Visibility and Types

Cornelis H.A. Koster
Technical University Berlin

Abstract:

In this paper it is argued that there is a strong connection between the issue of abstract types and the more general issue of information hiding in large program systems, since abstraction has to be enforced by the careful and controlled hiding of details. In the first part of the paper, the notions of visibility and interface are discussed. In the second part, it is shown how, by careful control of visibility through interfaces, data abstraction can be achieved. Finally a comparison is made between this approach and the class approach.

Keywords: abstract data types, visibility, data abstraction, information hiding, modularity, interfaces.

0. Types and Data-abstraction

By a type, we mean a collection of objects (the instances of that type) characterized in some fashion, together with a name for that collection, the typename. Customarily the typename is used to denote the type.

The instances of a type can be characterized by a common data representation for instances of the type, as in ALGOL 68 or PASCAL, and furthermore by a set of operations, applicable to instances of the type, as in CLU [Liskov/Zilles 1974]. This notion of type can be used as a vehicle for data-abstraction in the following sense:

- (naming) The typename serves as a name for the characterizing data representation, allowing through that name the declaration and further manipulation of objects with that representation.
- (hiding) The type allows to forget details of that representation, in particular by providing operations applicable only to instances of the type, and by forbidding operations based on knowledge of the representation of an instance of the type.

The concept of mode-declaration or type-declaration, as in ALGOL 68 and PASCAL, in conjunction with the strong typing of those languages serves just as well for characterizing a type as the SIMULA class concept.

Yet, most of the present discussion on abstract data types is concerned with the development, starting out from the class-concept of SIMULA 67 [Dahl/Myhrhaug/Nygaard 1968], of special mechanisms for data abstraction, e.g. Clusters [Liskov/Zilles 1974], Forms [Wulf 1974] and other variants of the class-concept, e.g. [Brinch Hansen 1974, Hoare 1975], and ignores the essentially simpler mechanism of the type declaration.

The reason for this preference seems to be in the fact that a class encapsulates the representation of instances of a type together with its characterizing operations in a convenient fashion. Whether this encapsulation is worth the introduction of a special language construct is an interesting question. A more important question is, however, which of the two mechanisms (type-declarations or classes) is best suited for data abstraction. Both mechanisms fail with regards to the hiding of representational details, and special language con-

structs have to be introduced for that purpose.

We intend to show that the hiding of representational details for data abstraction can be performed by the same mechanism that is necessary for algorithmic abstraction. viz. tight control over the scope of named objects. We propose the use of type-declarations plus tight control over the scope of named objects rather than the introduction of ever more baroque variants of the class concept plus separate mechanisms for algorithmic modularity.

1. Visibility

As a basis for the discussion of definitional means for abstract types, we will introduce in this chapter the notions of visibility and interface, as well as some further terminology and notations, which allow a generalization of scope rules, suitable for data abstraction as well as for algorithmic abstraction.

1.1.1. Units

In the sequel we will avoid talking in terms of any specific programming language, and therefore will choose our terms either vague or universal. We visualize a program as broken up into units, which each can have the property that they may

- define some objects and
- apply some objects

1.1.2. Objects

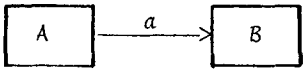
By an object we mean, depending on the programming language used, anything which can be associated with an identifier or with some other symbol through a declaration. A unit could therefore be a declaration or a statement, a larger unit could be a module or a block, and an object could be e.g. a value, a type or a routine.

1.1.3. Visibility of Objects

Each object may be identified in some context by an identifier with which it is associated. The communication of objects between units presupposes the knowledge of their identifiers.

We say that the object, which in the unit A is identified by a, is visible in the unit B, if any occurrence of a in B identifies the same object that it identifies in A.

We can indicate this state of affairs in a picture by



Since the relationship between an object and its identification is so close, we will use the identifier to denote the object when no confusion arises.

1.1.4. Interfaces

By means of an interface, every unit indicates

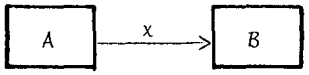
- which objects, visible within it, it is willing to make available to other units; these are termed the objects defined by the unit.
- which objects, visible within it, it has to obtain from other units; these are termed the objects applied by the unit.

The identifiers associated with those objects are indicated in the defines part and the applies part respectively of the interface.

- Depending on the particular programming language, such an interface may be explicit, or it may be implicit.
- As an example, Fig. 1 is typical for the visibility structure needed in compilers and other large programs.

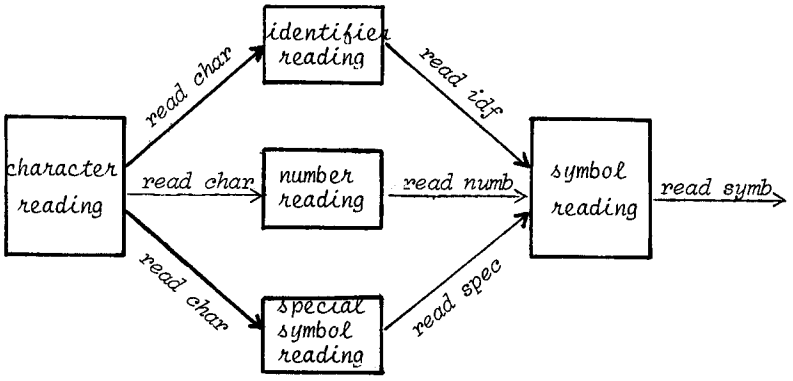
1.2. Visibility Rules

We want to study means for restricting the visibility of objects between units (as a generalization of scope rules) through the use of interfaces. Denoting the set of identifiers of objects defined by some unit A as δ_A and similarly those applied by A as α_A, the visibility relation

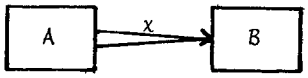


means: x is visible in $B \iff x \in \delta_A \wedge x \in \alpha_B \wedge$ x is visible in A

Figure 1

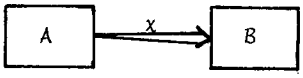


We will indicate the situation where the defines part of A implicitly passes all objects visible in A by

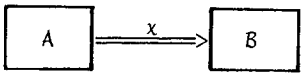


(x is visible in B $\Longleftrightarrow$ $x \varepsilon \alpha_B \wedge$ x is visible in A)

Similarly we will indicate by

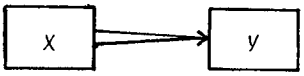


the situation where the applies part of B implicitly passes all objects defined by A, and by



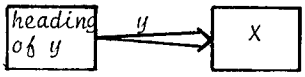
the situation where there is no restriction at all on the visibility.

In block-structured languages, the visibility between a block X and another block Y nested directly in it is of the type



since, apart from the possibility of declaring in Y objects with the same identifiers, all objects defined in X are visible in Y. The fact that the defines part of X implicitly passes all objects visible in X, and that the applies part of Y is also implicit is at the heart of a number of problems associated with block-structure [Wulf/Shaw 1973].

The visibility between a block X and the heading of a procedure y nested directly in it can be seen as an example of



since of all objects defined in the heading only the procedure name is explicitly made available to X, whereas in X there is no further restriction on visibility. A similar relationship holds between a block X and a class Y nested directly within it, some of whose constituents are hidden.

- If $\alpha_A \cap \delta_A \neq \emptyset$ it makes sense to study $\xrightarrow{+}$, the controlled level jumping.
- It may make sense to forbid cycles in order to disallow recursion, as e.g. in concurrent PASCAL [Brinch Hansen 1974].

1.3. Module Structure

From here on, we will use the term module to denote either a unit, or a collection of units and modules. The reasons for not introducing this term earlier are

- the fact that the term module is so overloaded that its use may be very misleading if it is not specified clearly what meaning of the word is intended
- the desire to focus first on the relations between smaller constituents of a program.

We now need the concept since we now want to discuss the overall structure of programs "in the large" [DeRemer/Kron 1974].

1.3.1. Structural Archetypes

We will discuss here two general forms of module structure from which others can be derived by nesting (see 1.3.2.)

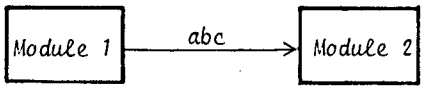
1.3.1.1. Chaos

Very widely used in assembly languages is a structure where, potentially, objects are visible anywhere, i.e., objects defined by some module can be applied by all other modules. We will term this state of affairs chaos, or at best controlled chaos.

As an example, the specifications

```
        :                       :
EXTERNAL  ABC           ENTRY   ABC
        :                       :
```

may establish the visibility relationship



We will term this structure chaos over $\longrightarrow$.

- It can be argued that, because of the close control over interfaces, this structure is preferable over block-structure for large programs
- Notice that the directed graph of the structure may be non-coherent, so that it can be split up into disjoint graphs.

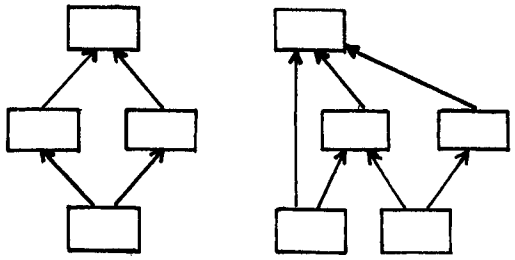
1.3.1.2. Hierarchy

Very widely advocated [Parnas 1972] is hierarchical structure, where the visibility graph can be divided into levels such that

a) from level *0* no object is visible

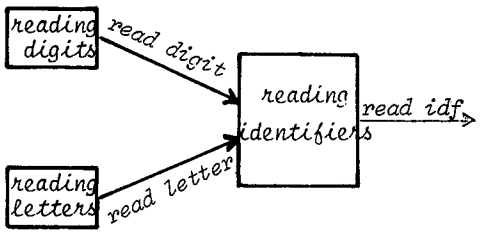
b) from level i only objects on levels $<i-1$ are visible, and at least one object is in fact applied from level $i-1$.

Example:



Such a weak hierarchy allows jumping over any number of levels.

- We can obtain a strong hierarchy by insisting that from level i only objects on level $i-1$ are visible. This leaves open the possibility of controlled level jumping through $\xrightarrow{+}$ (see fig. 1 in [Robinson et al 1975]) but is more difficult work, and such level jumps will therefore hopefully be minimized by the programmer.
- Block-structure is a hierarchy over $\Rightarrow$ with the property that every module has only one ingoing arrow (but for a root module). It allows the expression of the situation where one module provides the environment for one or more other modules.
- From the point of view of information hiding, a very interesting structure is anti-block-structure, where each module has an arrow to only one module, as in the example

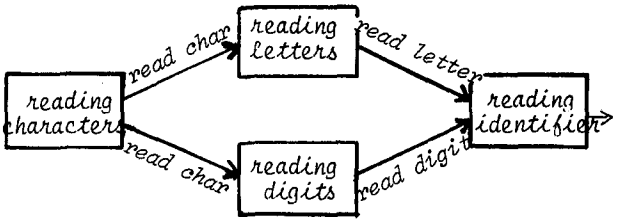


It allows the expression of the situation where many modules together equally contribute to the definition of a new one. This structure can be modelled in ALGOL languages by declaring the procedures *read letter* and *read digit* local to the procedure *read idf*, which shows that ALGOL allows, besides block-structure, some measure of anti-block-structure.

- Using block structure instead, the two procedures *read digit* and *read letter* would have to

be global to *read idf*, with the consequence that they are visible wherever *read idf* is visible.

- Anti-block-structure clearly has some advantages over block-structure. It is conjectured that part of the success of the class concept comes from its use of anti-block-structure.
- Anti-block-structure is not, however, adequate for expressing the situation that both *read digit* and *read letter* make use of a procedure *read char* which is used nowhere else, and should not be visible anywhere else.

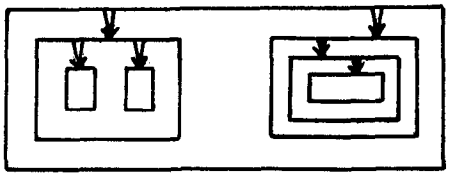


Using ALGOL, the procedure *read char* would have to be local to both *read letter* and *read digit*, which would only be possible by copying. Mixing anti-block-structure with block-structure by making *read char* global to both makes its visibility too large again.

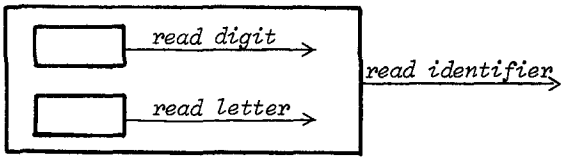
3.2. Nested Modules

Various forms of structure can be obtained by nesting units or modules within modules. It is suggested that the most profitable way of envisaging the structure of large systems is by seeing them as structures nested within structures.

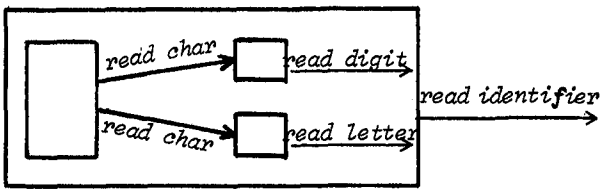
- Block-structure can most simply be depicted as



- also anti-block-structure can easily be depicted



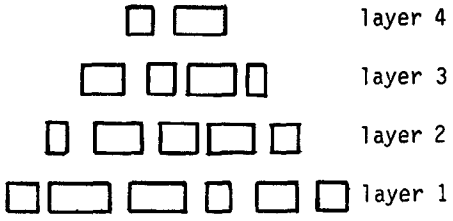
- It is possible to have a hierarchy within a hierarchy



which gives an adequate solution to the structuring problem used as a touchstone in the previous sections.

1.3.3. A Semantical Model of Module Structure

In this section one semantical model justifying the choice of a specific module structure is outlined. We will consider a program as an abstract machine [Waite + Poole], consisting of layers of units, e.g.



In a compiler, these layers could correspond to the character level, token level, symbol level and syntactic level respectively. The layers correspond to levels of abstraction: the units within one layer conspire in some fashion to present a more abstract, a higher level face than the underlying layer. One layer defines an abstract instruction set and abstract data.

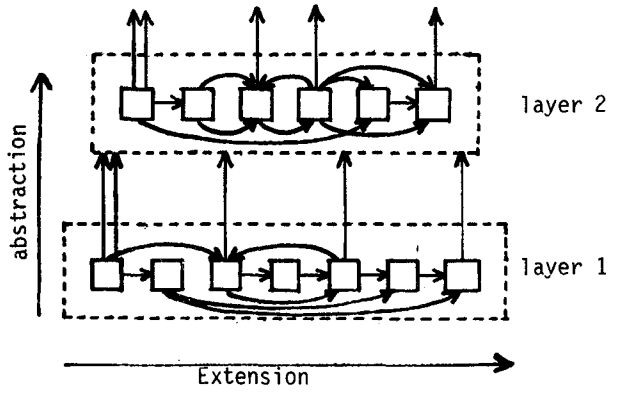
1.3.3.1. The CDL Model of Modularity

Units within one layer need access to

- the immediately preceding layer
- other units and data within one layer

If the vertical direction in the diagram represents abstraction, the horizontal direction can be called extension, since the units within one layer incrementally extend the power of the machine whilst remaining at the same level of abstraction. Notice that in particular recursion (on one same level of abstraction) may be necessary, which precludes a strict hierarchical structure.

Between layers, in the direction of abstraction, a strict hierarchy is enforced within one layer; in the direction of extension, less restrictions must be imposed on visibility.



As a consequence of these observations, we are presently gathering experience with a model of modularity based on the distinction between vertical abstraction interfaces and horizontal extension interfaces, delimiting very precisely the visibility of objects. The modularity structure is that of a strict hierarchy of layers over → with, within one layer, chaos over →. This is an alternative to the usual transparent hierarchy.

A next question, which should be answerable from the model is: what can be passed across an interface.

1.3.3.2. Objects

The following sorts of objects are to be considered:

- algorithms
  - procedures (closed / open)
  - functions
  - operators
  - macros
- data
  - variables
  - constants
- types

The possibility of passing algorithms across interfaces is essential, as well for extension as for abstraction.

Regarding constants, it can be argued that a constant (e.g. *17*) hardly ever stands for itself (viz. the follower, in the sense of Peano, of *16*) but has some specific meaning as, e.g., the encoding of a character. By this argument, a constant appearing at some level of abstraction cannot possibly mean the same thing at a different level of abstraction, and therefore should not be passed between layers. Thus, the encoding of the end-of-file-character should never be made available to the symbol section, for fear of some clever programmer "optimizing" by using the end-of-file-character in the place of the end-of-file-symbol. (This applies a fortiori to variables.) Thus, we might forbid the passing of constants between layers by not allowing them to appear in abstraction interfaces.

After extensive practical experience, this argument, to our present point of view, seems pedantically right, but not practicable with present-day programmers, who have not the foggiest notion of modular programming. Only when a new programming style has been evolved, can such consequences be drawn. The argument is given here to solicit thought, response and further experimentation.

In the model which we arrived at in the language CDL, after many intermediate models, constants can appear in abstraction interfaces, whereas variables cannot. In extension interfaces, any object may appear.

The passing of types across interfaces, which is analogous to the passing of algorithms, will be studied more closely in the next chapter.

2. Types

A type is a collection of values, characterized in some way - be it by enumeration, axiomatically, or through the set of algorithms applicable to instances (values) of that type and the properties of those algorithms. It is the latter algorithmic characterization which we will adhere to. The purpose of the following discussion is to explore the relationship between visibility and data abstraction.

2.1. Type and Realization

Ever since ALGOL W, programming languages of the ALGOL family have contained facilities to declare types. We will write our examples in the experimental programming language SLAN, which is an obvious descendant of ALGOL, so that the examples should be self-explanatory.

A type-declaration associates a type-name with a fine-structure, e.g.,

```
type compl      = struct (real re, im);
type complex    = struct (real im, re);
type string     = struct (int length, [1 : 128]
                                  char text);
type array      = [1 : 10, 1 : 10] real;
type intmodthree = int;
```

The fine-structure, i.e. the part after the equals symbol, serves two purposes:

- it specifies how an object of that type is represented
- it specifies the primitive access algorithms to an object of that type

As an example the given declaration for *string* makes the following primitive access algorithms available:

$$\text{string var } t; \Longrightarrow \begin{cases} t \\ t\ .length \\ t\ .text \\ t\ .text\ [i] \end{cases}$$

On the basis of the primitive access algorithms, further algorithms working on an object of that type are constructed. In many situations, it is imperative that the use of the primitive access algorithms is restricted and localized as far as possible. This

- simplifies proofs of correctness
- is essential for data abstraction
- is essential for security
- is necessary for uniformity of referents

Thus, we want to restrict the visibility of fine-structure.

2.2. Purposes of a Type

The purposes, for which a type is used in a programming language, can be summarized as follows, giving examples of each:

declaration of an object of that type
- without generating a new object

```
real const pi = 3.142;
real var max  = if a>b then a else b fi;
```

- with generation of a new object

```
complex var z;
```

denotation of an object of that type
- historical denotations for some privileged types

```
217; "abc"; -1.3125_10 -9; true
```

- general denotations for structure or row types

```
complex (0.712, -0.712)
row (1, 0, 0, 0)
```

identification of generic objects
- generic operators or procedures
  - declared explicitly

```
op x = (matrix const a, b) matrix: ~~
```

  - declared implicitly
    - visible

```
:=   and  [ ]
```

    - invisible (e.g. coercions)

```
real var x;  x := x + 1; ...
```

static check of domain
- case clause, conformity case clause

```
case colour in red: ...; green: ... esac
```

or

```
case lispcell in
     (atom x): ~~;
     (ref cell y): ~~
esac
```

- static check of assignment- (or parameter-) compatibility

Abstract types must be made available in such a way that it is possible to declare variables of that type, denote constants of the type, identify generic operations (for the very least the assignation) and perform static checks.

2.3. Abstract Types and Interfaces

The abstraction of a type from its realization can be done across an interface between a defining en-

vironment, where the name and the fine-structure of the type is fully known and can be made use of, and an applying environment where only the type-name is available, and any knowledge about the fine-structure cannot be made use of, as in figure 2

In SLAN, only in the reach of the type declaration is its fine-structure visible; it cannot be made visible to other units, since it cannot be passed by means of an interface. Appropriate algorithms can be made visible instead.

All questions of security, as well as the burden of the proof of correctness of the realization of type, are put on the shoulder of the man who writes the defining environment. The underlying attitude is that the definer of the type is an infinitely good programmer, whereas the applier is infinitely mediocre, bad or malicious. Of course this argument is repeated at every interface.

A corollary is that the user must never be forced to write something which depends on the realization of the type. As an example, the type definitions

```
type compl = struct (real re, im);
type compl = struct (real im, re)
```

differ only trivially, and any program written for the one realization will work for the second realization except for one point: any denotations, whose value depends on the ordering of fields, e.g.,

```
compl (0.712, -.712)
```

will be wrong under the other realization.

All the purposes of a type, outlined in the previous section, can be fulfilled but for one - the general denotation, which is fine-structure dependent. We therefore forbid the use of those general denotations outside of the defining environment.

2.4. Denoting Instances of Types

We have to provide a way for the user to denote values (instances) of a type. Ultimately, all objects are realized as a collection of values of one of the historical types (int, real, bool, maybe char and string) for which an obvious denotation is to be available. We can do this without the introduction of any new concept by making use of the concept of algorithm: for each type, supply a denotation procedure to denote new instances of the type, which can be parametrized to supply bounds or initializations.

Examples in the form of initialized declarations:

```
vector var t := vec (100);
matrix var x := zeromat (n, n);
stack var mystack := stack (200);
```

This solution gives full algorithmic control over the creation of new instances. Since the denotation procedure can be an open procedure, no inherent inefficiency is to be feared.

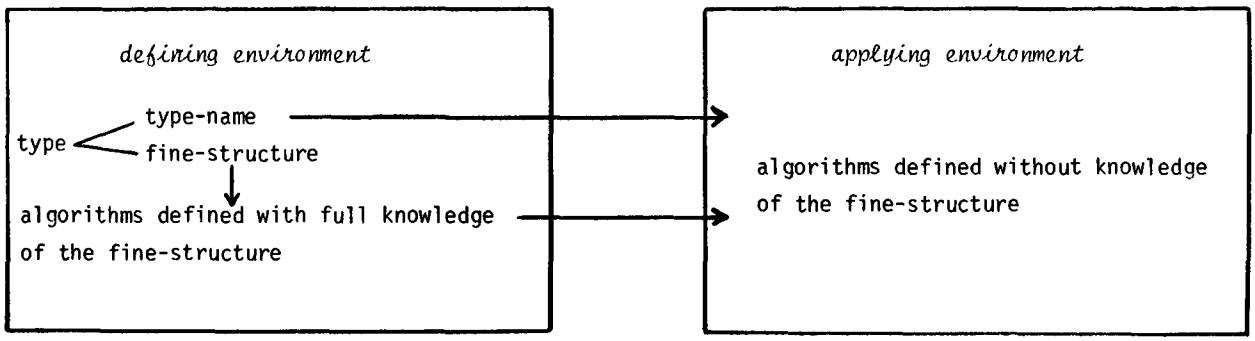
It is in this fashion that matrices and vectors are introduced in SLAN, where their defining modules are part of the library.

2.5. An Example: Matrices

Let us consider the (somewhat rhetorical) question: What is a matrix?

One answer is to say, a matrix M consists of a

Figure 2



square array of elements $M_{i,j}$ selected by pairs of integers $i,j \in [1:n]$ through a subscription algorithm. The essential point is the existence and properties of the subscription algorithm, not the representation as a square array of elements.

Of course the subscription algorithm can be obtained very simply by declaring

```
type matrix = [1 : n, 1 : n] real
```

which will allow

```
matrix m; ... m[i, j] ...
```

However there may be situations, where this representation is unsuitable, such as:

a) the matrix is large, but symmetric. Memory can be saved by declaring

```
type matrix = [1 : n * (n + 1) ÷ 2] real
```

with subscription algorithm

```
m [if j ≤ i then i * (i-1) ÷ 2 + j
           else j * (j-1) ÷ 2 + i fi]
```

b) the matrix is very large, but a band matrix of width *width*

```
type matrix = [1 : n, -width : +width] real
```

with subscription algorithm

```
m [i, j - i ]
```

c) the matrix is very large but sparse

```
type matrix = [1 : n] column;
type column = struct (real elem, ref column
                                    next)
```

where the subscription algorithm is left as an exercise to the reader.

All of these are, under specific circumstances, adequate representations of matrices. The choice between one representation and the other must be possible without invalidating any programs using matrices, but this is impossible because of the widely varying subscription algorithms - unless care has been taken to separate the representation from the access mechanism.

In all of these cases, subscription should take place by the use of suitable access operators, e.g. the pair

```
op sub = (matrix const x, [1 : 2] index i)
                                real const: ~~;
op sub = (matrix var x, [1 : 2] index i)
                                real var: ~~;
```

where only within the body of the operators, knowledge of the fine-structure of the type *matrix* is necessary. All programs using matrices will subscribe by

```
m sub (i, j)
```

independent of the actual representation of matrices. Notice that this solution, besides allowing complete freedom in the realization of an abstract type, also satisfies the uniform referent principle. To answer the question at the start of this section, an American folk saying can be quoted (somewhat censored)

"If it looks like sh.t, it smells like sh.t, and it tastes like sh.t, then it must be sh.t."

which is an admirable formulation of the concepts underlying abstract data types.

3. Comparison with the Class Concept

The ideas outlined in the previous sections reduce the adequate treatment of abstract data types to the question of visibility, a problem which in large systems has to be faced anyway. Data abstraction can be handled completely analogously to algorithmic abstraction. The duality between those two forms of abstraction should be clearer in this approach than in any approach which introduces a special language construct, viz. classes, to deal with data abstraction. Any program which can be formulated in terms of classes can quite simply be rewritten in terms of type declarations and module interfaces, but the reverse is not true, as we hope to point out in the next sections. As an example for comparison, we show here declarations for a character stack in SIMULA and SLAN. (see Fig. 3)

3.1. Some Problems with the Class Concept

As originally introduced in SIMULA, classes do not allow the (partial) hiding of fields and algorithms. The need for this mechanism was felt only later, but is at the heart of our approach. In the given SIMULA example, nasty side effects and bad program-

Figure 3

SIMULA:

```
class charstack (length); integer length;
begin character array cell [1 : length];
    integer pointer;
    procedure push (item); character item;
        begin if full
              then stack overflow;
              pointer := pointer + 1;
              cell [pointer] := item
        end of push;

    character procedure pop;
        begin if pointer = 0
              then stack underflow;
              pop := cell [pointer];
              pointer := pointer - 1
        end of pop;

    boolean procedure empty;
        empty := pointer = 0
    boolean procedure full;
    full := pointer ≥ length;
    pointer := 0
end;

    ref (charstack) stack;
stack := new charstack(100);
while ¬ full.stack do push.stack ("x");
```

SLAN:

```
module char stacking defines charstack, push,
            pop, empty, full, new charstack:
    type charstack = struct (int pointer,
                length, array char cell);
    alg push (charstack var s, char const
                                    item):
        if full (s)
        then stack overflow fi;
        s.cell [s.pointer incr 1] := item
    end alg push;

    alg pop (charstack var s) char:
        if empty (s)
        then stack underflow fi;
        pop := s.cell [s.pointer];
        s.pointer decr 1
    end alg pop;

    alg empty (charstack var s) bool:
        s.pointer = 0
    end alg empty;

    alg full (charstack var s) bool:
        s. pointer = s.length
    end alg full;

    alg new charstack (int const length)
                                charstack:
        charstack (0, length, array (length)
                                    char)
    end alg new charstack
end module char stacking;

charstack var stack := new charstack (100);
while ¬ full (stack) do push (stack, "x");
```

ming style can be the consequence of the visibility of the fields *pointer* and *length*.

In SIMULA an instance of a class is a structure embracing all data fields together with a set of algorithms. This limits the possible realizations of a type to structures, which is unnecessarily restrictive and obscures the essential simplicity. It is by no means clear why

```
type vector = array real
```

or

```
type intmodthree = count 3
```

should be packaged as structures.

Fields of a class can be accessed by means of the dot-notation for field-selection as long as they are data fields or parameterless procedures, but for fields which are parametrized procedures sud-

denly brackets are needed, as in

push.stack ("x")

This can be seen as non-uniformity of referents [Geschke/Mitchell 1975].

The most serious problem with classes comes from the basic idea to encapsulate the definition of one type: If one wants to define two or more related types, say *ford* and *opel*, as well as conversion algorithms between them, he cannot define the two classes by one declaration, hiding all intimate details. Neither can he define two separate classes *ford* and *opel* each hiding their intimate details, because for the definition of either, intimate knowledge of the other is needed. Making the intimate details of the one visible to the other makes them available to any user of the other, because of the block-structure. Defining the one local to the other also does not give a solution (unless one bends the scope rules until this problem is solved [Hoare 1975]). An altogether different mechanism giving very precise control over visibility is needed - which was our thesis.

The visibility structure needed is either the one in Figure 4, or the simpler one in figure 5.

3.2. Conclusion

We hope to have shown the important relationship between visibility and data abstraction, and to have demonstrated that, by providing a general control over visibility, the need for special language constructs for data abstraction other than the type declaration can be obviated.

In relation to the class concept, the use of which seems to become a consensus in the PASCULA tradition, the advantages claimed for our scheme are:

- It is conceptually simpler and more general, because it subsumes the treatment of module structure
- It allows a symmetric treatment of algorithmic abstraction and data abstraction which should lead to better programming style
- It allows a solution of the uniform referent problem
- It allows the simple and secure treatment of related types and conversions between them.
- The use of parametrized denotation procedures allows a simple treatment of subtypes.

Figure 4

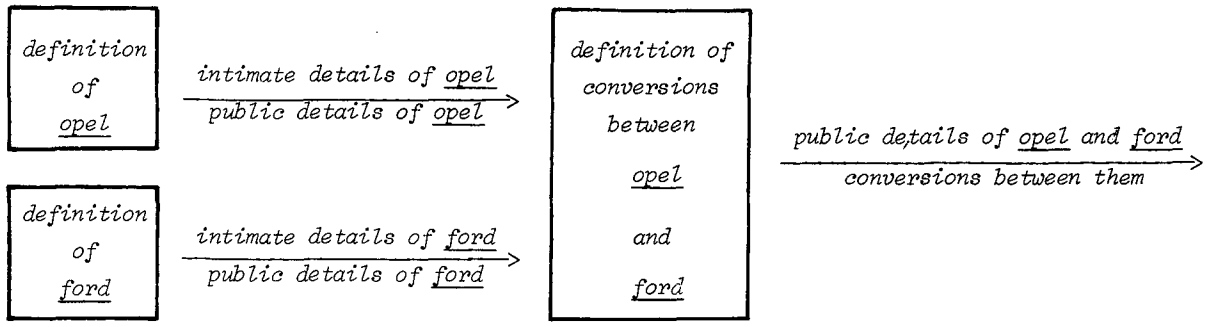
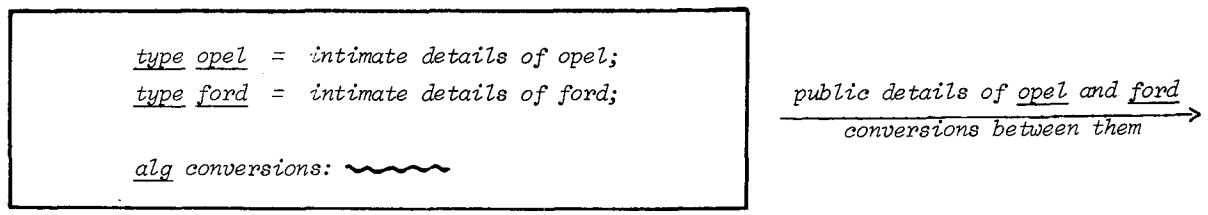


Figure 5



It is not the intention of this paper to deprecate the merits of the class concept, but to provoke an awareness of alternatives and the interest to investigate in a wider framework the concepts involved.

Acknowledgements

My thanks go to my colleages of WG 2.4, especially Jean Ichbiah, who seems to have coined the term visibility for the systems implementation language LIS [Ichbiah 1974], to Frank DeRemer, whose ideas are closely related [Deremer 1975], and to Bernd Krieg-Brückner [Krieg 1974] for many discussions.

The CDL model was designed, implemented, evaluated and redesigned a number of times by Jean-Pierre Dehottay, the ideologist of abstract machines, and Michael Stahl, both from the Technical University of Berlin. SLAN is a common effort of a number of researchers at the Technical University of Berlin.

REFERENCES

Brinch Hansen, P.: Concurrent PASCAL, a Programming Language for Operating System Design; Information Science Technical Report No. 10, California Institute of Technology, April, 1974.

Dahl, O.J., Myhrhaug, B. and Nygaard, U.: The Simula 67 Common Base Language; Norwegian Computing Center, Oslo 1968.

DeRemer, F. and Kron, H.: Programming-in-the-Large versus Programming-in-the-Small, Proceedings of the International Conference on Reliable Software; SIGPLAN Notices Vol. 10 No. 6, June 1975, presented to WG 2.4 in Dec. 1974.

Geschke, C.M. and Mitchell, J.G.: On the Problem of Uniform References to Data Structures, Proceedings of an International Conference on Reliable Software; SIGPLAN Notices Vol. 10 No. 6, June 1975

Hoare, C.A.R.: The Structure of an Operating System, working material for the International Summer School on Language Hierarchies and Interfaces, Munich, August 1975.

Ichbiah, J.A.: Visibility and Separate Compilations Proceedings of the IFIP WG 2.4 Meeting in La Grande Motte, France, May 1974.

Koster, C.H.A.: Using the CDL Compiler Compiler, in: Compiler Construction, an Advanced Course; Springer Lecture Notes in Computer Science 21, 1974.

Koster, C.H.A.: Beating the Global, Proceedings of the IFIP WG 2.4 Meeting in La Grande Motte, France, May 1974.

Krieg, B.: Increasing the Flexibility and Reliability of the Class Concept; Bericht Nr. 7429, Technical University of Munich, presented to WG 2.4 in December 1974.

Liskov, B.H. and Zilles, S.: Programming with Abstract Data Types, Proceedings SIGPLAN Conference on Very High Level Languages; SIGPLAN Notices Vol. 9 No. 4, April 1974.

Parnas, D.L.: On the Criteria used in Decomposing Systems into Modules; CACM Vol. 15 No. 12, December 1972.

Poole, P.C. and Waite, W.M.: Portability and Adaptability, in: Advanced Course on Software Engineering; Lecture Notes in Economics and Mathematical Systems 81, Springer 1973.

Robinson, L., Levitt, K.N., Neumann, P.G. and Saxena, A.R.: On Attaining Reliable Software for a Secure Operating System; SIGPLAN Notices Vol. 10 No. 6, June 1975.

Ross, D.: Uniform Referents: An Essential Property for a Software Engineering Language, in: Tou (Ed.), Software Engineering, AP 1970.

Wulf, W.A. and Shaw, M.: Global Variable Considered Harmful; SIGPLAN Notices Vol. 8 No. 2, 1973.

Wulf, W.A.: ALPHARD: Toward a Language to Support Structured Programs, Carnegie-Mellon University, April 1974.