CLASSES AND OBJECTS

-

A DYNAMIC APPROACH

by

J. Steensgaard-Madsen*

TR78-356

DIKU, University of Copenhagen
Sigurdsgade 41
DK-2200 Copenhagen N
DENMARK

Department of Computer Science
Cornell University
Ithaca, New York 14853

*On leave until July 1979 with support from the University of Copenhagen and the Danish Natural Research Council.

Classes and Objects - a dynamic approach.

J. Steensgaard-Madsen *

DIKU, University of Copenhagen,
Sigurdsgade 41,
DK-2200 Copenhagen N
Denmark

Department of Computer Science
Cornell University
Ithaca, New York 14853

ABSTRACT

Data encapsulation, abstract data types and classes are terms associated with a concept not fully clarified or accepted. This paper presents a class concept that differs slightly from previous definitions by the associated dynamics. This allows us to interpret nested and recursive classes as well as class parameters.

We will distinguish between types and classes and permit types as parameters in a way that allows simple implementation.

A number of examples will be given to illustrate the class concept itself and its application to access control problems for concurrent programs. Synchronization primitives will be viewed as classes and the need for explicit high-level constructs like monitors is questioned.

Keywords: Programming language, encapsulation, abstract data type, class, object, synchronization, monitor.

* On leave until July 1979 with support from the University of Copenhagen and the Danish Natural Research Council.

Classes and Objects - a dynamic approach.

J. Steensgaard-Madsen

DIKU, University of Copenhagen,
Sigurdsgade 41,
DK-2200 Copenhagen N
Denmark

1. Introduction.

In this paper we address the problem of defining a class concept using only Algol 60 like scope rules. Intuitively, a class is a description of a set of objects, each implementing a structure of associated variables, operations and types. We will solve our problem by defining classes in terms of more primitive concepts, in particular procedures.

The scope rules of Algol 60 do suffice for hiding information. An Algol 60 programming technique for this has been described by Krutar (1973). It relies heavily on the use of procedures and functions as parameters. Proper use of procedures and functions as parameters in a language that supports such use would substantially reduce the need for an explicit language construct for classes.

However, classes are useful as a high-level structuring concept. Their relation to procedures is similar to the relation of iterative statements to jumps. The class and associated concepts are proposed as facilities for use in a Pascal-like programming language to compete with languages like Concurrent Pascal, Modula, Euclid, CLU and Alphard. Nested class definitions, recursively defined classes and class parameters will cause no problems.

Although the term "abstract type" is often used synonymously with "class", the latter term is preferred because we will distinguish between classes and types. Unification of the two concepts may be useful for certain kinds of research, but this does not mean they should be unified in a programming language. Keeping the concepts apart certainly increases the number of concepts, but it provides nice control over representation of classes and reflects execution costs in a better way. Types describe variables and representations of values that may be compared for equality and assigned to variables. Classes describe objects and enforce structured access to and operations upon data.

Files will serve to illustrate certain aspects of classes. The logical properties of files may easily be described by a class, if we can specify element types. Thus, we will conceive the analogy of a variable of type file of T in Pascal as an object of class "file", with T provided as parameter to the class. Rather than use the common notation file (T) we will use file of T, but this is just "syntactic sugar".

Type parameters need only describe the representation of values: in terms of implementation, the number of bits required. A type parameter may only be used as an actual parameter or to describe parameter lists and variables (static and dynamic in the sense of Pascal). This is a crucial decision for the intended efficiency of a possible implementation.

An intended use of the class concept may be illustrated if we again take files as an example of a class. Classes may be built on top of other, eventually machine dependent, ones. Thus, portability of programs will rely on a common set of specifications of classes, e.g. files, the implementation of which will be machine dependent. Everything above the operating system interface, which itself can be described by classes, should be describable in a high-level language supporting this class concept. As conceptual building blocks, classes provide powerful description of interfaces between hardware and software. Even the transition from a state of program execution to a state of manipulating the program as data can be conceived of as temporary use of a somewhat protected class giving access to system variables. A few examples with relation to operating systems will be presented, but further research in this area is needed to settle on appropriately defined common classes.

Automatic indivisibility of access to classes is not to be associated with this proposal. However, a class "semaphore" will be assumed and facilities like monitors as described by Hoare (1974) can be built on top of these.

No modification to the scope rules of Algol 60 will be introduced for the purpose of designing a class concept i.e., free access to non-local identifiers in surrounding blocks is permitted. However, other considerations may result in a desire to change this aspect of the scope rules. This topic will not be discussed further in this paper.

The remaining part of the paper is organized as follows. Section 2 presents the terminology and gives an informal introduction to the concepts and the syntax used in the examples. Section 3 describes the syntax for classes, objects and type parameters. Pascal is used as a basis, but the proposal should not be seen as a proposed extension to that particular language. Familiarity with Pascal is thus presumed. The semantics are also described more formally than in section 2, but only for a language somewhat simpler than the one used later in examples. Generalization from the simple to the more involved case should, however, cause no serious problems. Section 4 contains examples with various levels of detail, and section 5 briefly compares this to related work in the area. Finally, section 5 also contains a few comments on implementation.

2. Classes and objects.

A class is a description of objects. Each object implements one or more facilities - variables, arrays of variables, procedures, functions, objects, arrays of objects, types or classes. An attribute is a description of a facility and it provides an access identifier for the facility. A class exhaustively states the attributes of objects i.e., details of implementation are accessible only by means of access identifiers. An object may be denoted by an identifier, or an identifier and an appropriate number of index expressions. A facility of an object is accessed like the fields of a record, using an object denotation followed by a dot and an access identifier.

Drawing heavily on Pascal syntax we will use the following notation for a class definition

```
class
  <class identifier> of <import type identifiers>
    ( <import parameter list> ) :
  <export identifier> of <attribute type identifiers>
    ( <non-type attribute list> );
<block>
```

where block is like a Pascal block, augmented with class definitions and object declarations. Some obvious abbreviations apply when import or attribute parts are irrelevant.

An object declaration introduces a new object (or an array of objects) of a given class and provides a denotation for it, e.g. an identifier. Objects are declared in much the same way as variables. However, in an object declaration the analogy to a type is a class call i.e., a class identifier followed by the actual import types and parameters corresponding to the definition of the class. Individually identified objects or arrays of objects can be declared using one of the schemes

```
<identifier list> : <class call>

<identifier> ( <index bounds> ) : <class call>
```

An object denotation must be used only within its scope, which is a statement of the form

```
object <object declaration>;
<block>
```

and, however innocently this may look, we thus depart essentially from similar proposals. A block is, as in Pascal, a set of definitions and declarations followed by a statement part. We will allow the above statement as the statement part of a block i.e., it is a compound statement in Pascal terminology. This allows object declarations to appear like other declarations. Further we will allow nested object statements i.e.,

```
object <object declaration>;
{object <object declaration>;}
<block>
```

to be abbreviated:

```
object
   <object declaration>; {<object declaration>;}
<block>
```

A declaration

```
<identifier-1>, <identifier-2> : <class call>
```

is an abbreviation for

```
<identifier-1> : <class call>;
<identifier-2> : <class call>
```

This generalizes to longer identifier lists and to arrays of objects i.e.,

```
a ( 1..10 ) : <class call>
```

is short for

```
a (  1 ) : <class call>;
a (  2 ) : <class call>;
...
a ( 10 ) : <class call>
```

Multidimensional arrays of objects are expanded similarly in row-major order. Thus we note that parameters in apparently one class call may be evaluated repeatedly.

The use of a class is described by its class heading, e.g.

```
class queue of element ( bound : integer ) :
   define ( procedure append ( x : element );
            procedure remove ( var x : element ));
```

The attributes appear as parameters following the export identifier, here: "define". In the example above, the access identifiers are "append" and "remove", denoting two procedures with their parameters completely described. The identifier "element" denotes a type parameter for the definition of the class. Class "queue" also takes an integer parameter, "bound".

A declaration of a queue object may be

```
Q : queue of char ( 50 )
```

whereby Q will denote an object implementing two procedures denoted Q.append and Q.remove. Further details of implementation of Q are inaccessible to a user of Q and are said to be hidden.

The semantics of class definitions and object declarations can be given in terms of rewriting rules. These will transform a program in which the constructs appear into another program in which they do not. The latter program will be taken as the meaning of the former.

3. Syntax and semantics.

We will not give the syntax of a complete language. Instead we will depend on our readers intuitive understanding of a language almost identical to Pascal (Wirth, 1975) with extensions (Steensgaard-Madsen, 1978). However, the proposal is not intended for inclusion in Pascal.

We will extend the concepts of statement, parameter list and blocks as shown in figure 1 *, which uses the notation of the Pascal report. The notation <...> denotes explicitly that a number of Pascal syntax-rules still apply for a particular production. Procedures and functions are allowed to take type parameters, but the syntax is obvious from figure 1 and hence is not described.

A facility is denoted by an object denotation followed by a dot and an access identifier defined in the heading of the class to which the object belongs. An attribute type identifier may only be used for variable declarations, specification of parameters, definition of pointer types, or as an actual parameter.

* Numbered figures will be found in the last pages of this printing.

The basic idea behind the translation of class and object constructs into more primitive constructs is based on the technique described in Krutar (1973). Let us consider a particular scheme and rewrite it:

```
        { 1 } class A of B ( <parameter list-1> ) :
                    C of D ( <parameter list-2> );
              <block-1>;

        { 2 } object P : A of X ( <actual parameters> );
              <block-2>
```

is translated into

```
        { 1 } procedure A of B
                ( <parameter list-1>;
                  procedure C of D ( <parameter list-2> ));
              <block-1>;

        { 2 } procedure P of D ( <parameter list-2(B¦X)> );
                  { taking parameter list-2 from the      }
                  { definition of A and replacing B by X }
              <block-2>;
            begin
              A of X ( <actual parameters>, P )
            end
```

Rewriting class headings in this way applies to their appearance in a parameter list also. Note especially that rewriting this way defines the export identifier as a procedure identifier. Consequently, it must be used as such in block-1. With this we have arrived at another essential difference from similar proposals. Further, the correctness and meaning of block-1 will not depend on implicit relations between identifiers in the class heading i.e., relations established by an object declaration.

We can apply the scheme for a complete (nonsentical) example:

```
          class A of B ( q : B ) :
                C of D ( var u : B; var v : D );
            var U : B; V : integer; Z : char;
              { U := V is always illegal }
          begin C of integer ( U, V ) end;

          object P : A of integer ( 17 );
          var r : integer;  { r := P.v is always illegal }
        begin r := P.u end
```

is translated into:

```
    procedure A of B
     ( q : B; procedure C of D( var u : B; var v : D ));
      var U : B; V : integer; Z : char;
    begin C of integer ( U, V ) end;

    procedure P of D ( var u : integer; var v : D );
      var r : integer;
    begin r := P.u { i.e. r := u } end;
  begin  A of integer ( 17, P ) end
```

Note that the variable Z in A is hidden, wheras U and V become available in the body of P through the parameters u and v. The activation of P from within A is the tricky part that we hide by the syntax of classes and objects.

As is the case with many generalizations, the class and object concepts gain a life of their own as soon as they are used in non-trivial ways. Therefore we will restrict the applicability of our planned rewriting rules to the trivial cases to avoid pushing the complexity into the basic language that we are going to augment. However, this will not keep us from giving rather complex examples.

We will only describe the basic language informally. It is simpler than the language used in the examples since type parameters, classes and objects are not allowed. However, we need to provide a means for describing recursively defined parameter lists, or else we must forbid recursively defined classes in the augmented language. Such a means has been defined in Steensgaard-Madsen (1978); however, its inclusion here will only complicate our rewriting rules and consequently we will not allow recursively defined classes in the augmented language.

Our augmented language will also be simpler than the language used in the examples. We will not allow arrays of objects and no more than one identifier may appear in an object identifier list or a formal object identifier list. Of course type parameters are not allowed. With-statements will not be allowed to abbreviate access to object facilities and object statements may only be used as statement parts of blocks. Again, these restrictions serve only to simplify the rewriting rules or the facilities required of the basic language.

A program written in the augmented language may be transformed into a program in the basic language, which will be taken as the meaning of the former. Proceed as follows:

1. Make every identifier unique within the entire program text and remove all occurrences of $-character. Replace all dot-characters used to select a component of an object by a $-character.

2. Replace every object parameter with the (export) attributes of the heading of the class that is referenced in the class call, adding as prefix to every access identifier the formal object identifier followed by the $-character.

3. Replace the ":" or "):" of every class heading with "(procedure " or "; procedure " respectively, and add a ")" following the last attribute. Then replace all occurrences of class with procedure.

4. Repeatedly rewrite object statements, not containing other object statements, from the form

```
object OB : CL ( <actual parameters> ); <block>
```

into

```
  procedure OB ( <parameters> ); <block>;
begin  CL ( <actual parameters>, OB )  end
```

where <parameters> is the list of (export) attributes from the heading of CL, with prefixes "OB$" inserted before all access identifiers.

A more complete set of rewriting rules providing for both type parameters, recursion and free use of object statements can easily be described if we modify our basic language in the following way:

a. allow type parameters to appear more like any other kind of parameters within a parameter list.

b. provide a mechanism by which an entire parameter list may be associated with a name and permit recursive definition of parameter lists.

c. define a concept of a block statement (as in Algol 60).

For complete generality we would need a construct to match array of objects. This would not be too hard, if we were restricted to static bounds. For dynamically computed bounds we would have to rely on an intuitive understanding of a generalization from static to dynamic bounds of some construct of the basic language, and thus we would gain nothing by rewriting.

We conclude that the semantics of classes and individually identified objects can be described by rewriting rules. The generalization from individual identification to identification by indexing is a familiar one, which we will accept without having it described by rewriting rules.

4. Examples.

This section presents a number of examples. If the reader is familiar with other proposals of similar kind, it should not be difficult to follow the examples without detailed knowledge of syntax. See figure 1 and section 3 for details on syntax and semantics.

Only the first example is fully elaborated. Subsequent examples have been chosen merely to point out special features and are given in detail only if required to make a point. It is in the spirit of these concepts to study complete class definitions and class usage separately.

4.1. Topological sort.

Finding a total ordering which contains a given partial ordering is called topological sorting. A well known algorithm to do this can be found in Knuth (1968) and Wirth (1976). Below is a solution based on the use of a structure called bag (here implemented like a stack), made available as a class.

```
class bag of element ( size : integer ) :
   export ( procedure include ( x : element );
            procedure remove  ( var v : element );
            function  empty   : boolean );

   var
{!}  store ( 1 .. size ) : element;
       { store is the name of an array of individual }
       { variables, not a variable of array type      }
     top : integer;

   procedure INCLUDE ( x : element );
   begin
     if top >= size then error {definition omitted};
     top := top + 1; store ( top ) := x
   end { INCLUDE };

   procedure REMOVE ( var v : element );
   begin
     if top <= 0  then error;
     v := store ( top ); top := top - 1
   end { REMOVE };

begin { dynamics of class definition }
     top := 0;
   { binding of access identifiers }
{!}  export ( INCLUDE, REMOVE, top = 0 )
       { the expression "top = 0" is accepted }
       { as a function without parameters     }
end { definition of bag };
```

```
      const
      low = 'A'; high = 'Z'; stacklimit = 10;

      procedure readelement ( var ch : char ); external;
      { further definition omitted }

      object
  {!}    Suc ( low..high ) : bag of char ( stacklimit );
         out               : bag of char
                             ( ord(high) - ord(low) + 1 );
      var
         first, second,
         this              : char;
         prec (low..high)  : integer;

   begin { main program }

      for this := low to high do prec ( this ) := 0;

      { read neighbours and record for every element:    }
      {    1) the number of preceding neighbours          }
      {    2) the identity of all succeeding neighbours }

        while not input.eot do begin
          readelement ( first ); readelement ( second );
          prec ( second ) := prec ( second ) + 1;
          { include second in the bag  }
          { of successors to first     }
  {!}       Suc ( first ).include ( second )
        end;

      { find elements with no preceding neighbours }

        for this := low to high do
          if prec ( this ) = 0 then out.include ( this );

      { generate the elements in sequence of a total    }
      { ordering by writing and removing all elements   }
      { with no preceding neighbours left               }

        while not out.empty do begin
          out.remove ( first ); write ( first );
  {!}     with Suc ( first ) do
            while not empty do begin
              remove ( second );
              prec ( second ) := prec ( second ) - 1;
              if prec( second ) = 0 then
                out.include ( second )
            end
        end
   end { program }
```

4.2. Nested class definitions.

You may object that storage is not used in an economical way in the topological sort example, because all bags require a predetermined number of hidden variables. A more flexible solution can be obtained by nesting the bag definition in a class that provides for common storage administration. Thus, we get an example of a nested class definition and a class facility.

```
class pool of element ( size : integer ) :
  define
    ( class bag :
        def ( procedure include ( e : element );
              procedure remove  ( var v : element );
              function  empty   : boolean ) );
{ block omitted }
```

Here, pool provides bag as an attribute that may be implemented using a common storage administrator hidden in a pool object. The topological sort example would require the following changes in the object declarations:

```
object        charpool : pool of char ( 200 );
                   out : charpool.bag;
  Suc ( low .. high ) : charpool.bag;
```

and of course a replacement of the definition of class bag with a definition of class pool.

4.3. Dynamic binding.

The following rather artificial example serves only to enhance the difference between this approach to classes and others, characterized by their static binding of access identifiers. We restrict the example to its bare minimum and leave out trivial detatils. The Pascal concept of a set type is used unmodified. The example shows the first steps for defining a concept similar to the set concept, but without the restrictions put on base types in most Pascal implementations and accepted here also.

```
class bitmap ( { set of 0 .. } n : integer ) : define
  ( procedure insert ( i : integer );
    function  member ( i : integer ) : boolean );

  const N = 35 { machine dependent };

  procedure simple_case;
    var representation : set of 0 .. N;
    procedure INSERT ( i : integer ); begin ... end;
    function  MEMBER ( i : integer ) : boolean;
      begin ... end;
  begin
    representation := []; define( INSERT, MEMBER )
  end;

  procedure involved_case;
    var
      representation ( 0 .. n div N ) : set of 0..N;
    ...
  begin ... define ( ... ) end;

begin
  if n > N then involved_case else simple_case
end
```

4.4. Useful class specifications.

This subsection lists a number of class specifications that may be useful in defining specifications for use in portable programs.

4.4.1. Sequential files.

```
class file of T :
  define ( procedure rewrite;
           procedure reset;
           procedure write ( e : T );
           procedure read  ( var v : T );
           function  eof : boolean );
```

4.4.2. Direct access files.

```
class direct_access_file of T :
  define( procedure reopen( id( integer ) : char );
          procedure create( id( integer ) : char );
          procedure put   ( n : integer );
          procedure get   ( n : integer );
          length: integer;
          var buf   : T );
```

4.5. Sorting.

Type parameters may also be used in ordinary procedures. A general sorting routine may be defined with a heading like

```
procedure sorting of T
 ( function sequence ( a, b : T ) : boolean;
   procedure input ( procedure release ( e : T ) );
   procedure output( procedure retrieve( var v : T )));
```

The procedure sorting requires three parameters. The first defines the ordering upon which the sort is to be performed (e.g. sequence(a, b) = a < b). The second procedure, input, will be called by sorting with a procedure parameter, release, that allows a sequence of records to be transferred to sorting one by one in successive calls of release. Similarly, the third procedure, output, will be called from sorting when the sorted sequence of records may be obtained one by one in successive calls of retrieve.

Another procedure may be built from this, taking objects of class file as parameters:

```
procedure filesorting of T
  ( function sequence ( a, b : T ) : boolean;
    object   inf, outf              : file of T );

  var count : integer;

  procedure INPUT ( procedure transfer ( e : T ));
    var u : T;
  begin
    inf.reset; count := 0;
    while not inf.eof do begin
       inf.read( u ); transfer( u ); count := count + 1
    end
  end;

  procedure OUTPUT ( procedure transfer( var v : T ));
     var w : T;
  begin
    with outf do begin
      rewrite;
      while count > 0 do begin
        transfer( w ); write( w ); count := count - 1
      end
    end
  end;

begin sorting of T ( sequence, INPUT, OUTPUT ) end
```

4.6. Recursively defined classes.

The example below illustrates recursive definition of a class. It implies that the class definition contains a procedure call (cf. delimit) with a list object as actual parameter. Thus a declaration of such an object must appear in the class definition.

```
class list of item : define
    ( procedure atom( e : item );
      procedure sublist
           ( procedure delimit
                  ( object v : list of item )));
{ block omitted }
```

An object of class list provides facilities to build a list. An atom may be appended to the right end of a current list by means of the atom procedure. A sublist may similarly be appended as a single component by use of the sublist procedure. The number of components in a list (or a sublist) is determined by the dynamic behaviour of a procedure that takes a list object as parameter: whenever the procedure terminates the last component has been included in the list build by use of that object.

As an example consider the following procedure for building a list from an external text-representation, e.g. (ab(c())d):

```
procedure build( object u : list of char );
begin { nextch = '(' }
  input.read ( nextch );
  while nextch <> ')' do begin
    if nextch in letters then u.atom( nextch )
    else if nextch = '(' then u.sublist( build );
    input.read ( nextch )
  end
end
```

4.7. Triangular structure.

Consider the example below a scheme for definition and use of a triangular structure. First we declare an object, rowlength, of class incr. This provides access to a function, rowlength.i, that in successive calls returns the values 1, 2, 3, ... Then we declare an array of objects, row, each of which provides an array of variables, col (1..up). The value of up is determined by computing rowlength.i, and according to the meaning of declaration of array of objects this implies repeated computation of the function. Thus, row (i) provides access to the variables row (i).col (1..i).

```
class incr : define ( function i : integer );
  var k : integer;
  function f : integer;
  begin k := k + 1; f := k end;
begin k := 0; define ( f ) end;

object
  rowlength     : incr;
    {rowlength.i = number of calls of rowlength.i}

class vector of T ( length : integer ) : export
  ( var col ( integer ) : T; up : integer );
  var  x ( 1..length ) : T;
       lgth              : integer;
begin lgth := length; export ( x, lgth ) end;

object
  row( 1..10 ) : vector of integer( rowlength.i );
   { row( i ) : (var col(1..i) : T; up : integer) }
begin
  ...  q := row( p ).up; row( p ).col( q ) := 0; ...
end;
```

4.8. Operating system applications.

One advantage of our approach to encapsulation is that access patterns can be enforced by proper class definition. The literature treats numerous problems on access patterns required to handle shared data. Common problems are known by names like "readers and writers", "dining philosophers", and "bounded buffer". This class concept provides a tool for solving these access pattern problems. We will assume a predefined class, semaphore (cf. Dijkstra, 1968), to provide synchronization primitives and apply the class concept to build more advanced tools like the conditional critical region statement proposed by Hoare (1972) and primitives for simple and correct construction of monitors as proposed by Brinch Hansen (1973).

The examples below should not be taken as final proposals of suitable classes for the purpose of writing programs for concurrent processes, but they are a first attempt in this area.

4.8.1. Kernel interface.

Class headings are well suited to describe interfaces to hardware and basic software. In the following sections we will assume the existence of a kernel that implements semaphores as described by the heading

```
class semaphore ( initial : binary ):
  define ( procedure P; procedure V );
```

Further, we will assume that the kernel provides a facility for dynamic creation of processes. The actions of a process will be described by a procedure, hence no explicit process construct will be introduced. The kernel facility might be described by the class heading:

```
class
  control
    ( procedure activity; store_size : integer ) :
  define
    ( procedure start ( maxtime  : integer );
      procedure kill; status : state );
{ block omitted; it is implementation dependent  }
{ and may or may not be based on simpler classes }
```

The parameter activity represents the action of the process. The process will start running as a consequence of a start-instruction. Activity requires no parameters. However, the body of the actual parameter may contain a parameterized call. For convenience we may even allow a procedure call as the actual parameter and thus obtain the flexibility of parameterized processes. A discussion of whether this is an appropriate way to introduce processes is outside the scope of this paper

4.8.2. Synchronization structures.

We may accept various low-level synchronization primitives as predefined if we can construct desired and easily available structures on top of these. Semaphores will be chosen as primitives and conditions as proposed by Hoare (1974) will be built on top of these. Closely following Hoare's paper we get the definition in figure 2.

The class conditions ensures that wait and signal operations can only be accessed within an object statement of class entry (an entry-statement for short). Entry-statements provided for by one condition object are mutually exclusive. Wait and signal operate on condition-variables, which we identify by integers in the range 1 to "number". Use of the class is of course completely described by the class heading:

```
class conditions ( number : integer ) :
  provide
    ( class entry :
        define ( procedure wait   ( i : integer );
                 procedure signal ( i : integer )));
```

In the implementation a condition variable i is represented by semaphore cond (i) and a count of waiting activities condcount (i).

A key variable is urgentcount, which states the number of activities hung up due to their performing a signal operation. It is used to assure that no activity will be allowed to enter an entry-statement until all signal operations have been completed, i.e. urgentcount = 0 if the semaphore mutex is "open".

Hoare illustrates the monitor concept by an administration of a bounded buffer. Closely following his design, we build a class similar to his monitor using a condition object, s, which contains two condition variables. Using entry-statements as statement parts for the procedures APPEND and REMOVE we know that these will be mutually exclusive like entries in Hoare's monitor.

```
class buffer of portion ( n : integer ):
  define( procedure append ( x : portion );
          procedure remove ( var x : portion ));

  object s : conditions ( 2 );
      { object s prepares for mutual exclusion }
      { and synchronization                    }

  var buffer ( 0..n-1 ) : portion;
      last, count, N    : integer;

  const nonempty = 1; nonfull = 2;

  procedure APPEND ( e : portion );
    object u : s.entry;
        { object u ensures mutual exclusion and }
        { enables access to wait and signal     }
  begin
    if count = N then u.wait( nonfull );
    buffer( last ) := e;
    last := ( last + 1 ) mod N;
    count := count + 1;
    u.signal ( nonempty )
  end;

  procedure REMOVE ( var v : portion );
    object u : s.entry;
  { remainder of definition of REMOVE is omitted }

begin N := n; define ( APPEND, REMOVE ) end;

object channel : buffer ( 100 );
  ...
```

The definition of class buffer, and declaration of the object channel belong in an environment in which a number of activities are created. According to the section on kernel interface one activity, a producer, may have "channel.append" as a parameter but not "channel.remove", which may be reserved for another group of activities, consumers. Consequently, this approach to class definitions seems to offer a solution to the problem of different access rights to one object from different users.

Use of the class condition is straightforward for construction of monitors: every shared procedure is defined by an entry-statement, provided for by an object of class condition.

An even finer degree of mutual exclusion is possible because it is stated explicitly. The problem known as "readers and writers" illustrates a case which may call for a particular primitive, similar to conditions, to handle a particular kind of access pattern. An implementation in terms of semaphores is straightforward.

4.8.3. Conditional critical region.

Remember we consider the construct

```
        object
          <object declaration>; { <object declaration>; }
      <block>
```

to be a statement. This may be associated with a previous object declaration which provides class facilities. If an object only provides a class facility, further use of the object is enforced to be within object-statements. Thus we may obtain a strong connection between a declaration and statements using the declaration.

We will define a class to match declaration of shared variables and the associated conditional critical region statement proposed by Hoare (1971):
region <shared variable> *when* <expression> *do* <statement>.

```
        class sharing of T : define
          ( class when( function B ( x : T ) : boolean ) :
                access( var shared : T ));
```

to be used in a block as follows:

```
    type resource = record p : char; ... end;

    object v : sharing of resource;
     ...
  begin
    ...
    object critical : v.when( ... );
    begin critical.shared.p := 'A'; ...  end
    ...
  end
```

Let us not be concerned with problems of efficiency and define the class shared by the use of conditions. Our previous example shows how conditions can be implemented in terms of semaphores.

Class when requires a function parameter B by which the statement guard can be computed. If the guard is false, the calling activity will wait for a signal on condition 1. Whenever a signal is received in this state, it is immediately propagated to other activities waiting similarly. The guard is then tested again and when it becomes true, access to the shared variable is granted. Finally, as some guard may have become true as an effect of the access a signal on condition is sent.

```
  class sharing of T : define
    ( class when ( function B ( x : T ) : boolean ) :
          access ( var shared : T ));

    object r   : conditions ( 1 );

    var SHARED : T;

    class WHEN ( function B ( x : T ) : boolean ) :
        access ( var shared : T );
      object now : r.entry;
    begin
      while not f ( SHARED ) do begin
        now.wait ( 1 ); now.signal ( 1 )
      end;
      access ( SHARED ); now.signal ( 1 )
    end;

  begin define ( WHEN ) end;
```

5. Discussion.

This section will relate the proposed class concept to other similar ones. Further, a few comments on implementation will be given.

The terminology is taken from SIMULA 67 (Dahl, 1972), the first language to provide a similar concept. However, SIMULA 67 did not associate information hiding with the concept, probably because the need for hiding was not clearly recognized by the computer science community when the language was designed. A facility for hiding has been added lately.

The <u>inner</u> statement of SIMULA 67 matches the use of the export identifier, but the latter is more powerful due to the provision for parameters and its not being restricted in use as a statement. Further, SIMULA 67 defines the concept of <u>virtual</u>, which is replaced by formal class parameters in this proposal.

A SIMULA 67 class is associated with a coroutine rather than a subroutine linkage mechanism, but nevertheless SIMULA 67 is the language providing a concept which comes closest to this proposal. Information hiding is better integrated in our proposal and it seems an advantage not to have an explicit initialization operation like the new-function in SIMULA 67.

Two problems are clearly separated by our approach: information hiding and synchronization of concurrent activities. This opposes the common trend to integrate solutions to these problems and make a fair comparison to more recent languages difficult. Integration has advantages with respect to program verification. However, separate solutions has the advantage that a variety of tools for synchronization and access to shared data may be implemented. Each tool must be described by theorems, rather than axioms, and the proof of the theorems may be very hard. But so will a proof of correct implementation of a specialized language construct, e.g. conditional critical regions.

Other proposals of class-like concepts, e.g. in Concurrent Pascal by Brinch Hansen (1977), rely on a rather complex and ad hoc set of scope rules, justified however, by a desire to prevent certain programming errors. This kind of complexity seems to be expected in a language providing a class concept judged by the DoD language requirements (DoD, 1978). We have seen that we can do with a simple set of rules, which may be compared to the rules governing access to fields of a record.

A unique feature of this approach is the dynamic binding of access identifiers. This is of importance when it is possible to determine from class parameters or the dynamic program behaviour which one of a number of possible implementations of an attribute will be appropriate in the situation, e.g. polar or cartesian implementation of a type complex.

Provision for class and object parameters, nested and even recursive class definitions seems also unique in its generality but Concurrent Pascal and Modula (Wirth, 1977) do have some similar provisions in a simpler form.

In addition to the proposal of the class concept, types has been introduced as parameters. This has been done in a very simple way and it should be mentioned explicitly that parameterized type definitions are not implied by the proposal. Compared with languages like Euclid (Lampson, 1977), CLU (Liskov, 1977), and Alphard (Wulf, 1976) our approach is extremely simple. Of course we have to pay for the simplicity with more cumbersome programs in certain cases, e.g. access to components of structured formal types as allowed in those languages. However, we have not restricted ourselves seriously, because we can always provide attributes for corresponding operations.

Implementation of type parameters is straightforward, because the only operations needed to be parameterized are assignment, test for equality, storage allocation, and parameter passing. The latter operation will be at its simplest if parameter passing is always by reference, which favor a read-only rather than a call-by-value scheme. This has tacitly been assumed in some of the examples.

Our rewriting rules give us guidelines for correct implementation of classes and objects. In relation to this, at least one "optimization" is plausible, namely the representation of facilities in the activation record of the block containing the object statement in which the declaration belongs. Experience gained by Pascal programs structured directly as if transformed by the rewriting rules indicates that efficient implementation is possible and straightforward.

6. Conclusion.

We have in this paper shown how a Pascal-like programming language can be augmented with the following concepts: types as parameters, class definitions, classes as parameters, object declarations and object parameters. Application of these concepts has been illustrated by a number of examples. The semantics for the class and object concepts have been defined by rewriting rules. Efficient implementation is claimed to be straightforward primarily based on experience with programs executionally equivalent to ones using classes and objects.

Our definition of the class and object concepts allows us to assign meaning to nested and recursive class definitions. Examples show this provides a tool to enforce structures that previously has been considered unique in themselves, e.g. conditional critical regions. Use of the class concept should be an aid to programming by stepwise refinement and in the construction of reliable programs.

We have been able to view the synchronization primitive semaphores as a class and have combined this with methods of controlling behaviour to form high-level structures like conditional critical regions and medium-level primitives for construction of monitors.

In general, the class concept provides a tool for describing interfaces between modules, which may range from hardware, through operating systems to objects described by (other) classes. This implies there exists a similar broad application area for the concept.

7. Acknowledgements.

I want to thank E. Reeh Nielsen and A.P. Ravn, both of University of Copenhagen, for their many hours spent in discussions during a preliminary design phase. Further, I am greatly indebted to F. Schneider, J. Donahue, G. Andrews and most of all David Gries, all of Cornell University, who patiently helped me clarify the presentation. However, none of these may be blamed for any inconsistency or error which may be found herein.

8. References.

Brinch Hansen, P. : "Operating system principles", Prentice-Hall, 1973.

- : "The Architecture of Concurrent Programs", Prentice-Hall, 1977.

Dahl, O.-J. : "Hierarchical program structures", in : "Structured Programming", Academic Press, 1972.

Dijkstra, E.W. : "Cooperating Sequential Processes", in : "Programming Languages", F. Genuys (Ed), Academic Press, 1968.

(DoD) Department of Defence Requirements for High Order Computer Programming Languages : "Steelman", 1978.

Hoare, C.A.R. : "Towards a theory of parallel programming", in : "Operating Systems Technique", Hoare and Perott (Eds), Academic Press, 1972.

- : "Monitors: An Operating System Structuring Concept", Communications of the ACM, Vol. 17, No. 10, Oct. 1974.

Knuth, D.E. : "The Art of Computer Programming", Vol. 1, Addison-Wesley Publishing Company, 1968.

Krutar, R.A. : "Restricted Global Variables in Algol 60", SIGPLAN Notices, Vol. 8, No. 12, Dec. 1973.

Lampson, B.W. et. al. : "Report On The Programming Language Euclid", SIGPLAN Notices, Vol. 12, No. 2, Feb. 1977.

Liskov, B.H. et.al. : "Abstraction Mechanism in CLU", Communication of the ACM, Vol. 20, No. 8, Aug. 1977.

Steensgaard-Madsen, J. : "Pascal - clarifications and recommended extensions", DIKU, Sigurdsgade 41, DK-2200, Copenhagen N, Denmark.

Wirth, N. : "PASCAL Report", in : "PASCAL User Manual and Report", K. Jensen and N. Wirth, Springer Verlag, 1975.

- : "Algorithm + Data Structures = Programs", Prentice-Hall, 1976

- : "Modula: a Language for Modular Multiprogramming", Software - Practice and Experience, Vol. 7, No. 1, 1977.

Wulf, W.A. et.al. : "An introduction to the construction and verification of Alphard programs", IEEE Transactions on Software Engineering, SE-4, 4, Dec. 1976.

```
<object statement> ::=
  object <object declaration>; {<object declaration>; }
  <block>

<object declaration> ::=
  <object identifier>{,<object identifier>} : <class call> ¦
  <array of object id> ( <limits>{,<limits>} ): <class call>

<object identifier> ::= <identifier>

<array of object identifier> ::= <identifier>

<class call> ::= <class identification> ¦
  <class identification> ( <actual parameter list> )

<limits> ::= <expression>..<expression>

<class identification> ::= <class identifier> ¦
  <class identifier> of <actual type identifier list>

<actual type identifier list> ::= <identifier list>

<identifier list> ::= <identifier> {,<identifier>}

<class definition> ::= <class heading>; <block>

<class heading> ::=
  class <class identifier> <import attributes> :
    <export identifier> <attributes>

<import attributes> ::= <attributes>

<attributes> ::= <empty> ¦ ( <parameter list> ) ¦
  of <formal type identifier list> ¦
  of <formal type identifier list> ( <parameter list> )

<formal type identifier list> ::= <identifier list>

<parameter list> ::= <formal section> {;<formal section> }

<formal section> ::= <...> ¦ <class heading> ¦
  object <formal object identifier list> <bound types> :
     <class identification>

<bound types> ::= <empty> ¦
  ( <type identifier> {,<type identifier>} )

<formal object identifier list> ::= <identifier list>

<compound statement> ::= <...> ¦ <object statement>
```

Figure 1.

```
class conditions ( number : integer ) :
  provide
    ( class entry :
        define ( procedure wait   ( i : integer );
                 procedure signal ( i : integer )));

  object
    urgent                : semaphore ( 0 );
    mutex                 : semaphore ( 1 );
    cond ( 1..number )    : semaphore ( 0 );

  var
    condcount ( 1..number ) : integer;
    max, urgentcount        : integer;

  { ------ nested class definition ------ }

   class entry :
     define ( procedure W ( i : integer );
              procedure S ( i : integer ));

     procedure wait ( i : integer );
     begin
       condcount( i ) := condcount( i ) + 1;
       if urgentcount > 0 then urgent.V
       else mutex.V;
       cond( i ).P;
       condcount( i ) := condcount( i ) - 1
     end;

     procedure signal ( i : integer );
     begin
       if condcount( i ) > 0 then begin
         urgentcount := urgentcount + 1;
         cond( i ).V; urgent.P;
         urgentcount := urgentcount - 1
       end
     end;

   begin
     mutex.P; { urgentcount = 0 }
     define ( wait, signal );
     if urgentcount > 0 then urgent.V else mutex.V
   end;

  { ------ end of definition of "entry" ------ }

begin
  for max := 1 to number do condcount ( max ) := 0;
  max := number; urgentcount := 0; provide ( entry )
end
```

Figure 2