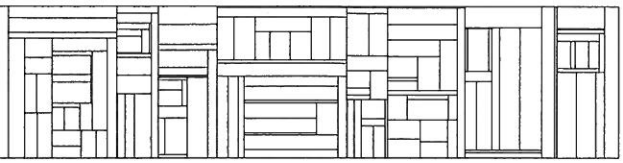
ISSN 0105-8517

Block Structure and Object-Oriented Languages

Ole Lehrmann Madsen

DAIMI PB – 230
November 1987

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 71 88 *Telex: 64767 aausci dk*



BLOCK STRUCTURE AND OBJECT-ORIENTED LANGUAGES*

Ole Lehrmann Madsen
Computer Science Department, Aarhus University, Aarhus, Denmark

November 16, 1987

Abstract. Programming languages that support the object-oriented perspective on programming may be divided into two groups. One group of languages originating from Simula follows the Algol tradition with respect to block structure, static name binding and compile-time type checking. Another group of languages originating from Smalltalk is more in the style of the Lisp tradition with a flat set of definitions (classes), dynamic name binding and run-time type checking. The purpose of this paper is to analyze the role of block structure in object-oriented languages. It will be demonstarted that block structure is useful from both a conceptual and technical viewpoint.

*To appear in: *Research Directions in Object-Oriented Programming.* Edited by B.D. Shriver and P. Wegner, MIT Press, 1987.

1 Introduction

Simula 67 [Simula] and Smalltalk [Goldberg & Robson 85] are two examples of programming languages that support the object-oriented programming style. The two languages are different in a number of ways: Simula contains Algol 60 as a subset and supports block structure, static (lexical) name binding, and compile-time type checking. Smalltalk has none of these features. Smalltalk is more in the style of Lisp with a flat set of definitions (classes), dynamic name binding and run-time type checking.

The purpose of this paper is to discuss the role of block structure in object-oriented languages. In [Wulf & Shaw 73], [Hanson 81], [Tennent 82], and [Clarke et al. 80] the role of block structure in Algol-60, Pascal and Ada is discussed. In these languages block structure appears in the form of nested procedures and blocks. In languages like Simula 67 ([Simula]) and Beta ([Beta]) block structure is also present in the form of nested classes.

A class describes the structure (intension) of a category of objects. In Smalltalk an object is characterized by the set of methods (procedures) that it will respond to. In Simula and Beta an object is characterized by a set of *attributes*, which may be variables, procedures and *classes*. It will be demonstrated that the use of classes as attributes of an object is useful from both a conceptual and technical viewpoint. Block structure provides *locality*: By declaring a procedure local to a class or procedure reflects the fact that the procedure only has meaning as part of its enclosing procedure or class. In the same way by declaring a class local to another class states that the local class is only meaningful in a limited context, i.e. as part of an object.

In section 2 block structure as used in this paper is described. The rest of the paper contains a number of examples of using block structure in the form of nested classes. The examples will be given in Beta. In Simula the use of nested classes is limited by a number of restrictions. Beta does not have these restrictions.

2 Block Structure

First the terminology used in the paper is described. This includes the programming language notation used for describing the examples. Next the role of block structure is discussed and finally we comment on some of the discussion of block structure in the literature.

Terminologi

A programming language supports *block structure* if procedures, classes, and blocks can be textually nested as in Simula and Beta. An Algol-60 *block* has a description of the form

begin $D1; D2; ...Dn; I1; I2; ...Im$ **end**

where $D1, D2, ...Dn$ are declarations and $I1, I2, ..., Im$ are imperatives to be executed.

In Algol a block can be used as an imperative in the form of an inner block or for defining a procedure. In Simula a block may have a superclass and it may also be used for defining a class.

In Beta the term *object* is used as a common name for instances of procedures, classes and blocks. In the language defined below, block is covered by the syntactic category object-description. Block structure is supported since object-descriptions may be arbitrarily nested.

The language used for describing the examples is restricted to a minimum. Except for syntax it is a subset of Beta. The syntax is given in figure 1.

Reference-declarations may be either static or dynamic:

- A static-reference-declaration specifies a static reference corresponding to each name in the name-list. A static reference will constantly denote an object described by object-specification; these objects are generated together with the generation of the object containing the declaration.

- A dynamic-reference-declaration specifies a dynamic reference corresponding to each name in the name-list. A dynamic reference may denote any object qualified by the class-name; the objects are generated by executing a new-imperative.

There are two forms of assignment-imperative:

```
<program> ::= <object-description>
<class-declaration> ::= <name> : class <object-description>
<procedure-declaration> ::= <name> : proc <formal-parameters>
                                    <object-description>
<formal-parameters> ::= (<input-parameters>)to(<output-parameters>)
<static-reference-declaration> ::= <name-list> : <object-specification>
<dynamic-reference-declaration> ::= <name-list> : ↑ <class-name>
<object-specification> ::= <class-name> | <object-description>
<object-description> ::= <super-class> begin <declaration-list>
                                    do <imperative-list>
                                    end
<super-class> ::= <class-name> | empty
<imperative> ::= <procedure-activation>
               |  <object-description>
               |  <object-name> := <value-expression>
               |  <object-name> :- <object-expression>
```

Figure 1: Syntax of example language

- $V := exp$ is a usual assignment-imperative where the value of exp is assigned to V.
- V:- W describes a reference assignment. V will now denote the object W. V must be a dynamic reference.

Dynamic references correspond to reference variables in Simula and instance variables in Smalltalk.

Both static and dynamic references have a "type" in the form of an object-specification or class-name. For a dynamic reference, this means that it may only denote instances of that class or one of its subclasses.

In figure 2 is given an example of a program written in the programming language notation used in the paper. Comments are enclosed by { and }.

The example language includes so-called *singular objects*, which are objects described directly without referring to a class or procedure. The whole program is a singular object described by B1. The static reference V denotes a singular object described by B2. B3 is a singular object describing an imperative in the form of an Algol-60 like block.

Most of the examples in this paper may (except for syntax) be expressed in the Beta programming language. Constructs not available in Beta are explicitly mentioned. The examples are in general not express-

```
begin {B1}
      C: class S {S is the superclass of C}
         begin {C-objects have 4 attributes}
               A1: D; {A1 is a static reference denoting}
                      {an instance of class D}
               A2: ↑ D; {A2 is a dynamic reference that}
                        {may denote any instance of D}
               P: proc (X: integer,Y: boolean)to (Z: integer);
                  begin {P is a procedure attribute}
                        {with two input parameters, X,Y}
                        {and one output parameter, Z.}
                        {X,Y,Z are themselves instance of classes}
                  end ;
               T: class ... {T is a class-attribute}
         do I {I is an imperative that may be executed}
         end ;
      E: C; {E denotes an instance of class C, a C-object}
      V: {V denotes a singular object described by B2}
         begin {B2} I: integer end ;
do V.I := 7;
   begin {B3}
         X: integer
   do X := V.I;...
   end ;
end
```

Figure 2: program example

ible in Simula.

The role of Block Structure

In the object-oriented programming style, a program execution is viewed as a *physical model* that simulates the behaviour of some real or imaginary part of the world. Using terminologi from Delta the considered part of the world is called the *referent-system* ([Delta]). In order to create a model of the referent-system concepts covering the relevant phenomena must be developed.

For a concept we shall use the classic terms which are: the *name* used to denote the concept, the *intension*: the properties of the phenomena covered by the concept, and the *extension*: the set of phenomena covered by the concept.

The *model system* (or program execution) contains elements corresponding to the phenomena and concepts selected as important for the desired perspective on the referent-system. Classes and procedures model concepts and objects model phenomena.

Abstraction mechanisms in programming languages are important. Most object-oriented programming languages support the three fundamental subfunctions of abstraction: classification, aggregation and generalization. The inverse functions exemplification, decomposition and specialization are similarly supported.

A class definition is a description of the intension of the instances (extension) of the class. This description includes: one or more superclasses specifying which classes/concepts that the new class specializes, a set of attributes characterizing instances of the new class, and an imperative-list that describes an action-sequence associated with instances of the class.

The attributes of a class/procedure may be described by referring to other classes/procedures, i.e. aggregation is taking place. The attributes may describe components that are a fixed part of the surrounding object, or components which are references to objects. Here block-structure or *locality* is important: *Locality makes it possible to describe that an object is characterized by a concept in the form of a local class or procedure.* This restricts the existence of instances of such local classes or procedures to the lifetime of the enclosing object in which they are defined. In the remaining sections of this paper a number of examples of this will be

given.

Block structure is not a mechanism for "programming in the large" in the sense that a program should be structured as a large program consisting of nested procedures and classes. A programming language must contain facilities for modularizing a program into minor parts. Especially aggregation should be supported by a construct like the Ada package allowing another hierarchy than block structure. In [BETA 83a] a language independent mechanism for program modularization is described.

A concept/abstraction is timeless in the sense that it has no state that changes over time. Since classes are used to model concepts, classes should not have state. An object is a phenomenon which has a state that may change over time. Objects may have the same *form*, i.e. belong to the same class; but they have different *substance*. Substance is physical material characterized by having a volume and a unique location in time and space. Examples af objects are people, furniture, etc.

There are however phenomena which do not have substance ([Delta],[Beta]). A transformation (a partially ordered set of events) is an example of a phenomenon appearing in a program execution. The concepts covering such phenomena are typically modelled by procedures or concurrent process-types (like tasks in Ada). Values, types and functions in programming languages model concepts where the phenomena are measurable properties of the objects (the substance).

Discussion of Block Structure

There are many aspects of block structure being discussed in the literature. Here we shall comment on this discussion.

- *Locality.* The major advantage of block structure is locality. This makes it possible to restrict the existence of an object and its description to the environment (object) where it has meaning.
- *Scope rules.* There are (at least) the following aspects of scope rules for names declared within an object:
    1. They only exist when the object exist. This is a consequence of locality.
    2. Access to global names and redeclaration of names.

Global names may or may not be seen within a block. In [Wulf & Shaw 73] it is being argued that the use of global variables within nested blocks is a source of errors.

It is considered a problem that a name can be redeclared within an internal block. There is however no reason to allow such redeclaration in case it is found to be a problem.

Also it has been criticized that it may be difficult to see which global names are being used within an internal block. Again this this is not inherently tied to block structure and can be avoided. In languages like Euclid ([Lampson 77]), a block must explicitly import from the enclosing block all names being used.

As mentioned above block structure is not a mechanism intended for "programming in the large". In languages like Algol 60 and Pascal this is the case. It is merely inteded for "programming in the small". In this case the above problems tend to be minor.

3. Access to names within a block from "outside" the block may be restricted. Hidden/protected of Simula is an example of this.

- *Syntax.* In [Hanson 81] it is said that:

  > Block structure can make even moderately large programs difficult to read. The difficulty is due to the physical separation of procedure-headings from their bodies....

  In [Tennent 82] it is demonstrated that this is merely a matter of syntax. By using the syntax from Landin's ISWIM it is possible to place internal procedure declarations after the body of the block.

3 Class Grammar

Here we shall give an example of a class with a local class attribute. The actual example is inspired by Liskov and Zilles ([Liskov & Zilles 74]), but is typical for a number of situations. The example defines a grammar that is going to be used for constructing a precedence parser. For this purpose a class corresponding to the symbols (*tokens*) of the grammar

```
grammar: class
        begin ... {grammar representation}
            token: class
                begin ... {token representation}
                    isTerminal: proc ()to (b: boolean);...
                    isNonTerminal: proc ()to (b: boolean);...
                end ;
            precRel: proc (S1, S2: token)to (C: char);
                begin ...end ;
        end
```

Figure 3: Class Grammar

must be present and a precedence relation function must be defined. The grammar can be structured as shown in figure 3.

It is possible to declare instances of class *grammar* in the following way:

$G1$: *grammar*; A, B: $G1$.*token*;

G1 is an instance of class *grammmar* and *A,B* are instances of *G1.token*. For the tokens *A,B* operations like *A.isTerminal* and *G1.precRel*(A, B) are possible.

Consider another set of instances:

$G2$: *grammar*; X, Y: *G2.token*

G1 and *G2* are both instances of class *Grammar*. *A,B* and *X,Y* are not instances of the same class. *A,B* are instances of *G1.token* and *X,Y* are instances of *G2.token*. Intuitively this is what we want, since *A,B* are *G1*-tokens and *X,Y* are *G2*-tokens. The two class of tokens are clearly different.

Also the two functions *G1.precRel* and *G2.precRel* are different functions.

By declaring *token* local to class *grammar* we have the possibility to distinguish between tokens from different grammars. Also since class *token* is local to class *grammar*, a token has no existence without a grammar. This also seems to be intuitively correct.

In Liskov and Zilles, class *token* is declared outside class *grammar*. The representation of a *token* is therefore not restricted to the definition of class *grammar*. Liskov and Ziles are aware of this problem:

> Token is a good example of a type created to control access to implementation details. .... Therefore, the new type, *token*,

is introduced to limit the distribution of information about how the grammar is represented.

In our opinion class *token* should be defined as part of class *grammar* since token-objects are only meaningful in relation to a specific grammar. Also knowledge about the representation of a token should be restricted to class grammar.

In the example above the static reference A denotes a token from the grammar $G1$.

A dynamic reference like

G1token: ↑ *G1.token*

may denote any *token* from $G1$.

It is also possible to declare a dynamic reference that may denote *token* objects from any *grammar*. Such a dynamic reference may be declared as follows:

gramToken: ↑ *grammar.token*

Note the difference from the declaration of e.g. A using *G1.token* where $G1$ is a specific *grammar* object. In the declaration of *gramToken*, *grammar* is a class name. *GramToken* may denote a *token* object from any *grammar*.

4 Procedural Programming

In this section we shall show how to support procedural programming in the style of Pascal or Ada. (called *operator/operand-style* in [Cox 84]). Simula and BETA support Pascal-like procedural programming since a program may consist of a collection of procedures that manipulate a set of data structures. The data structures being implemented as instances of classes used like Pascal records. It is more interesting to consider procedural programming in the style of Ada using packages. This is the subject for the rest of this section.

4.1 Functional Classes

Simula has often been criticized for the unnatural asymmetry between operands of a function. Consider the class *complex* in figure 4. Here it seems unnatural that one of the arguments of *plus* serves a special purpose.

```
complex: class
        begin ...
            plus: proc (a: complex)to (b: complex); ...
        end ;
C1, C2, C3: complex
...
C3 := C1•plus(C2)
```

Figure 4: Complex Class

This has been criticized by many people. In CLU ([Liskov & Zilles 74]) they have decided to qualify the operations by means of the class-name instead of the instance-name. The above operation would then look as follows in CLU:

$$C3 := complex\$plus(C1, C2)$$

A consequence of this is that that all operation-calls must be denoted in this way. This does not fit into the object-oriented style.

In Smalltalk the asymmetry has been kept. Numbers are viewed as instances of a class and respond to messages. We are not necessarily convinced that this is a natural way of modelling numbers in a programming language. Below we shall show that the more traditional view can in fact be modelled within the Simula/BETA world.

Consider the definition of a complex package in figure 5. The class *complexPck* defines a set of attributes that implement complex numbers. The object C is an instance of *complexPck*. The attributes of C may then be used as shown in the example. In CLU, the operations are qualified by a type name. Here they are qualified by an object name. As it may be seen, the definition of complex numbers is "functional". There is no asymmetri between the arguments of the operations. In the rest of this papers objects like C will be called *package objects*.

4.2 Mutually Dependent Classes

In CLU a class defines one abstract data type. In Ada it is possible to define a package consisting of *mutually dependent types*, i.e. types that must know about each others representation. It is straight forward to generalize the technique used for class *complexPck* to define mutually dependent classes. In figure 6 is shown a class that describes package

```
begin
    complexPck: class
                begin
                    complex: class begin I,R: real end ;
                    create: proc (R,I: real)to (C: complex);
                            begin do C.R := R; C.I := I end ;
                    plus: proc (A,B: complex)to (C: complex;)
                          begin
                          do C.I := A.I + B.I; C.R := A.R + B.R
                          end ;
                end ;

    C: complexPck;
    X,Y,Z: C.complex;
do
    X := C.create(1.1,2.2); Y := C.create(3.1,0.2); Z := C.plus(X,Y)
end
```

Figure 5: Complex Package

objects with attributes consisting of n classes and m operations.

The notation used in these examples have two immediate drawbacks:

- It may be awkward always to have to qualify attributes of a package object with the name of the package object. This may be avoided by a mechanism similar to the with-statement of Pascal or inspect-statement of Simula (the with-statement is not in Beta):

  ```
  with aT do
  begin a: T1; b: T2; c: T3
  do a := F1(b,c)
  end
  ```

  This is of course possible if there is only one instance of the class. A similar problem appear with the use of Ada packages and Ada has similar solutions for this. Also C++ [Stroustrup 86] provides a solution for this problem.

- If only one instance of the class is needed, it may also be desirable to avoid declaring the class. This can be avoided by describing a package object as a singular object:

```
T: class
   begin
         T1: class ...
         T2: class ...
             ...
         Tn: class ...

         F1: proc (x: T2; y: T3)to (z: T1);...
         F2: proc ...
             ...
         Fm: proc ...
   end

aT: T;
a: aT.T1; b: aT.T2; c: aT.T3;
...
a := aT.F1(b, c)
```

Figure 6: Definition of a set of mutually classes

```
aT: begin
        T1: class ...; T2: class ...;...
        F1: proc ...; F2: proc ...;...
    end
```

The examples in this section describe package objects that have class- and procedure attributes. There is thus no state associated with these package objects. Since an <object-description> may contain variable declarations, it is possible to describe package objects with state. A singular package object like aT is then quite similar to an Ada package. A class, like T or *complexPck*, describing package objects will correspond to an Ada generic package. Finally an instance of such a class will correspond to an instantiation of an Ada generic package.

An Ada generic package may be parameterized in various ways. In Beta, a class (called pattern in Beta) may have virtual pattern attributes which can be used like generic formal types etc. For examples of using virtual patterns see [BETA 87a] and [BETA 87b].

Finally it may be noted that a class like T corresponds to an algebraic structure:

$$T = (T1, T2, \ldots, Tn, F1, F2, \ldots, Fm)$$

5 The Prototype Abstraction Relation Problem

Consider a phenomenon that may be viewed as a prototype of a set of other phenomena. Such a prototype phenomenon may be modelled as an instance of a class describing the properties of it and other similar prototype phenomena. The prototype phenomenon has a state which is changing over time, so it seems unnatural to describe it as a subclass of the class describing its properties.

The prototype phenomenon bears a certain relation to the set of phenomena of which it is a prototype. This relation indicates that the prototype phenomenon should be modelled as a class.

The problem is known as the *Prototype Abstraction Relation Problem* and has been formulated by Brian Smith ([Smith 84]). The problem is best described by an example.

- Consider a class *flightType*, which defines the properties of flight descriptions in a flight table.
- *SK273* between Copenhagen and Los Angeles is an example of such a flight. *SK273* is a flight that takes place every Monday, Wednesday, and Friday. The scheduled departure time is 11:30am.
- During a period of time the properties of *SK273* may change. E.g. the scheduled departure time may be changed. This indicates that *SK273* should be modelled as an instance of class *flightType*.
- *SK273* may be viewed as a a prototype of the actual flights that take place between CPH and LA. The actual flights are characterized by attributes such as actual departure time, actual flight time, etc.
- *SK273* may also be viewed as a class with the actual flights as instances. Also *SK273* could be a subclass of class *flightType*.

A solution to this problem may be formulated using block structure in the form of classes as attributes. The solution is presented in figure 7.

- Class *flightType* is a class describing the properties of the flight prototypes in the flight table. The attributes of instances of *flightType* are source and destination of the flight, frequency, scheduled

```
flightType: class
    begin source, destination: city;
          frequency: setOfWeekDay;
          departTime: timeOfDay; {departure time}
          flightTime: timePeriod;
          flight: class
                begin departuredate: date;
                      actualDepartTime: timeOfDay;
                      actualFlightTime: timePeriod;
                      departureDelay: proc ()to (d: timePeriod)
                      begin
                            d := actualDepartTime − departTime
                      end ;
                end ;
    end ;

SK273: flightType
       where source = Copenhagen, destination = LosAngeles,
             frequency = {Mon, Wed, Fri},
             departTime = 12 : 30am,
             flightTime = 11h : 30m;

myFlight: SK273.flight
          where departureDate = Feb.1.84,
                actualDepartTime = 12 : 45am
                actualFlightTime = 11h : 15m;
```

Figure 7:

departure time, scheduled flight time, etc. In addition there is an attribute class *flight* describing the properties of the actual flights of the prototype. Attributes of the actual flights are departure date, actual departure time, actual flight time, departureDelay, etc.

- *SK273* is an instance of class *flightType*.
- Instances of class *SK273.flight* are the actual flights between CPH and LA.
- *myFlight.departureDelay* will give the time that one specific actual flight has been delayed.

The **where**-construct is used to indicate bindings of the variables in an object. In Beta this can be expressed by means of virtual patterns.

6 Smalltalk Metaclasses

In Smalltalk-80 all system components are represented by objects. Since all objects are instances of a class, the classes themselves are represented by instances of so-called *metaclasses*. Metaclasses give rise to both a philosofical and technical discussion.

As stated in section 2, a class is a model of an abstraction and abstractions are timeless in the sense that they have no state changing over time.

A Smalltalk class viewed as an object may have a state changing over time.

It is true that concepts develop over time and may be changed. A concept is thus only "stable" during a period of time. When a concept changes, it is the whole structure (intension) of the concept that changes. This cannot be modelled by Smalltalk metaclasses. Here it is only possible to model changes by means of class variables. Changing the structure of a class corresponds to editing the definition of a class. This kind of change is of course possible in any programming system, but in Smalltalk-80 (and other systems) this has not been reflected in any philosofical model.

Technically metaclasses are useful for describing variables and methods (class variables and class methods) that are common to all instances of a class. Class methods are useful for generation of instances and initialization of instance variables. They do however not provide a solution for initialization of class variables.

Below we shall show that the pure technical use of metaclasses can be simulated by block structure. The description of the Smalltalk class in figure 8 may be simulated by the class in figure 9.

The class *metaT* corresponds to the metaclass for class T. The singular object *aMetaT* is an instance of the metaclass with two additional variables Xv and Yv. The object *aMetaT* corresponds to class T viewed as an object. There is however a difference between the instance *aMetaT* and the class T. T is described as a class attribute of *aMetaT*.

Simulation of class variables is a bit complicated. The attributes Xv and Yv of the singular instance *aMetaT* simulates the actual class variables. The singular objects X and Y contain the operations on X and Y. These operations must access the variables Xv and Yv in order to ensure that all instances of class T access the same class variables.

class name	T
superclass	S
instance variable names	$N\ M$
class variable names	$X\ Y$
class methods	$new\ \ldots$
instance methods	$M1\ \ldots M2\ \ldots$

Figure 8: Sketch of Smalltalk class

```
metaT: class metaS
       begin
            X:  {Simulation of class variable X}
                begin  {declaration of operations on X}
                       op1: proc ...
                            begin  {access aMetaT.Xv} end ;
                       ...
                end ;
            Y: ... {like X}

            new: proc ()to (R: T);...

            T: class S
               begin
                       N, M: ...
                       M1: proc ...
                       M2: proc ...
               end ;
       end ;

aMetaT: metaT  {The superclass of the object}
        begin Xv, Yv: ...;
              {Storage for X and Y}
        end
```

Figure 9: Model of a class and metaclass using block structure

In the example, the variables *Xc,Yc,N,M* have not been given any qualification (type) following the Smalltalk tradition.

Instances of class *aMetaT.T* may be created by *aMetaT.new*.

We shall not here postulate that this is better than metaclasses in Smalltalk. The purpose is just to show that the more technical usage of metaclasses, can be simulated with more traditional language constructs.

There is another technical use of metaclasses in the sense that all system components are represented by instances of classes. The text/description of a class is itself an object. This is useful in a programming system. Here it is however important that the levels be separated. The level of the program execution and the level of program modification are different and should be kept separate.

7 Conclusion

The purpose of this paper has been to show that block structure as found in Simula 67 and Beta, but abandoned in Smalltalk-80, is a natural and powerful mechanism. When modelling phenomena, it is useful to be able to characterize an object by means of a class. In any case block structure is useful for a number of technical problems in programming. This has been demonstrated by giving examples of the use of block structure.

Class *grammar* is perhaps the most common type of example where a class (*token*) is described locally to a grammar. The prototype abstraction relation problem is just a special version of this problem.

The examples in section 4 shows that even the procedural style of programming can be supported within a language which is primarily intended for the object-oriented style of programming. As pointed out by others, ([Cox 84],[Nygaard & Sørgaard 85]), a programming languages should not only support one style. Object-oriented programming, procedural programming and to a limited extent functional programming are supported by languages like Simula, Beta and C++.

Smalltalk-80 metaclasses seem to be a technical trick which may be handled by more traditional constructs.

In [BETA 85] block structure has been used to define objects containing locally defined objects. Such *compound objects* are shown to be an alternative to guarded input/output commands.

In Beta the notions of class, procedure, function and type have been

unified into one abstraction mechanism, the pattern. Instances of a pattern may then be used as objects, procedure activations or variables. A single general concept thus serves a number of purposes. In contrast Ada contains a number of concepts that are similar, but have been treated differently in many situations. This has been critizised by Peter Wegner ([Wegner 83]).

Acknowledgement. This work has evolved during discussions with Jørgen Lindskov Knudsen, Bent Bruun Kristensen, Birger Møller-Pedersen, Kristen Nygaard, Kristine Stougård Thomsen and Terry Winograd. Per Fack Sørensen pointed out an error in the treatment of class variables in a previous version of this paper. Part of this work has been supported by Stanfords *Center for the Study of Language and Information* and by the *Danish Natural Science Research Council* FTU grant no. 5.17.5.1.25.

8 References

1. [Beta 83a] B.B. Kristensen, O.L Madsen, B. Møller-Pedersen, K. Nygaard: Syntax Directed Program Modularization. In: *Interactive Computing Systems* (ed. P. Degano, E. Sandewall), North-Holland, 1983.

2. [Beta 83b] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Abstraction Mechanisms in the BETA Programming Language. *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages, January 24-26 1983, Austin, Texas.*

3. [Beta 85] B.B. Kristensen, O.L. Madsen, B. Møller Pedersen, K. Nygaard: Multi-sequential Execution in the BETA Programming Language. *Sigplan Notices, Vol. 20, No. 4, April 1985.*

4. [BETA 87a] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: The BETA Programming Language — Part 1: Abstraction Mechanisms — Part 2: Multi-Sequential Execution. *In: B.D. Shriver, P.Wegner (ed.),* Research Directions in Object Oriented Programming, *MIT Press, 1987.*

5. [BETA 87b] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Classification of Actions or Inheritance also for Methods. *Proceedings of the Second European Conference on Object Oriented Programming, Paris, June 1987.*

6. Clarke L.A., J.C. Wiledon, A.L. Wolf: Nesting in Ada Programs is for the Birds. *Proc. ACM Symposium on the Ada Programming Language, SIGPLAN Notices,* **11**, *139-145 (1980).*

7. Cox B.R: Message/Object, An Evolutionary Change. *IEEE SOFTWARE, Jan. 1984.*

8. Goldberg A., D. Robson: Smalltalk-80, The Language and its Implementation. *Addison Wesley 1985.*

9. [Delta] E. Holbæk-Hanssen, P. Håndlykken, K. Nygaard: System Description and the Delta Language. *Norwegian Computing Center, Publ. no 523, 1975.*

10. Hanson D.R.: Is Block Structure Necessary. *Software Practice and Experience, Vol. 11 853-866 (1981).*

11. Lampson B.W. et al: Report on the Programming Language Euclid. *SIGPLAN Notices* **12**, *2 (1977).*

12. Liskov B., S. Zilles: Programming with Abstract Data Types. *Sigplan Notices, Vol. 9, No. 4, 50-59 (1974).*

13. Nygaard K., P. Sørgård: The Perspective Concept in Informatics. *In: G. Bjerkness, P. Ehn, M.Kyng (ed.)* Computers and Democracy – A Scandinavian Challenge. Gower Aldershot, England 1987

14. [Simula] O.-J. Dahl, B. Myrhaug, K. Nygaard: SIMULA 67 Common Base Language. *Norwegian Computing Center, Oslo 1968.*

15. Smith B.: Personal Communication. *Stanford 1984.*

16. Stroustrup B.: The C++ Programming Language. *Addison-Wesley, 1986.*

17. Tennent R.D.: Two Examples of Block Structuring. *Software-Practice and Experience, Vol. 12, 385-392 (1982).*

18. Wegner P.: On the Unification of Data and Program Abstraction in Ada. *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages, January 24-26, 1983, Austin, Texas.*

19. Wulf W.A., M. Shaw: Global Variables Considered Harmful. *Sigplan Notices, Vol. 8, 28-34 (1973).*