# Nesting in an Object Oriented Language is NOT for the Birds

*P. A. Buhr* — C. R. Zarnke***

* Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1

** Waterloo Microsystems Inc., 175 Columbia St. W., Waterloo, Ontario, Canada, N2L 5Z5

*ABSTRACT*

The notion of nested blocks has come into disfavour or has been ignored in recent program language design. Many of the current object oriented programming languages use subclassing as the sole mechanism to establish relationships between classes and have no general notion of nesting. We argue that nesting (and, more generally, hierarchical organization) is a powerful mechanism that provides facilities that are not otherwise possible in a class based programming language. We agree that traditional block structure and its associated nesting have severe problems, and we suggest several extensions to the notion of blocks and block structure that indirectly make nesting a useful and powerful mechanism, particularly in an object oriented programming system. The main extension is to allow references to definitions from outside of the containing block, thereby making the contained definitions available in a larger scope. References are made using either the name of the containing entity or an instance of the containing entity. The extensions suggest a way to organize the programming environment for a large, multi-user system. These facilities are not available with subclassing, and subclassing provides facilities not available by nesting; hence, an object oriented language can benefit by providing nesting as well.

**Key words:** Object-Oriented, Nesting, Block Structure, Programming-in-the-Large

## 1. Introduction

The notion of block structure, as in nested blocks (nesting), has come into disfavour or has been ignored over the last decade. This is reflected in the fact that many of the current object oriented programming languages use subclassing as the sole mechanism to establish relationships between classes (Smalltalk [GOLDB83], LOOPS [BOBRO83], C++* [STROU86], Objective-C [COX86]) and have no general notion of block structure.

*C++ allows textual nesting, but its meaning is as if the contained class were not nested at all; instead, the contained class is considered to appear at the same level as the containing class. Hence, in C++, references from the contained class to variables in the containing class are not allowed; this fails to meet the normal criterion for block structure.

However, there are a few important exceptions. Both Simula [SIMULA87], the first programming language to provide an object oriented programming style, and its successor BETA [KRIST87] support the idea of block structure, in particular, nesting of class definitions, as well as the programming language LOGLAN82 [BARTO84]. However, the authors of Simula realized that block structure in Simula was too restrictive, since a class can only be referred to in the block where it is declared, and extended the concept of block structure in BETA by allowing references to a class from outside the class in which it is declared [MADSE87].

Blocks, which are formed by several programming language constructs such as BEGIN blocks, procedures, classes, packages, etc., provide two main functions: to introduce new name spaces into the definitional structure of a programming language and to introduce a new level for the control of visibility of names within a name space. Subsequent activation of a block causes allocation of all the data items contained within it. Because block structure provides a general mechanism for introducing and structuring name spaces, each program has its own name space and nesting gives the ability to create new ones. In this paper, we wish to review criticisms against block structure and to rebut them, to discuss briefly the extensions to block structure that were introduced in Simula and extended in BETA, and to extend these ideas even further with the intent of providing a mechanism for structuring an entire system using blocks. (The framework for this discussion assumes a unilingual programming environment [WEGBR71, TEITE78, GOLDB83, TEITE84, HEERI85, ATKIN85, MEYER87, BUHR87].) Our extensions depart substantially from traditional notions in block structured languages.

## 2. Normal Block Structure

The normal Algol scope rules preclude references to any definitions not within the current block or one of its containing blocks. The basic reason for this restriction is to assure what we call **existence**, that is, that at execution time the block frame in which a variable is declared has been instantiated. Existence is not an issue for entities such as constants and simple types. The main criticisms against block structure given in [CLARK80] and [HANSO81], many of which are addressed in [TENNE82], focus on the problems of excessive visibility of names in containing blocks. These visibility problems can usually be handled by adding appropriate facilities to the programming language so that importing and exporting of names is under explicit control. Our major complaint is the opposite situation, that is, the inability to refer to definitions in blocks other than itself or its ancestors. It is our contention that items in such blocks can be referenced and yet still be able to guarantee *statically* that they or data they use will exist at execution time (i.e. the compiler can guarantee closure).

## 3. Extending References to Components of a Block

Referring to fields of a record is the simplest example of the kinds of references that we are talking about. The reason that internal references are possible for a record is that a field is selected from an instance of the record, which contains instances of all the fields of the record. Simula augments the concept of a data record to contain procedures as well as data items in the CLASS construct, collectively called **attributes** in Simula but which we prefer to call **components**, and uses the same syntax to refer to both procedures and data items, for example:

```
CLASS xxx
   VAR v : ...
   PROC p(...) ...
END CLASS

VAR x : xxx
x.v        ←┬— references to components inside a block from outside
x.p(...)   ←┘
```

Procedures defined in the class must execute in the environment (context) of the class. As a result, a procedure must be qualified by an instance because the scope rules allow global references from the procedure to the class instance.

The next extension to class definition is in BETA, which allows references to class components of a CLASS, for example:

```
CLASS xxx
   CLASS yyy
      ...
   ...

VAR x : xxx
VAR y : x.yyy      ← qualify internal class yyy by an instance of xxx
```

Extending references to classes within a block makes them a true component by enlarging their visibility to that of other components. That is, conventionally a class is only accessible within the containing block where it is defined and now it is also accessible outside of it. Qualification is allowed in the type specification of a declaration, where normally only a simple type name is allowed. The qualification uses conventional syntax and its meaning is consistent with the meaning of qualification in executable statements, that is, to select a component from a definition. The meaning of this kind of reference is to create an instance of the contained class and to use the qualifying instance as its context. In the above example, if yyy referred to components in xxx (or even outside of xxx if this was allowed), the contexts for those components are available. Both Simula and LOGLAN82 allow class definitions to be nested within other class definitions but the contained class definition cannot be referred to outside the containing one.* BETA is less restrictive, but it does limit references to one level down in the textual (static) definition structure (see also [HOUSE86]).

*This is the conclusion we have drawn from the one reference manual that we have on LOGLAN82. We are open to correction on this point about LOGLAN82.

## 4. Multiple Dependent Inner Class Instances

As discussed in [MADSE87], references to nested classes provides the ability to have multiple instances of a contained class associated with a single instance of an containing class, for example:

```
VAR x : xxx
VAR y1, y2, y3 : x.yyy
```

Here there are three instances of class `yyy` (`y1, y2, y3`) associated with the instance of class `xxx` (`x`). This allows the children to carry out actions jointly with their common parent. Both C++ and Smalltalk have a storage class (`static` and `classVariableNames`, respectively) which allow variables to be created that are unique to a class, independent of the number of instances of the class that are created; or in the case of Smalltalk, the number of subclasses that create it implicitly. This *does* mimic having multiple children for a single parent by having the static variables be considered to be the parent. Unfortunately, this *does not* generalize to allow multiple instances of the containing class with multiple instances of the contained class in each, for example:

```
CLASS xxx
   VAR x, y, z : ...        ← local variables of xxx
   CLASS yyy ...
   ...

VAR x1, x2: xxx                        ← create 2 parent classes
VAR y1x1, y2x1, y3x1 : x1.yyy    ← create multiple children in each parent
VAR y1x2, y2x2, y3x2 : x2.yyy
```

Here there are separate instances of `x`, `y` and `z` created for each instance of `x1` and `x2`, and `y1x1`, `y2x1` and `y3x1` access one set in `x1` and `y1x2`, `y2x2` and `y3x2` access the other set in `x2`. If `x`, `y` and `z` had static storage class, only one instance of them would be created for both `x1` and `x2`.

Like Madsen, we feel dependent instances are an extremely useful and powerful facility in an object-oriented programming language. Madsen gives several examples of the use of nested classes in [MADSE87]. The following are more examples which we feel are important in the construction of large systems.

### 4.1. User-Definable Variable-Size Data Structures

Multiple dependent instances can provide a programming technique that is a compromise between universal garbage collection and no garbage collection. The technique is as follows: a containing class can act as a separate heap for storage allocation and its internal class(es) can be the types that are allocated in the heap. This is similar to the idea of a `collection` in Euclid [LAMPS77] or an `area` in PL/I [IBM81]. The following outline is a simple example of how a variable-length string data type might be defined using dependent instances; a more general scheme has been used to implement a variable-string class in C++.

```
CLASS VstrHeap(NoOfVar, NoOfChar : INT)
   RECORD VstrDesc                                   descriptor for each VstrVar
      length : 0..NoOfChar                             length of string
      posn   : 1..NoOfChar                             position in VstrText
   END RECORD
   VAR VstrVar  : [1..NoOfVar] VstrDesc              indirect pointers to strings (handles)
   VAR VstrText : [1..NoOfChar] CHAR                 contains text for all strings

   CLASS Vstr
      VAR VarNo : 1..NoOfVar                         subscript of descriptor

      INFIX '←'( ... ) ...                           definition of assignment
      PROC substr( ... )  ...
      PROC length( ... )  ...
      PROC concat( ... )  ...
   INITIALIZATION
      VarNo ← AllocVstrDesc()                        get descriptor entry
   TERMINATION
      FreeVstrDesc(VarNo)                            free descriptor entry
   END CLASS

   PROC AllocVstrDesc() ...
   PROC FreeVstrDesc() ...
END CLASS

VAR vh : VstrHeap(20, 5000)  create heap
VAR a, b, c : vh.Vstr        create variable-length string variables
...
a ← 'abc'
b ← 'xy'
c ← a.substr(2, b.length)
```

Clearly, there are drawbacks to this approach such as having to explicitly declare the containing class, and should multiple heaps be created, instances in one are *not* readily interchangeable with those in another; however, universal garbage collection has its drawbacks, too, such as the cost in execution time, storage space and system complexity. It is felt that the compromise provided by multiple dependent instances is a reasonable one. It provides flexibility to the type definer in creating sophisticated data structures that are dynamically created at a reasonable cost in execution time, storage space, and convenience to the user of the data type and yet do not require explicit freeing, thereby eliminating the dangling pointer problem.

## 4.2. User-Definable File Definitions

One of our goals is to be able to define, in an object-oriented paradigm, types that correspond to files in traditional systems. This is discussed in detail in [BUHR86] where only three new programming language constructs are needed to construct a file-like definition; these definitions rely heavily on nesting. Only the OBJECT construct will be mentioned here. An OBJECT is similar to a CLASS except instances of it are not directly accessible after declaration and its storage is not managed directly by the compiler. The skeleton structure for a sequential file definition is:

```
OBJECT SeqFile(RecordType : TYPE)  ← polymorphic type definition
   VAR FirstRecord, LastRecord : REF RecordType
   declaration for the records and routines to manage them in the file
INITIALIZATION
   FirstRecord ← LastRecord ← NULL
TERMINATE
   termination code, none for SeqFile
END OBJECT
```

Unlike variables in a block that are accessible immediately after declaration, files are not accessible after they are declared because they are created in a separate memory that is not always accessible by the hardware. Files must be made accessible either implicitly (Multics, Smalltalk, IBM System 38) or explicitly. We feel that explicit access is an important facility to have. As well, information must be created and maintained about each user's access to a file. For example, when reading from a file, a pointer to the current record read must be maintained for each reader. However, this access information must not be stored in the file because it may be difficult to find and adjust it correctly after system failure.

While current object-oriented programming languages do not allow the definition of files, some mechanism exists to access the system's files and, in general, it is done in the following way. There is a CLASS for the access information which contains all the access routines and access related variables. This separates the access information from the file information and allows multiple access declarations for multiple accessors to a particular file. When creating the access information, it must be tied to the particular file that it is making accessible, for example;

```
VAR f  : SeqFile(INT)     ← or some other mechanism to create a new instance of a file
VAR fa : AccessInfo(f)    ← pass the file as a parameter to establish a connection
                            between the access information and the file
fa.write(3)
i ← fa.read()
```

Terminating access is done in the termination code for the access structure. What we dislike about this approach is that the system must still provide some explicit mechanism that can be invoked by the initialization code and termination code to cause the file to become accessible/inaccessible by the hardware (e.g. creation/freeing of page tables or system buffers). As well, there is only a coincidental connection through the parameter to the access information that establishes the fact that the file will be made accessible. The compiler cannot be aware that this connection will ultimately make the file accessible/inaccessible and hence it cannot make any inferences about this important aspect of the behaviour of file-like objects.

An alternate solution is to define the access information class in the object. Then, the way to access a file is through the nested class because it contains all the access routines. When a nested class is used in this way it is called an **access class**, for example:

```
OBJECT SeqFile(RecordType : TYPE)
   VAR FirstRecord, LastRecord : REF RecordType
   declaration for the records in the file

   CLASS SeqAccess
      ESCAPE EndOfFile
      VAR CurrentRecord : REF RecordType

      PROC read RETURNS r : REF RecordType   read a record
      PROC reset                             position to 1st record
      PROC write(r : RecordType)             write a record
      PROC update(r : RecordType)            replace a record (same length)
      PROC recreate                          destroy all records
   INITIALIZATION
      CurrentRecord ← NULL
   TERMINATION
      termination code, none for SeqAccess
   END CLASS
INITIALIZATION
   FirstRecord ← LastRecord ← NULL
TERMINATE
   termination code, none for SeqFile
END OBJECT
```

The access class would be used in the following way:

```
VAR f  : SeqFile(INT)
VAR fa : f.SeqAccess

fa.write(3);
i ← fa.read()
```

After the instantiation of SeqAccess, the SeqFile must be made accessible so that SeqAccess and its routines can refer to data items in it. Similarly, de-allocation of the access class makes the object inaccessible. Thus, the duration of object access is the duration of the access class. The actual mechanism for making the file accessible/inaccessible is not explicit but implicit through creation and freeing of the access class, which is something that the compiler has full knowledge of and hence, can possibly take advantage of. Thus, the access class provides the necessary storage creation, duration, and scope to implement explicit accessibility and support concurrent accessors, and it is tied into the programming language. Other issues dealing with serializing concurrent access to the object's internal resources when using nesting are discussed in [BUHR86, KRIST87].

## 5. Controlling Visibility

Traditional blocks (BEGIN, procedure, etc.) prevent access to their data components from outside because the block (and hence its data components) has not been instantiated; on the other hand, instances of classes originally allowed access to *all* internal components except internal classes (Simula67), because a class instance contains instances of all its internal components. However, the notion of abstract data types made it desirable to restrict visibility to components of classes, but not to prevent access completely. Thus, a mechanism for selectively making components visible was needed. The simplest approach is to segregate components of a class into two categories: visible/invisible, as in Ada®

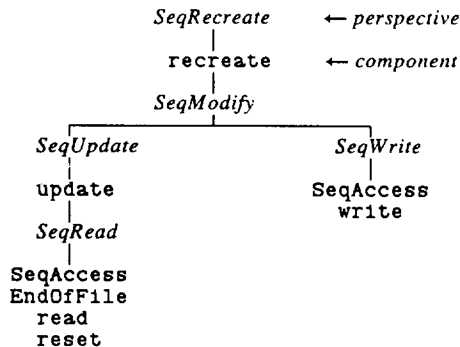
®Ada is a Registered Trademark of the U.S. Department of Defense

[ADA83], Simula (protected/hidden) and C++ (public/protected/private). The following is a brief discussion of a more complex visibility scheme (see [BUHR88] for complete details).

While components can be divided into two broad categories, those that are visible to gain access to the object's functionality, and those that are strictly internal for implementation of this functionality, there are situations where not all the visible components need to be or should be visible. In other words, the subdivision into visible and invisible is too coarse; a finer partitioning is wanted in many cases. Often there are natural groupings of visible components related to different services provided by the class. For example, a sequential file type provides a storage service and access to this storage. The visible components of a sequential file type could be grouped into the following functional groups: reading, updating, extending, modifying and recreating the information in the storage container. In general, each functional group is implemented by several components and each group contains only a subset of the visible components. Normally programming languages do not support specification of the functional groupings. However, these inherent groupings provide an extremely useful way of describing other facilities not directly related to the operation of the object, in particular security and arbitration of concurrent accessors.

## 5.1. Perspectives

A **perspective** is the name we give to a group of components that are logically related by the function that they jointly provide. This is like naming operation lists in ALPHARD [SHAW81]. Usually there are several perspectives defined for an object. These groupings are given names and often form a hierarchy of functionality. For example, the following diagram shows the perspectives for `SeqFile`, the hierarchical relationships among the perspectives, and the components that make up the perspectives.




Each perspective includes all the components below it in the hierarchy. Notice that a component may be associated with several perspectives; for example, `SeqAccess`, which is needed to access a `SeqFile` no matter what type of access is desired, appears in `SeqRead` and `SeqWrite`. The exact number of perspectives depends on the implementer's intuitive notion of what constitutes an abstraction. These perspectives are declared and their

relationship established by using a partially ordered set, as in:

```
PERSPECTIVE p : (SeqRead,
                 SeqWrite,
                 SeqUpdate > SeqRead,
                 SeqModify > SeqUpdate & SeqWrite,
                 SeqRecreate > SeqModify)
```

(Partially ordered sets are not peculiar to perspectives but can be used for other purposes in the programming language.) These names are then associated with components appropriate for the perspective.

```
OBJECT SeqFile
   PERSPECTIVE p : ...
   CLASS SeqAccess {SeqRead, SeqWrite}
      ESCAPE EndOfFile {SeqRead}
      PROC read {SeqRead} ...
      PROC reset {SeqRead} ...
      PROC write {SeqWrite} ...
      PROC update {SeqUpdate} ...
      PROC recreate {SeqRecreate} ...
```

Any component without a perspective is strictly internal to the object and is not part of the operations provided to the user. Hence, the perspectives define a user's interface to the object. Notice that there are now multiple interfaces for an object, which is different from most programming languages that support strong abstraction mechanisms, where there is a single interface per definition. The mechanism for granting programs and users visibility through perspectives is not discussed here, but it is similar to user definable capabilities as the perspectives are used both at compile time and execution time. For example, a perspective can be associated with a declaration, as in: `VAR fa : f.SeqAccess {SeqRead}`. Here the compiler can perform checks that only those components in `SeqRead` are used by `fa`. The instance can also use this information at execution time to check if a user is authorized to make the access to `f` and if so, schedule the access appropriately with other users depending on the functionality the perspective allows.

## 6. Structuring the System

The basic motivation behind much of the previous discussion is to apply the notion of strong typing across the entire system. In particular, we want to guarantee type consistency between a user's program and the system, between a user's program and conventional files (these being the sole kind of persistent data in a conventional system) and between interactive commands and the operations they invoke. (For further discussion of our motivation, see [BUHR87].) There are two important consequences of this motivation. First, all interfaces for modules that define the system, the files, and the commands must be retained and be accessible during compilation. Second, files must be objects that are completely definable in the programming language as this gives them an interface.

The interfaces for modules defined in the system and by users will be grouped appropriately into **libraries**, each containing modules that are likely related to one another in some fashion. Each user would likely have at least one library that he uses and

augments. System modules, shared by many users, would likely be grouped into several libraries. These libraries are essentially the same as those described in Ada. In conventional systems, a library would have to have a representation in the file system so that it will persist, and the compiler would have to have at least some rudimentary knowledge of the file system and library structure so that it can access a library during compilation.

We propose that the libraries be defined in a hierarchy (as they undoubtedly would be in a conventional file system) that is visible within the programming language; the modules in each library then constitute an extension of this hierarchy. As well, we propose that each module node in the hierarchy be able to be extended to define all of the submodules (procedures and other scope-defining constructs) of that module. Thus, a single hierarchy defines the libraries, the modules and the blocks of user and system programs. Each node will consist of a list of components contained within that node. Although it is not essential, our intention is that these components be stored, not in textual form, but in an internal form as the symbol table that would result from the compilation of the declarations at that node. This eliminates reparsing when that node is used in another compilation. Because declarations are not in textual form, they must be maintained and modified with a special program which we call the **program editor**. The reader should understand that, although declarations are stored in symbol table form, examples in the paper will be illustrated in textual form (i.e. what the program editor might display when browsing through the hierarchy).

Although each node must contain at least the visible components defined in it, we intend that most nodes will also contain all invisible components, too, so the node will consist of all items declared within the corresponding module or block. Each component may have its source code associated with it, comprising the body (executable code) for that component; some components, such as uninitialized variables, will not have code. The source code can be stored in textual or parse tree form as either will suit our purposes. A component that defines a new scope (such as a class) will point at the node corresponding to this new scope; this produces the hierarchy we have just described.

We call this hierarchy of library, module and block definitions the **definitional hierarchy**. The hierarchy corresponding to a single module shows the traditional structure of that module in terms of nested blocks and submodules. In fact, textual nesting of blocks could be considered as an artificial way of representing this hierarchy; except for BEGIN blocks, which are not supported in the hierarchy, the two are essentially equivalent. We intend that the various kinds of nodes (libraries, modules, etc.) can be used throughout the hierarchy, except that a library node may not contain data items as components. As has been seen in Simula and BETA, concepts like libraries and modules may be able to be implemented by a single construct in the programming language. This reduces the number of different kinds of nodes in the definitional hierarchy but does not alter the hierarchy.

A brief example of structuring a system using this approach is:

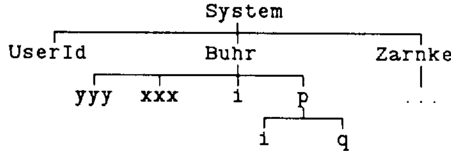
```
ENVIR System
    all the system declarations
    ENVIR UserId ...     ← type defining a user

    VAR Buhr, Zarnke : UserId    ← create two new users
```

An ENVIR is a special construct (see [BUHR87]) that is similar to a class in most respects but has the following property relevant to this discussion: instantiation of an ENVIR also creates a new node in the definitional hierarchy (as if a new block definition had been made). The program editor can then be used to add definitions to it, in the same way as for conventional blocks, as in:

```
ENVIR Buhr
   CONST yyy = ...     ┐
   CLASS xxx ...       │
   VAR i : INT         ├ added using the program editor
   PROC p(...) ...     │
      VAR i : INT      │
      PROC q(...) ...  ┘
```

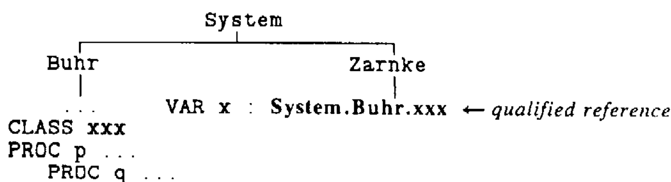
This results in the following definitional hierarchy:




## 6.1. Referencing in the Hierarchy

If the conventional visibility rules applied within the hierarchy, then a component would have to be placed so that all the blocks (or programs) that needed to refer to it were descendant blocks. This would produce a structure that was top heavy because all shared components would have to be placed at or near the root. The extreme case is when all definitions are at the same level so all may reference one another (the **flat name space** approach).

Instead, we intend that a user employ the freer organizational style used at the module level – where entities are grouped by conceptual relationship rather than actual dependence – for the definition of nodes (similar to symbolic links in a file hierarchy). This, in turn, requires a way to refer to components anywhere within the hierarchy, not just to those contained in higher level nodes. We adopt the usual qualification-name syntax for such references, where the qualifiers are the names of successively lower-level nodes in the hierarchy. If one node in the hierarchy needs to refer to a component in another node, then it uses a qualified name relative to the common ancestor of the two nodes. A simple example is sharing a type or constant contained in another user's environment, as in:




Here, the type of x is defined in a location which is not directly above it. It is also possible use the qualification in executable statements, such as the following procedure calls from Zarnke's code:

```
System.Buhr.p(...)
System.Buhr.p.q(...)  ← call of nested procedure
```

The above reference to the nested routine q will, in general, not be done, but it does illustrate the extent to which referencing in the definitional hierarchy is allowed. The reader may wonder why routine q is embedded inside of routine p when it needs to be used outside of p. From user Buhr's standpoint, q is never used outside of routine p and that is why he located it inside of p. If another user wishes to use routine q, this requires unfolding the existing definitions. However, the other user may not be able to make such a modification because he is not authorized to make changes to Buhr's environment. Thus, the other user would be forced to copy the code (if that is allowed) or write his own routine q; both these solutions are unacceptable because they violate the notion of reusability which is so important in programming-in-the-large. Allowing the other user to reference q directly solves these problems. In the future, if more users wish to access q, the code might be reorganized to reflect this new kind of usage. Our contention is that, if the choice is between allowing references that are not allowed by traditional scope rules and duplicating/copying the definition to another location where it will be accessible/visible, surely it is much more preferable to allow the reference to the original definition. Without these extensions, traditional block structuring is too restrictive and so the organization facilities that it provides are relegated to programming-in-the-small [KRIST87, p. 19]. With these extensions, a system can be structured according to usage relationships and yet references can be made from throughout the hierarchy, encouraging reusability [COX86]. This increase in visibility is the major factor in making block structure significantly more useful.

The main difficulty with our approach is to guarantee that the compiler can tell when a qualified reference can and cannot be allowed. In the nested procedures example of p and q, the reference to procedure q from outside procedure p may not be valid if q has a reference to any variable outside itself. Even if a block frame for p was created to act as the context for the execution of q, the lack of initialization of this block frame would make the execution of q unsafe. Because of the definitional hierarchy, the compiler can determine if q depends on its context and hence indicate when a reference to such a dependent entity is invalid. We call an entity that does not depend on its static context **independent**.

Our approach to structuring the system will now be contrasted with the structure of existing systems. Module interconnect languages are discussed as a mechanism to perform this structuring [DEREM76, PARNA85]. We agree with the benefits that might be achieved by such languages, but they likely require much of the interface information that is stored in our definitional hierarchy, and so it makes more sense to us to extent the definitional hierarchy to include module interconnect language information rather than creating a new language to do this.

In Smalltalk [GOLDB83], classes can be grouped into categories by the browser; however, this structure is not in any way related to the structure of the class definitions or the language itself. Essentially, all Smalltalk CLASS definitions are in a flat name space and the browser provides some artificial structure to this flat space but only for the purpose of examination by a user, not for references by objects. Browser names are not part of the Smalltalk language. This same flat name space is the way many Lisp [WILEN86] systems are implemented, too. In a large program development multi-user system, forcing all programs and all users to use the same name space is, in general, not acceptable. This problem is further exacerbated if the system also shares the same name space.

In C++ the structuring mechanism is the file system; a user employs it to store his programs. The classes of a C++ program are organized in a flat space within a source file and then a programmer can create some artificial structure for the source files through the UNIX® file system and preprocessors. However, the file system is outside the realm of the programming language (and hence no type checking) although its various levels may be very similar to the name spaces created through block structure in a program. Hence, while classes in these flat name space may have complex relations with one another, there is no requirement that this structure be reflected in any organizational scheme that is known in the programming language. Similar problems exist in Ada where the `with` statement is used to include packages but there is no connection between the name of the entity specified in the `with` statement and the actual library that contains it. Simula has the same problem when using a library class as a block prefix; how is the class prefix name found when it is not defined within the current source program?

Notice that in our system if an extended reference is made to an entity which also contains an extended reference to another entity, there are no problems. Each entity will be looked up in the context of the reference. This is not the case in UNIX/C where included source files are looked up in the current environment (unless absolute path names are specified) and not the included file's environment. This is because including is based on copying and our scheme does not copy but uses a definition intact as if it appears at its point of static definition. This is exactly how references are made to procedures, classes, etc. in the programming language. Hence, the same scheme is used to reference what are traditionally thought of as library routines and routines within a program.

®*UNIX is a Registered Trademark of AT&T Bell Laboratories.*

## 6.2. Separate Compilation

A node in the definitional hierarchy may have properties that are largely independent of its kind (i.e. module, procedure, etc.), such as whether it is separately compilable. We propose that any node in the hierarchy can be identified as being separately compilable, for example:

```
CLASS p COMPILABLE
   ...
   CLASS q COMPILABLE
      ...
      PROC r(...) COMPILABLE
      ...
   PROC s(...) INLINE
   ...
   PROC t(...)
```

Here each procedure that has a COMPILABLE clause is eligible to be compiled separately, and components with no compilation clause are compiled as part of the node that contains them. In the above example, t is compiled as part of the compilation of p (like Algol68C and Simula). INLINE has the standard meaning. There is no order on compilation; however, at least the interface for all referenced items must be defined before a node using them can be compiled. Normally, module nodes will be designated separately compilable but individual procedures might be declared as being separately compilable during their development. Separate compilation at this level, however, will exact a cost at execution time. Adding or removing a compilation clause can be done without a change to the structure of the program. Traditional systems often force a flattening of the nested scope into modules or packages to introduce a compilation unit.

As well, it is possible to use the definitional hierarchy to define precisely the location of object code generated by the compilation of each compilation unit. By introducing a definitional-time declaration of a code area, it is possible to have each compilation unit specify where its code will be placed, as in:

```
CLASS p COMPILABLE m  ← p's code is placed in code area m
   ...
   CLASS q COMPILABLE n  ← q's code is placed in code area n
      ...
   ...
```

It is our intention to be able to execute object code directly from these code containers and not necessarily to copy object code together to form the executable equivalent of a PROGRAM in Pascal (code containers are like Multics segments). Hence, we can largely do away with traditional linking (i.e. copying separate compilation units together) and prefer to dynamically execute from the original copy of the code. The user can control the number code containers that must be made accessible at execution time by controlling which compilation units go in which containers.

## 7. The Complex Situations

One anomaly that results from adopting this scheme is that two separate names might have to be specified to instantiate a class contained in another class: one for the instance that is the context for the instantiation and one for the type for the instance, for example:

```
CLASS aaa
   CLASS bbb
      VAR x : ...
      CLASS ccc
         x  ← global reference to x in bbb
      END CLASS
   END CLASS

   bbb.ccc CLASS ddd  ← prefix is a qualified reference to a type
      ...
   END CLASS

   VAR b : bbb
END CLASS

VAR a : aaa
VAR d : a.ddd
```

Unfortunately, the declaration of d is invalid because, in order to instantiate ddd, it is necessary to provide an instance of bbb because there is a dependence between the superclass for ddd, that is, ccc and class bbb. However, a reference such as a.b.ddd is also illegal because ddd is not a component of bbb. All this can be determined by the compiler. To deal with this we imagine a specification such as:

```
                 ┌─ instance context
                ┌┴┐
VAR d : a.b|aaa.ddd
                └───┬───┘
                    └─ type
```

where the instance name to the left of the | indicates the context, and the (potentially qualified) name to the right of the | indicates the type of the instance.

The inclusion of both subclassing and nesting complicates the rule for looking up names that are not completely qualified. This is because there are now two paths that can potentially be searched, for example:

```
CLASS www
   CLASS xxx
      ...
   ...
CLASS yyy
   www.xxx CLASS zzz
      x  ← reference to a variable not defined in the current scope
   ...
```

The name x can be looked up in one of two sequences of blocks: zzz, xxx, www, and possibly the context of www; or zzz, yyy, and then the context of yyy. The first search path follows the path of the subclassing chain; the second follows the hierarchy of definition. A similar problem exists with multiple inheritance, that is, which order are the ancestors looked in to resolve an unqualified name.

The search rule we have chosen is to search the superclass chain to its end but not the context containing the superclass and then proceed back to the starting point and search the definitional hierarchy. This will require that two contexts be maintained at execution time for each class instance, including each superclass instance. These contexts can all be determined at compile time. This particular rule was chosen because of the way we are structuring persistent entities in our system [BUHR87].

## 8. Implementation

We have started implementation of an integrated programming environment that is structured using extended block structure. The implementation work is in C++ using nested blocks to structure the work. This requires that we simulate nested classes by passing the containing class as an explicit context. We have considering the idea of writing a preprocessor to support nested classes in C++, but C++ is rather unstable at the moment. As well, we have used the storage management technique of multiple child classes in the implementation of dynamically sized types, such as variable-length strings (which appears on the Usenix C++ examples tape). We have compiled a library of storage managers that can be used by implementors of dynamically sized types. Having a number of storage managers allows us to try different ones with a particular type to find a good one.

## 9. Conclusion

Our idea of extended block structure is a straight-forward generalization of block structure and its usage follows naturally from the methods of programming in object oriented programming. Our scheme does not destroy the ideas of abstract data types and encapsulation. If a programmer does not want references to internal components, they can be protected. Extended block structure provides facilities that are not available in existing object oriented programming languages; these facilities can be used to solve programming-in-the-large problems, such variable-sized data structures and file definition. But most importantly, using nesting to form the definitional hierarchy and extended references to access items within the hierarchy provides a means to achieve strong typing across the entire system for a statically-typed unilingual programming environment. Other interesting problems might also lend themselves to elegant solutions using extended block structure.

## 10. Acknowledgments

We would like to thank Ron Pfeifle for helping with some of the background material, and Glen Ditchfield and Lauri Brown for reading the final draft.

## 11. References

ADA83 United States Department of Defense. "Reference Manual for the Ada Programming Language". *ANSI/MIL–STD–1815A–1983*, February 1983, Springer–Verlag, New York

ATKIN85 Atkinson, M. P., Morrison, R. "Types, Binding and Parameters in a Persistent Environment". *Persistence and Data Types Papers for the Appin Workshop*, Persistent Programming Research Report 16, University of Glasgow, Dept. of Computing Science, Scotland, August 1985, pp. 1-24

BARTO84 Bartol, W. M., et al. "Report on the LOGLAN82 Programming Language". Polska Akademia Nauk, Instytut Podstaw Informatyki, Panstwowe Wydawnictwo Naukowe, Warszawa–Lodz, 1984

BOBRO83 Bobrow, D. G., Stefik, M. "The LOOPS Manual". Xerox Corporation, 1983

BUHR86 Buhr, P. A., Zarnke, C. R. "A Design for Integration of Files into a Strongly Typed Programming Language". *Proceedings IEEE Computer Society 1986 International Conference on Computer Languages*, Miami, Florida, October 1986, pp. 190-200

BUHR87 Buhr, P. A., Zarnke, C. R. "Persistence in an Environment for a Statically-Typed Programming Language". *Persistent Object Systems: their design, implementation and use.*, Persistent Programming Research Report 44, University of Glasgow, Scotland, August 1987, pp. 317-336

BUHR88 Buhr, P. A., Zarnke, C. R. "Protection in a Multi-User Object-Oriented Environment". in preparation.

COX86 Cox, B. J. *Object Oriented Programming*, Addison–Wesley, 1986

CLARK80 Clarke, L. A., Wilden, J. C., Wolf, A. L. "Nesting in Ada Programs is for the Birds". *SIGPLAN Notices*, vol. 15, no. 11, November 1980, pp. 139-145

DEREM76 DeRemer, F., Kron, H. H. "Programming-in-the-Large Versus Programming-in-the-Small". *I.E.E.E. Transactions on Software Engineering*, vol. SE-2, no. 2, June 1976, pp. 80-86

GOLDB83 Goldberg, A., Robson, D. *Smalltalk–80: The Language and its Implementation*, Addison–Wesley, May 1983

HANSO81 Hanson, D. R. "Is Block Structure Necessary?". *Software–Practice and Experience*, vol. 11, 1981, pp. 853-866

HEERI85 Heering, J., Klint, P. "Towards Monolingual Programming Environments". *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 2, April 1985, pp. 183-213

HOUSE86 House, R. T. "Alternative Scope Rules for Block-Structure Languages". *The Computer Journal*, vol. 29, no. 3, June 1986, pp. 253-260

IBM81 IBM. *OS and DOS PL/I Language Reference Manual*, Manual GC26–3977–0, September 1981

KRIST87 Kristensen, B. B., Madsen, O. L., Moller-Pedersen, B., Nygaard, K. "The BETA Programming Language". *Research Directions in Object-Oriented Programming*, Shiver, B., Wegner, P. (eds.) MIT Press, Computer Systems Series, 1987, pp. 7-48

LAMPS77 Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., Popek, G. L. "Report on the Programming Language Euclid". *SIGPLAN Notices*, vol. 12, no. 2, February 1977, pp. 1-79

MADSE87 Madsen, O. L. "Block Structure and Object Oriented Languages". *Research Directions in Object-Oriented Programming*, Shiver, B., Wegner, P. (eds.) MIT Press, Computer Systems Series, 1987, pp. 113-128

MEYER87 Mayer, B. "Eiffel: Programming for Reusability and Extendibility". *SIGPLAN Notices*, vol. 22, no. 2, February 1987, pp. 85-94

PARNA85 Parnas, D. L., Clements, P. C., Weiss, D. M. "The Modular Structure of Complex Systems". *I.E.E.E. Transactions of Software Engineering*, vol. SE-11, no. 3, March 1985, pp. 259-266

SHAW81 Shaw, M. *ALPHARD: Form and Content*, Springer-Verlag, New York, 1981

SIMULA87 *Simula - Data Processing, Programming Languages*, Swedish Standard SS636114 SIS, Stockholm, Sweden, June 1987

STROU86 Stroustrup, B. *The C++ Programming Language*, Addison-Wesley, 1986

TEITE78 W. Teitelman, J. W. Goodwin, A. K. Hartley, et al., *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978

TEITE84 W. Teitelman, *The Cedar Programming Environment: A Midterm Report and Examination*, Xerox Palo Alto Research Center, Technical Report CSL-83-11, June 1984

TENNE82 Tennent, R. D. "Two Examples of Block Structuring". *Software–Practice and Experience*, vol. 12, 1982, pp. 385-392

WEGBR71 Wegbreit, B. "The ECL programming system". *Proceedings of AFIPS 1971 FJCC*, vol. 39, AFIPS Press, Montvale, N. J., pp. 253-262