a case where two classes called A and B provide a instance variable with the name x. It might be that x has completely different semantics in the two classes. If we now merge these classes to a new class C, then we might have a serious problem. Furthermore, assume that class A has precedence over class B. All methods provided by A treat x from the semantic viewpoint of A. If B provides equally named methods, then they are shadowed by A. However, there may also be methods only provided by B, which in turn treat the variable x from the semantic viewpoint of B. Applying these merged methods usually leads to inconsistencies.

It is a good practice to choose specific names for variables which somehow reflect their semantics instead of using very generic names. The merging of object classes is a complex operation. It often appears to be tempting to merge classes. However, this operation demands a lot of skill from the designer in order to prevent undesired side effects.

3.2.8. Pseudo Objects

Pseudo objects provide a means to send messages to the current active object and to invoke shadowed methods.

Self

Sending a message to the Self object is identical to sending a message to the current message receiver:

```
self Selector &rest Parameters                                [Instance/Class Object]
```

The use of this object is only valid within a method.

Example: A class called Number already provides a method '+'. A sub class shall be defined furnishing a new method '*' being expressed in terms of '+':

```
(create-class Number
    (sub-class-of Object)
    (instance-variables
        (Number 0 "The actual number."))
    (instance-methods
        (+ (DX) (incf Number DX))))

=> NUMBER

(create-class Multipliable-Number
    (sub-class-of Number)
    (instance-methods
        (* (Times)
            (let ((Old-Number Number))
                (dotimes (I (1- Times) Number)
                    (self '+ Old-Number))))))

=> MULTIPLIABLE-NUMBER
```

Then we create a Multipliable-Number instance,

```
(multipliable-number 'new 'mn1) => MN1
```

and add the number 5.

```
(mn1 '+ 5) => 5
```

In order to make the messages to the mn1 instance visible we trace it using the ordinary Common Lisp trace macro.

```
(trace mn1 self) => (MN1 SELF)
```

Now we send the '*' message,

```
(mn1 '* 4)
```

and get the following output from the tracer:

```
 Calling (MN1 * 4)
  Calling (SELF + 5)
  SELF returned 10
  Calling (SELF + 5)
  SELF returned 15
  Calling (SELF + 5)
  SELF returned 20
 MN1 returned 20
=> 20
```

Super

It is often desirable to make use of shadowed methods in order to extend them. Extending methods rather than overwriting them is the basic concept provided by OPUS for incremental specification.

Sending a message to the pseudo object super is similar to sending the message to the class first specified in the sub-class-of slot of the create-class form. However, all possible side effects take place in the original

receiver of the message sending a message to super. The syntax is very similar to the one of the self pseudo object:

```
super Selector &rest Parameters                                [Instance/Class Object]
```

Example: Assume the following scenario. A sub class to Object has to be defined keeping track of the instantiation time of its instances. The Object class already provides a method called NEW which appears to be a good candidate to be extended.

```
(create-class Time-Stamped-Object
    "The creation time of every instance is recorded in the class."
     (sub-class-of Object)
    (class-variables
        (Name-Time nil "property list of (<Name> <Time>) pairs"))
    (class-methods
        (NEW (&optional Instance-Name)"
            in:  &optional Instance-Name.
            Extension of Object NEW. Record the instance creation times."
            (let ((Name (super 'new Instance-Name)))
                (setf (getf Name-Time Name) (get-universal-time))
                Name))
        (CREATION-TIME (Name)"
            in:  Name.
            out: Time.
            Return the time of creation."
            (decode-universal-time (getf Name-Time Name)))))
```

The NEW Instance-Name message sent to the super pseudo object creates an instance and returns the name of it. The creation time of the two instances created is stored in the Name-Time class variable:

```
(time-stamped-object 'new 'tso1) => TSO1
(time-stamped-object 'new 'tso2) => TSO2
```

This time is read by sending the creation-time message to the class. It returns the nine decoded universal time values specified in Steele, page 445 [8]:

```
(time-stamped-object 'creation-time 'tso1)

=> 48
; 16
; 18
; 12
; 6
; 1989
; 0
; NIL
; 5

(time-stamped-object 'creation-time 'tso2)

=> 49
; 16
; 18
; 12
; 6
; 1989
; 0
; NIL
; 5
```

The tso2 instance was created one second later then the tso1 instance.

The super pseudo object can be called as many times as required. Furthermore, the message being sent to the super object does not necessarily equal the one sent to the receiver object invoking super.

3.2.9. Initialization of Objects

Like ordinary data structures, objects have to be initialized in order to start from a defined state. During the creation of objects the *init message* is sent to them. Sub classes of the Object class inherit an init method which simply returns the name of the created object.

In contrast to the initforms of object variables, which are computing values independent of other object variables, the init method typically handles the computation of values dependent on multiple object variables. Furthermore, the init method is responsible for returning the value of the create-class form for the creation of classes, and the NEW message for the creation of instances respectively. The init method is invoked **after** the initforms of the object variables have been evaluated.

Regardless of the fact that class variables are persistent (their initforms are not reevaluated when the create-class form is reevaluated), the class init method is invoked each time the create-class form is evaluated.

Example:

```
(create-class Circle
    (sub-class-of Object)
    (instance-variables
        (Radius 5.0 "Initial radius.")
        (Perimeter nil "Perimeter"))
    (instance-methods
        (PROPERTIES ()"
            out: list of radius and perimeter.
            Return the properties of the circle."
            (list Radius Perimeter))
        (INIT ()"
            out: Perimeter of circle.
            Is called during the instance creation."
            (setq Perimeter (* 2 pi Radius)))))
=> CIRCLE
```

The creation of a circle instance is setting up the radius and the perimeter:

```
(circle 'new 'circle1) => CIRCLE1
(circle1 'properties) => (5.0 31.41592653589793)
```

Instead of having a fixed initial value for the radius, the INIT method can be exploited in conjunction with the class method NEW to define a radius at instance creation time and to derive the perimeter of the circle:

```
(create-class Circle
    (sub-class-of Object)
    (class-methods
        (NEW (&optional Instance-Name Radius)"
            in:  Instance-Name, Radius.
            out: Perimeter
            Create a circle, define its radius and its perimeter."
            (funcall (create-instance Self Instance-Name) 'init Radius)))
    (instance-variables
        (Radius nil "Radius.")
        (Perimeter nil "Perimeter"))
    (instance-methods
        (PROPERTIES ()"
            out: list of radius and perimeter.
            Return the properties of the circle."
            (list Radius Perimeter))
        (INIT (R)"
            in:  R.
            out: Perimeter.
            Define radius and derive perimeter."
            (setq Radius R)
```

```
            (setq Perimeter (* 2 pi Radius)))))
=> CIRCLE
```

Note that the class method NEW has been shadowed because the inherited NEW method from the Object class is not expecting that the INIT method requires any arguments. Now we supply the initial radius of a circle by supplying one more argument:

```
(circle 'new 'circle2 10) => CIRCLE2
(circle 'properties) => (10 62.83185307179586)
```

3.2.10. Error Handling

The design and implementation of programs frequently leads to unexpected complexity, which in turn is the cause of many programming errors. In interactive programming environments like Common Lisp it is especially tempting to execute partially implemented code. The question whether the trial and error technique is a favorable programming approach is beyond the scope of this paper. However, OPUS provides a simple but extendable error handling scheme.

Missing Methods

It often happens that messages are sent to an object not providing a matching method. In case none of the super classes furnishes a matching method, the NO-MATCHING-METHOD message is sent to the object. Sub classes of the Object class automatically inherit a default NO-MATCHING-METHOD method. The parameter list of this method is given as:

```
NO-MATCHING-METHOD Selector &rest Parameters                    [Object class/instance method]
```

Selector and Parameters are bound to the selector and the arguments of the message causing the error. The default behavior defined in the Object class for instances and classes is to call the Common Lisp error function displaying the receiver object and the message. The NO-MATCHING-METHOD method may be shadowed or extended by any sub class of Object to alter the default behavior.

For example, we define a Group class to group elements. This Group class provides a broadcast method to forward a message to all elements of the group. The elements of the group are instances of a Geometrical-Object class or any sub class of it. Because the sub classes of the Geometrical-Object class introduce additional methods, the broadcast of messages exploiting these methods will cause an error in case they are sent to elements of the group not furnishing these methods. To avoid ending up in the debugger we shadow the NO-MATCHING-METHOD instance method of the Object class with one method defined in the Geometric-Object class:

```
(create-class Group
    "A group consist of arbitrary many elements."
    (sub-class-of Object)
    (instance-variables
        (Elements nil "List of elements."))
    (instance-methods
        (ADD-ELEMENT (Element)"
            in: Element. out: Element-List
            Add a new element."
            (push Element Elements))
        (BROADCAST (Selector &rest Parameters)"
            in: Selector + Parameters = Message.
            The message is propagated to all elements of the group."
            (dolist (Element Elements)
                (apply Element Selector Parameters)))))
```

```
(create-class Geometric-Object
    "Basis class for geometric objects."
    (sub-class-of Object)
    (instance-methods
        (NO-MATCHING-METHOD (Selector &rest Parameters)"
            in: Selector + Parameters = Message
            Shadow the default error handler which signals
            a non-continuable error and halts."
            (format t
                "Instance ~S does not support the ~S selector~%"
                Self
                Selector)))))


(create-class Rectangle
    "A geometric rectangle object."
    (sub-class-of Geometric-Object)
    (instance-variables
        (Length 10 "Long side.")
        (Width 5 "Short side."))
    (instance-methods
        (LENGTH () (print Length))
        (WIDTH () (print Width))))


(create-class Line
    "A geometric line object."
    (sub-class-of Geometric-Object)
    (instance-variables
        (Length 100 "Line length"))
    (instance-methods
        (LENGTH () (print Length))))
```

First we create three instances,

```
(group 'new 'group1) => GROUP1
(rectangle 'new 'rectangle1) => RECTANGLE1
(line 'new 'line1) => LINE1
```

and append the two Geometric-Object instances to group1.

```
(group1 'add-element 'rectangle1) => (RECTANGLE1)
(group1 'add-element 'line1) => (LINE RECTANGLE1)
```

Then, we broadcast the length message.

```
(group1 'broadcast 'length)

100
10
=> NIL
```

The initial length of line1 and rectangle1 are printed out. If we try to broadcast the width message then the NO-MATCHING-METHOD error handler defined in the Geometric-Object class prints a message to the screen.

```
(group1 'broadcast 'width)

"Instance LINE1 does not support the WIDTH selector"
5
=> NIL
```

Note that the broadcasting proceeds without terminating in a non-continuable error situation.

Specific Methods without Implementation

OPUS perceives specific methods consisting exclusively of the method definition differently. Despite not having an implementation, the method shadows any methods of super classes sharing the same name.

However, sending a message to an object matching an unimplemented method causes a continuable error. In interactive environments like Common Lisp it is often the case that a system is only partially loaded. This might lead to very bizarre behavior if no distinction would be provided between implemented and unimplemented methods.

3.2.11. Debugging

No programming paradigm is able to prevent programming errors and misconceptions. In order to cope with problems they first have to be comprehended. Debugging tools make the flow of control more explicit. In object-oriented environments they exhibit messages sent and the objects involved.

Tracing Objects

Because OPUS treats objects as functions in terms of message syntax, objects can be traced by employing the Common Lisp built-in trace macro :

`trace {object-name}*` [Macro]

On the other hand, untrace is used to stop tracing again:

`untrace {object-name}*` [Macro]

If untrace is invoked with no further arguments then all currently traced objects will be untraced. An example is given in the section describing pseudo objects.

Tracing all Messages

Using the built-in trace macro of Common Lisp has its limitations. It can get very complex to predict the set of all involved objects. Furthermore, objects may be created dynamically such that the programmer is not aware of their existence. Also, the number of objects might be too large to be dealt with by the programmer in an explicit way.

The Object class of OPUS provides a trace-all method which in turn traces every message sent in the system:

`trace-all enable` [Object Class Method]

Setting the enable parameter to a non-nil value enables trace-all, whereas a nil value disables the trace-all feature. In contrast to the Common Lisp trace macro, the trace-all method distinguishes between instance and class messages which is especially helpful in the case of exploiting pseudo objects. Reconsider the Rectangle example from the "Extensions with Sub Classes" section:

```
(create-class Rectangle
    "Geometric object."
    (sub-class-of Object)
    (instance-variables
        (Height 0 "Rectangle height.")
        (Width 0 "Rectangle width.")
        (Area   0 "Area of rectangle."))
    (instance-methods
        (DIMENSIONS (Dimensions)"
            in:  Dimensions = (<Height> <Width>)
            out: Area
```

```
            Modify the dimension of the rectangle and
            return its new area."
            (setq Height (first Dimensions) Width (second Dimensions))
            (setq Area (* Height Width))))
    (class-methods
        (NEW (Instance-Name Dimensions)"
            in:  Instance-Name, Dimensions = (<Height> <Width>)
            out: Instance-Name
            Create a new instance and set-up the dimension of it."
            (progl
                (super 'new Instance-Name)
                (funcall Instance-Name 'dimensions Dimensions)))))
```

If we now enable trace-all,

```
(object 'trace-all t) => nil
```

and create an initialized instance of the Rectangle class,

```
(rectangle 'new 'rectangle1 '(5 6))
class message: (RECTANGLE NEW RECTANGLE1 (5 6))
 class message: (SUPER NEW RECTANGLE1)
  instance message: (RECTANGLE1 INIT)
  (RECTANGLE1 INIT) returns:
  RECTANGLE1
 (SUPER NEW RECTANGLE1) returns:
 RECTANGLE1
 instance message: (RECTANGLE1 DIMENSIONS (5 6))
 (RECTANGLE1 DIMENSIONS (5 6)) returns:
 30
(RECTANGLE NEW RECTANGLE1 (5 6)) returns:
RECTANGLE1
=> RECTANGLE1
```

then one gets a good idea of the interaction taking place between instances and classes.

Trace specific Messages, Classes, or Messages

The usefulness of a trace operation is often determined by the amount of output created. In many cases trace-all is too generic. Trace-thing lets the user define specific trace conditions:

`trace-thing &optional` *`Instance Class-Name Selector`* [Function]

where Instance can be an anonymous instance or nil, Class-Name can be a symbol refering to a class or nil, and selector can be a symbol representing a selector or nil. A nil value always denotes a wild card. Therefore, trace-thing with no arguments is identical to the trace-all message. Examples:

```
(trace-thing) ; trace every message to all instances of all classes
(trace-thing nil 'taxi) ; trace every message to all instances of taxis
(trace-thing nil nil '+) ; trace only + messages to all instances of all classes
```

Untrace-thing accepts the same arguments which can be used selectively to disable trace conditions:

`untrace-thing &optional` *`Instance Class-Name Selector`* [Function]

4. The Built-in Class OBJECT

Instead of furnishing a large set of functions, OPUS embodies large parts of its functionality in the Object class. This class is usually the root of all other classes in the system. It provides a default behavior for creating/deleting and inspecting objects.

In the following sections we provide a survey of all instance methods and class methods furnished by the Object class. Types of parameters are denoted by curly braces, e.g., a-number {fixnum} represents a parameter of type fixnum.

4.1. Class Methods

CLASS-OBJECT-P [*Object Class Method*]

Return a non-nil value if the receiver is a class object, nil otherwise. Used to distinguish class and instance objects.

out: Is-Class-Object-p {boolean}.

DESCRIBE (&key [*Object Class Method*]

Stream

(*Variable* *Describe-Class-Variable*)

(*Method* *Describe-Class-Method*)

(*Instance* *Describe-Class-Instance*)

(*Class* *Describe-Class-Class*)

(*Own* *Describe-Class-Own*)

(*Inherited* *Describe-Class-Inherited*)

(*Generic* *Describe-Class-Generic*)

(*Specific* *Describe-Class-Specific*)

(*Unique* *Describe-Class-Unique*)

(*Super-Class-Of* *Describe-Class-Super-Class-Of*)

(*Sub-Class-Of* *Describe-Class-Sub-Class-Of*))

Generate a description of an object. The degree of detail provided in the description is controlled by a set of keywords, which in turn have defaults being defined with global variables.

in: &key *Stream* {stream}

The output of the description is sent to <Stream>.

Variable {boolean or symbol} default: *Describe-Class-Variable*

If <Variable> has a non-nil value then the variables of the receiver object matching <Variable> are described. The t value for <Variable> matches every variable (it can be viewed as a wild card). In case the value of <Variable> is a symbol then only the variable is described having an equal name.

The description of class variables includes the name of the variable, its current value, and its documentation string. Instance variables, on the other hand, are described by their name, their initform and their documentation strings.

Method {boolean or symbol} default: *Describe-Class-Method*

Equal to the Variable keyword, the value of the Method keyword is matched against the existing methods if it has a non-nil value. A t value results in the description of all variables whereas a symbol describes only the method which is equal in name.

Instance {boolean} default: *Describe-Class-Instance*

The instances of the class are described in case of a non-nil value for the Instance keyword.

Class {boolean} default: *Describe-Class-Class*

The class is described in case of a non-nil value for the Class keyword.

Own {boolean} default: *Describe-Class-Own*

Own variables and methods are limited to the properties defined in the described class which are **not** inherited.

Inherited {boolean} *Describe-Class-Inherited*

Describe the inherited properties. These properties do not include shadowed properties (not even methods extended by making use of super), nor own properties.

Generic {boolean} default: *Describe-Class-Generic*

If non-nil, describe the generic methods.

Specific {boolean} default: *Describe-Class-Specific*

If non-nil, describe the specific methods.

Unique {boolean} default: *Describe-Class-Unique*

If non-nil, describe the unique methods.

Super-Class-Of {boolean} default: *Describe-Class-Super-Class-Of*

If non-nil, return a list of immediate sub classes.

Sub-Class-Of {boolean} default: *Describe-Class-Sub-Class-Of*

If non-nil, return a list of immediate super classes.

Example: Sending the following describe message

```
(rectangle 'describe :method t :variable t :instance t :class t :inherited nil)
```

to the rectangle object defined in the "An Example: Grouping of Graphic Objects" section results in:

```
Class Object: RECTANGLE
Geometric object.
Super Class of:
Sub Class of: OBJECT
Class Methods:
 Own:
  From Class: RECTANGLE
```

```
    Generic Methods:
      NEW (Instance-Name Dimensions)
         in:  Instance-Name, Dimensions = (<Height> <Width>)
         out: Instance-Name
         Create a new instance and set-up the dimension of it.
  Instance Methods:
   Own:
    From Class: RECTANGLE
     Generic Methods:
      DIMENSIONS (Dimensions)
         in:  Dimensions = (<Height> <Width>)
         out: Area
         Modify the dimension of the rectangle and
         return its new area.
  Instance Variables:
   Own:
    From Class: RECTANGLE
      AREA initform: 0 Area of rectangle.
      HEIGHT initform: 0 Rectangle height.
      WIDTH initform: 0 Rectangle width.
```

DELETE &Key (*Delete-Instances* T) (*Delete-Subclasses* T) [*Object Class Method*]

Delete the class object.

in: &key *Delete-Instances* {boolean} default: t

If Delete-Instances is set to a non-nil value, then all instances of the class are deleted too.

Delete-Subclasses {boolean} default: t

A non-nil value for Delete-Subclasses also deletes the sub classes of the class.

DOC &rest Arguments &key Stream &allow-other-keys [*Object Class Method*]

Describe the class and its sub classes. Furthermore, print out a graph of classes. The DOC method accepts exactly the same arguments as the Object class method describe.

in: &rest *Arguments* &key *Stream* &allow-other-keys

All arguments are forwarded to the describe method.

INIT [Object Class Method]

Is invoked at create-class evaluation and reevaluation time respectively. The typical use is to initialize class variables. Returns the name of the receiver class.

out: *Self* {symbol}.

NEW (&optional Instance-Name) [*Object Class Method*]

Create an instance of the class.

in: &optional *Instance-Name* {symbol}

If Instance-Name is not supplied then a unique instance name is created.

out: *Instance-Name* {symbol}.

NO-MATCHING-METHOD (Selector &rest Arguments) [*Object Class Method*]

Error handling method invoked in case of not having an own or an inherited method matching the selector of the message sent to class.

in: *Selector* {symbol}

Selector is bound to the selector of the original message sent to the class.

&rest *Arguments* {list of: {s-expression}}

Arguments is bound to the list of arguments of the original message sent to the class.

SUB-CLASS-OF [Object Class Method]

Return the list of names of all immediate super classes.

out: Sub-Class-Names {list of: {symbol}}.

SUPER-CLASS-OF [Object Class Method]

Return the list of names of all immediate sub classes.

out: Super-Class-Names {list of: {symbol}}.

TRACE-ALL *Enable* [*Object Class Method*]

Enable and disable the tracing mode of all messages sent.

in: *Enable* {boolean}

A non-nil value enables the trace, whereas, a nil value disables it.

OPUS::VALUE *Variable-Name* &optional (*Value* Nil Value-Boundp) [*Object Class Method*]

If *Value* is provided then the value of the variable denoted by *Variable-Name* is set to it and the value is returned. Otherwise, the current value of the Variable is returned. This method is intended as a debugging aid only in an interactive fashion. It should never be used to modify the value of an object variable from within a program on a regular basis because this violates the principles of data abstraction.

in: *Variable-Name* {symbol}

The name of an object variable.

&optional *Value* {t}

A new value for the object variable.

out: *Updated-Value* {t}.

4.2. Instance Methods

CLASS-OBJECT-P [*Object Instance Method*]

Return a non-nil value if the receiver is a class object, nil otherwise. Used to distinguish class and instance objects.

out: *Is-Class-Object-p* {boolean}.

INSTANCE-OF [*Object Instance Method*]

Return the name of the class of which the receiver is an instance.

`out: Class-Name {symbol}.`

`COPY &optional Instance-Name` [Object Instance Method]

Copy an instance.

`in: &optional Instance-Name {symbol}.`

If Instance-Name is not provided then a unique symbol is created for the name of the instance.

`out: Instance-Name {symbol}.`

`DELETE` [Object Instance Method]

Deletes the receiver object.

```
DESCRIBE (&key                                                    [Object Instance Method]
    Stream
    (Variable *Describe-Instance-Variable*)
    (Method *Describe-Instance-Method*)
    (Instance *Describe-Instance-Instance*)
    (Class *Describe-Instance-Class*)
    (Own *Describe-Instance-Own*)
    (Inherited *Describe-Instance-Inherited*)
    (Generic *Describe-Instance-Generic*)
    (Specific *Describe-Instance-Specific*)
    (Unique *Describe-Instance-Unique*)
    (Instance-Of *Describe-Instance-Instance-Of*))
```

Generate a description of an object. The degree of detail provided in the description is controlled with a set of keywords, which in turn have defaults being defined with global variables.

`in: &key Stream {stream}`

The output of the description is sent to <Stream>.

`Variable {boolean or symbol} default: *Describe-Instance-Variable*`

If <Variable> has a non-nil value then the variables of the receiver object matching <Variable> are described. The t value for <Variable> matches every variable (it can be viewed as a wild card). In case that the value of <Variable> is a symbol, then only the variable having an equal name is described.

The description of class variables includes the name of the variable, its current value, and its documentation string. Instance variables, on the other side are described by their name, their initform and their documentation strings.

`Method {boolean or symbol} default: *Describe-Instance-Method*`

Equal to the Variable keyword the value of the Method keyword is matched against the existing methods if it has a non-nil value. A t value results in the description of all variables, whereas a symbol describes only the method being equal in name.

Instance {boolean} default: *Describe-Instance-Instance*

The instances of the class are described in case of a non-nil value for the Instance keyword.

Class {boolean} default: *Describe-Instance-Class*

The class is described in case of a non-nil value for the Class keyword.

Own {boolean} default: *Describe-Instance-Own*

Own variables and methods are limited to the properties defined in the described class which are **not** inherited.

Inherited {boolean} *Describe-Instance-Inherited*

Describe the inherited properties. These properties do not include shadowed properties (no even methods extended by making use of super), nor own properties.

Generic {boolean} default: *Describe-Instance-Generic*

If non-nil, describe the generic methods.

Specific {boolean} default: *Describe-Instance-Specific*

If non-nil describe the specific methods.

Unique {boolean} default: *Describe-Instance-Unique*

If non-nil, describe the unique methods.

Super-Class-Of {boolean} default: *Describe-Instance-Super-Class-Of*

If non-nil, return a list of immediate sub classes.

Sub-Class-Of {boolean} default: *Describe-Instance-Sub-Class-Of*

If non-nil, return a list of immediate super classes.

INIT [Object Instance Method]

Invoked at instance creation time. A typical use is to initialize instance variables. Return the instance name.

out: *Self* {symbol}.

NO-MATCHING-METHOD Selector &rest Arguments [Object Instance Method]

Error handling method invoked in case of not having an own or an inherited method matching the selector of the message sent to the receiving instance object.

in: *Selector* {symbol}

Selector is bound to the selector of the original message sent to the class.

&rest *Arguments* {list of: {s-expression}}

Arguments is bound to the list of arguments of the original message sent to the class.

`OPUS::VALUE` *`Variable-Name`* `&optional (`*`Value`* `Nil Value-Boundp)` [*Object Instance Method*]

If *Value* is provided then the value of the variable denoted by *Variable-Name* is set to it and the value is returned. Otherwise, the current value of the Variable is returned. This method is intended as a debugging aid only in an interactive fashion. It should never be used to modify the value of an object variable from a program on a regular basis because this violates the principles of data abstraction.

`in:` *`Variable-Name`* `{symbol}`

The name of an object variable.

`&optional` *`Value`* `{t}`

A new value for the object variable.

`out:` *`Updated-Value`* `{t}.`

6. Index

7. References

[1] M. Stephik and D. G. Bobrow, "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, , pp. 40-61, 1984.

[2] M. Vitins, R. Huebscher and A. Repenning, "A Knowledge-based Approach for the Configuration of Technical Systems," *Eight International Workshop*, Avignon, 1988, pp. .

[3] N. Wirth, *Algorithms and Data Structures*, B. G. Teubner, Stuttgart, 1983.

[4] F. P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, , pp. 10-19, 1987.

[5] J. Martin and S. Oxman, *Building Expert Systems: A Tutorial*, Prentice Hall, 1988.

[6] P. Cointe, "Metaclasses are First Class: the ObjVlisp Model," *OOPSLA '87*, 1987, pp. 156-167.

[7] G. A. Ford and R. S. Wiener, *Modula-2, A Software Development Approach*, John Wiley & Sons, New York, 1985.

[8] G. Steele L., *Common LISP: The Language*, Digital Press, 1984.

[9] U. Schiel, "Abstraction in Semantic Networks: Axiom Schema for Generalisation, Aggregation and Grouping," *SIGART Newsletter*, , pp. 25-26, 1989.