Nesting in Ada Programs is for the Birds

Lori A. Clarke+
Jack C. Wileden
Alexander L. Wolf+

Department of Computer and Information Science
University of Massachusetts, Amherst
Amherst, Massachusetts 01003

+This work was supported in part by the Air Force Office of Scientific Research under grant AFOSR 77-3287.

Abstract

Given a data abstraction construct like the Ada package and in light of current thoughts on programming methodology, we feel that nesting is an anachronism. In this paper we propose a nest-free program style for Ada that eschews nested program units and declarations within blocks and instead heavily utilizes packages and context specifications as mechanisms for controlling visibility. We view this proposal as a first step toward the development of programming methods that exploit the novel language features available in Ada. Consideration of this proposal's ramifications for data flow, control flow, and overall program structure substantiates our contention that a tree structure is seldom a natural representation of a program and that nesting therefore generally interferes with program development and readability.

1. Introduction

The advent of Ada could signal the beginning of a new era in software development. For the first time in over a decade a new programming language, intended for production use and incorporating state-of-the-art language features, has been proposed. If accompanied by an appropriate development environment and suitable programming methods, the introduction of Ada could indeed mark a turning point in the history of software development. A major effort is now being directed toward the development of a supportive programming environment specifically tailored to Ada [2]. We contend that attention should also be directed toward the development of programming methods that exploit the novel language features available in Ada. In this paper we take a first step toward developing such methods by proposing a program style that offers guidelines concerning the way in which program units should be organized and combined in an Ada program.

Historically, the first non-trivial program organization consisted of a linear collection of independent units. FORTRAN [1] is a familiar example of a language using this program organization. A unit in FORTRAN is either a main program or a subprogram. Two FORTRAN units are completely independent of each other unless one explicitly invokes the other or they reference the same COMMON block.

In an effort to improve upon FORTRAN's weak mechanisms for data sharing and data type enforcement, ALGOL 60 [5] introduced a more elaborate program organization. An ALGOL 60 program consists of a collection of units and blocks organized as a tree structure. An ALGOL 60 unit is simply a procedure, while a block is a sequence of statements optionally preceded by a sequence of declarations. The tree structure is represented by textually enclosing, or nesting, lower level units and blocks within higher level units and blocks. While the term nesting is also commonly used to describe the embedding of statements within statements, such as nested if statements or nested loops, nesting of this sort does not concern us. Rather our concern is with the embedding of declarations that can result when units and blocks are nested, and hence we use the term nesting only in this sense in the remainder of this paper.

In ALGOL 60, nesting is used to control the scope of visibility of entities within a program. The scope of an entity's visibility is determined by the location of that entity's declaration in the program's tree structure. Access to an entity is restricted to the unit or block in which it is declared as well as any units or blocks nested therein. A declaration of an entity with a particular identifier in a given unit or block renders invisible, or hides, any declaration of entities with that same identifier appearing in ancestors of that unit or block. Therefore the visibility of a particular entity is bounded on one side by the boundaries of the unit or block in which it is declared and, potentially, on the other side by the boundaries of more deeply nested units or blocks in which its identifier is redeclared.

The entities in an Algol 60 program can be either procedures, data objects, or labels. When applied to labels or procedures, scope of visibility imposes restrictions on the possible control flow in a program. When applied to data objects, it levies restrictions on data flow.

The languages that have succeeded ALGOL 60 have incorporated more sophisticated data structures and control structures but, for the most part, have retained the ALGOL 60 concept of tree structured programs. Recently, CLU, Alphard, and other experimental languages [6] have emerged with constructs for supporting data abstraction [4]. Ada has adopted many of the data and control structures pioneered by ALGOL's successors and offers the package construct for data abstraction, but Ada has also retained ALGOL's program structure of nested program units and blocks. In Ada, a program unit is a subprogram, a package or a task*, while the definition (although not the syntax) of a block is the same as in ALGOL 60. We argue that given a data abstraction construct such as the Ada package and in light of current thoughts on programming methodology, nesting is an anachronism.

As an alternative to nesting, we propose a nest-free program organization, which is an essentially flat organization coupled with constructs for explicitly associating identifiers of program entities with the particular units in which those entities are accessed. Our objections to a tree structure for programs are based upon the generally unnatural organization that it produces and its inadequacy for precisely capturing a program's intended data references and control flow. Thus we advocate a program style for Ada that eschews nested program units and declarations within blocks and instead heavily exploits packages and context specifications as mechanisms for controlling visibility.

The remainder of this paper elaborates our arguments against a nested program structure and further details the nest-free Ada program style.

2. Arguments Against Nesting

In Ada, and in its predecessors, nesting has primarily been employed to govern control flow and data flow within programs. In this section we demonstrate the inadequacy of nesting for both of these uses and discuss how nesting interferes with program development and readability. Throughout this paper we use the program organization described in Figure 1 as the basis for examples illustrating our objections to nested program structure. Although these examples are all stated in terms of nesting within procedures, they could also have been phrased in terms of nesting within packages or blocks. Only where the invocation of a procedure is explicitly mentioned is the use of procedures as the nested objects significant.

procedure	declares	references	invokes
A	X,Y...	X,Y...	B,C
B	...	...	D,E
C	...	...	F,G
D	...	Y...	
E	Z...	Z...	
F	...	...	
G	...	Y...	

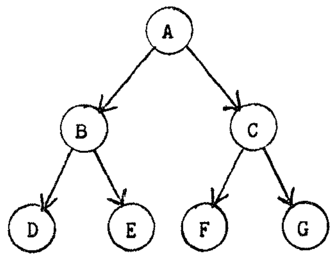
Example Program Organization

Figure 1

*To simplify the presentation, we restrict our discussion to subprograms and packages.

2.1 Control Flow Arguments

In Ada, as well as other languages that employ a tree-structured program organization, nesting affects the flow of control by restricting access to program units. Ada has essentially adopted the ALGOL 60 rules for controlling the invocation of subprograms. These rules are based upon each subprogram's location in the tree structure. A given subprogram within this structure may invoke its direct descendants as well as invoke any of its ancestors and any siblings, either its own or its ancestors', which textually precede it in the program listing. While nesting protects a subprogram from being invoked by any subprograms above it in the tree structure other than its parent, the subprogram can be accessed from any of its own descendants or those of its younger siblings. Thus, while it may appear that nesting would precisely capture a calling structure that is organized as a tree, this is not the case. The program invocations specified in Figure 1 are presented in Figure 2 in the form of a call graph, a graphical representation of the subprogram invocations found within a program. Since this call graph is a tree, it can also serve as the



Call Graph of the Program Organization Specified in Figure 1

Figure 2

program structure tree, which depicts the nested organization of a program. Figure 3 shows the textual representation of the program structure given by this tree. The program structure tree, and thus its associated textual representation, allows for the possibility of numerous other patterns of invocations. A potential call graph shows all possible subprogram invocations permitted by a particular program structure tree. The potential call graph for our example is shown in Figure 4*. As illustrated by the potential call graph, the textual representation in Figure 3 realizes not only the desired calling pattern of the example program, but many others as well. In particular, any program whose calling pattern is a subgraph of the potential call graph in Figure 4

*For simplicity, cycles of length one, i.e., self recursive procedure calls, have not been shown.

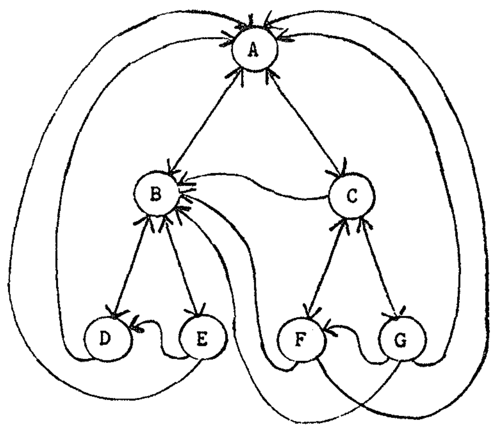
```
procedure A is
   X,Y : INTEGER;
   ...
   procedure B is
      ...
      procedure D is
         ...
      begin
         -- sequence of statements of D
         -- (referencing Y)
      end D;
      ...
      procedure E is
         Z : INTEGER;
         ...
      begin
         -- sequence of statements of E
         -- (referencing Z)
      end E;
      ...
   begin
      -- sequence of statements of B
      -- (invoking D and E)
   end B;
   ...
   procedure C is
      ...
      procedure F is
         ...
      begin
         -- sequence of statements of F
      end F;
      ...
      procedure G is
         ...
      begin
         -- sequence of statements of G
         -- (referencing Y)
      end G;
      ...
   begin
      -- sequence of statements of C
      -- (invoking F and G)
   end C;
   ...
begin
   -- sequence of statements of A
   -- (referencing X and Y, invoking B and C)
end A;
```

A Textual Representation of the Program Organization Specified in Figure 1

Figure 3

may be textually represented by the organization of Figure 3. In general, a given control flow organization may be represented by several different nested structures and a given nested structure may permit numerous distinct calling patterns. Hence, at best, nesting offers an imprecise representation of the intended calling structure of a program.

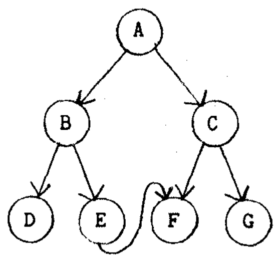
The example discussed above illustrates the limitations of nesting as a means for describing an intended calling structure that is organized as a tree. Nesting is even less suitable for representing a more general calling structure. For instance, suppose that the program organization



Potential Call Graph of the Program in Figure 3

Figure 4

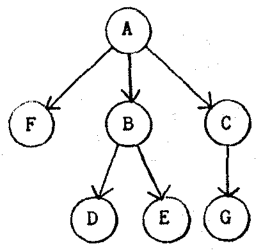
shown in Figure 1 is modified so that procedure E invokes procedure F. The resulting call graph is presented in Figure 5. Since this call graph is not a tree, it cannot be used as a program structure tree. Therefore, constructing a nested



Call Graph of the Modified Figure 1 Program Organization

Figure 5

program to realize the calling structure requires the additional effort of finding a suitable program structure tree. In general there are several such trees. One possible program structure tree that supports the pattern of invocations shown in Figure 5 is given in Figure 6. The potential call graph



A Program Structure Tree for the Call Graph of Figure 5

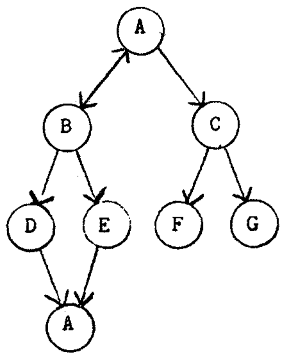
Figure 6

derived from this program structure tree does indeed subsume the call graph of Figure 5. In general, however, the translation from an intended calling structure to a suitable program structure tree is not a particularly natural operation. In practice, programmers usually discover a suitable program structure by moving procedures invoked by many other procedures to successively higher nesting levels in their programs. For example, the textual representation given in Figure 3 can be modified to support the call graph of Figure 5 by moving procedure F up to the point just ahead of procedure B, which results in the program structure tree of Figure 6. Thus, in general, one consequence of nesting is that large programs frequently begin with a long list of low level utility procedures.

The problem of finding a suitable program structure tree is even more complicated when two program entities have the same identifier, since hiding can then lead to unexpected results. For instance, suppose that the program organization of Figure 1 is modified by adding A to the list of procedures invoked by B. Suppose further that, at some later date, a programmer identifies a segment of code common to procedures D and E and decides to make the common segment into a new procedure, invoked by both D and E. Should the programmer choose to give that new procedure the identifier A, perhaps being unaware that that identifier has already been used, the resulting call graph would be the one shown in Figure 7. Applying the usual technique of moving the shared procedure to a higher nesting level would result in the program structure tree shown in Figure 8. This tree, and its corresponding textual representation, seemingly permits all the intended invocations indicated in the call graph. In this program structure, however, the old procedure A is no longer accessible to procedure B, since it is hidden by the new procedure A. As a result, B's invocation of A will now be invoking a different procedure A, with potentially disasterous results.

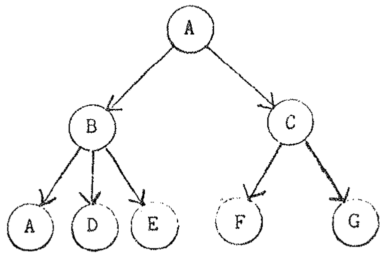
2.2 Data Flow Arguments

Nesting affects data flow in Ada programs in essentially the same way that it affects control flow, by restricting access to program entities. In the case of data flow, the entities in question include variables, constants, and types. To



Call Graph of Another Modified Figure 1 Program Organization

Figure 7



A Program Structure Tree for the Call Graph of Figure 7

Figure 8

simplify the presentation we restrict the ensuing discussion to variables, although similar arguments can be made for constants and types.

We contend that, as was true for control flow, a nested program structure provides an imprecise representation of intended data flow. Specifically, nesting permits unintended access to variables, and hiding can lead to unanticipated results. Both of these difficulties are illustrated by the example program organization of Figure 1 and the textual representation of Figure 3. According to Figure 1, variable X is intended to be referenced only by procedure A, while variable Y is to be referenced by procedures A, D, and G. However, being declared in procedure A, as they must be to permit the intended references, these variables can, in fact, be referenced from any of the seven procedures included in the program. Moreover, should a programmer subsequently insert a declaration for a new variable with identifier Y into procedure B, this new Y would hide the one declared in A from procedure D. D's references to Y would then affect the new Y rather than the old Y, with unpredictable consequences.

Furthermore, as was the case for control flow, nesting often leads to an unnatural program organization when variables are shared. Again referring to Figures 1 and 3, suppose that variable Z, declared and referenced in procedure E, is to be shared with procedure F. In some instances, a modification of this kind can be accomplished by moving one of the procedures involved. In this example, making procedure E the outermost procedure of the program will both preserve the intended calling structure and permit E and F to reference Z. As was previously pointed out, however, moving a procedure can often introduce further problems and typically results in an unnatural program organization. Another alternative for permitting E and F to share Z is to move the declaration of Z up to the point in procedure A where X and Y are declared. The principal disadvantages of this alternative are that Z is now declared in a procedure whose body contains no reference to Z and that the declaration is now far from the actual points of reference. Moreover, Z can now be accessed by procedures other than E and F, and a declaration of another variable with identifier Z in procedure B or C would hide the shared variable from E or F, respectively.

2.3 Program Development and Readability Arguments

Our primary motivation for proposing a program style for Ada is to foster a more natural program structure that facilitates development and enhances readability. A major impediment to program development and readability is the artificial ordering of units imposed by nesting. As noted above, maintaining a nested program structure during the development phase often requires that program units be repeatedly shuffled from location to location. This movement of code disrupts logical program development and, as pointed out in the above examples, may introduce subtle program errors. Furthermore, nesting leads to a program text in which the bodies of lower level units appear before the bodies of higher level units and hence prior to the context in which they are used. Moreover, in a nested program, variable declarations may be encountered well before their use due to intervening unit declarations. Most notably, the main procedure's variable declarations typically appear at the very beginning of the program text while its body appears at the very end. In addition, the use of blocks to declare variables at intermediate locations within a program unit is generally considered a poor programming practice that hinders readability. Confining all declarations of variables to the declarative part of a program unit precisely establishes the entities in use within the unit and provides a common point of reference for the unit's name space. In sum, the program structure resulting from nesting interferes with the logical exposition of the program text, thereby hindering both development and maintenance.

Ada's designers evidently recognized the limitations that nesting imposes on program development and readability, and attempted to overcome them by proposing the subunit facility. This facility permits the body of a program unit embedded in the declarative part of another unit to be removed from that declarative part and made textually distinct from the enclosing unit. Instead of the entire unit, only a stub need appear in the declarative part of the enclosing unit. The subunit facility appears to overcome the major drawbacks of a nested program structure. In particular, the text of programs developed in a top-down fashion can be organized in a top-to-bottom manner, as illustrated in the Ada reference manual [3, p.10-7], with only the stubs of referenced subunits appearing prior to the actual reference. However, the subunit facility in fact preserves nesting and hence some of its associated shortcomings with respect to readability. According to the Ada language definition, the textually separate subunit body is still considered to be logically located at the point where the stub appears, that is, nested within the declarative part of another unit. It is the location of this stub that determines the context, i.e., the visibility of other program units and data objects, within which this subunit is to be understood. Since the subunit is textually distinct from the stub whose location determines its logical context, this can make both writing and understanding the subunit extremely difficult.

3. A Nest-Free Program Style For Ada

Having discussed the drawbacks of a nested program structure, we now direct our attention to the manner in which an Ada program would be constructed using our nest-free program style. In this section we detail the overall program structure implied by our proposal, justify that structure in terms of programming methodology considerations, and discuss how our proposal fits within the framework of the Ada language design.

The nest-free Ada program style would generate programs that are linear collections of program units (i.e., subprograms, packages and tasks). No nesting of program units would be permitted and blocks would not be allowed to have declarative parts. Specifically, subprograms and tasks would not contain the declarations (or bodies) of other subprograms, tasks or packages and packages would not contain the declarations (or bodies) of other packages. Packages may contain subprograms and tasks, but this is merely a syntactic grouping to accomodate data encapsulation and information hiding.

In place of nested program units and embedded declarations, the nest-free program style heavily exploits the package and context specification constructs as the foundations for program organization. Besides supporting data encapsulation in a fairly natural way, packages can be used to describe variable visibility and intended control flow much more precisely than can be done using nesting. The context specification construct, used in conjunction with compilation units which are packages or subprograms, provides a means for explicitly indicating the relationships among program entities. This approach is not only more explicit but also more general and more flexible than a nested program organization, which relies upon the textual location of program units to implicitly define a tree structure governing control flow and variable visibility. Applying our approach to the program organization depicted in Figure 1 would result in a textual representation like that shown in Figure 9.

The Ada package and context specification constructs can be used to describe a program's desired control flow much more precisely than can be done using nesting. In a nest-free Ada program, a program unit explicitly indicates which subprograms it may directly access by using a context specification that lists the compilation units containing those subprograms. This results in a much closer correspondence between a program's potential call graph and its intended calling structure than can be obtained using nesting. For instance, the potential call graph of the program shown in Figure 9 is identical to its call graph, which appears in Figure 2. Furthermore, note that revisions to a program that result in additional sharing of subprograms only require modifications to the context specifications of those program units newly accessing the shared subprograms. Although not illustrated by this example, there are several other ways in which the Ada package construct can be used to improve the description of intended control flow. Specifically, the logical relationships among a set of subprograms, often based on their common use of some data objects, can be expressed by grouping them into a package. Moreover, the subprograms in a package are

```
package Y_PACK is
   Y : INTEGER;
   ...
end Y_PACK;
...
... -- subprogram specifications for
... -- subprograms B, C, D, E, F, and G
...
with Y_PACK, B, C; use Y_PACK; procedure A is
   X : INTEGER;
   ...
begin
   -- sequence of statements of A
   -- (referencing X and Y, invoking B and C)
end A;
...
with D, E; procedure B is
   ...
begin
   -- sequence of statements of B
   -- (invoking D and E)
end B;
...
with F, G; procedure C is
   ...
begin
   -- sequence of statements of C
   -- (invoking F and G)
end C;
...
with Y_PACK; use Y_PACK; procedure D is
   ...
begin
   -- sequence of statements of D
   -- (referencing Y)
end D;
...
procedure E is
   Z : INTEGER;
   ...
begin
   -- sequence of statements of E
   -- (referencing Z)
end E;
...
procedure F is
   ...
begin
   -- sequence of statements of F
end F;
...
with Y_PACK; use Y_PACK; procedure G is
   ...
begin
   -- sequence of statements of G
   -- (referencing Y)
end  G;
```

A Better Textual Representation of the Program Organization Specified in Figure 1

Figure 9

explicitly declared to be visible or hidden to program units outside the package. None of these control flow relationships can be satisfactorily described by nesting.

Packages and context specifications also allow for more precise control of variable visibility than can be obtained using a nested program organization. Local variables, which are used within only a single program unit, can simply be declared within that unit. Variables that are to be shared among several program units can be placed in the visible parts of packages and made directly accessible to the program units sharing them through the use of context specifications. Figure 9 illustrates how a package, in conjunction with context specifications, can be used to explictly describe the sharing of variables -- in this case variable Y. Note that revisions to a program that result in the sharing of a previously local variable only require placing the shared variable into a package and making appropriate modifications to the context specifications of those program units accessing the newly shared variable. Thus, in a nest-free program organization, no unit enclosing all the program units that are to access some set of variables need be found or created and program units need not inherit access to variables that they do not use simply due to their position in the nesting structure.

We contend that a nest-free program organization also improves the readability of Ada programs and facilitates program development. Using packages and context specifications to express a program unit's relationships, both to other program units and to data objects, results in a program organization in which program units can be arranged in any desired order*. Programmers who employ a nest-free program organization are freed from any necessity of fitting their programs into a tree structure and can hence more easily pursue a methodical and structured approach to programming. In particular, the text of a program developed using a top-down approach can have a top-to-bottom organization, with higher level units preceding lower level units in the program text. Allowing programmers to organize their programs' units into an ordering better suited to their style of programming enhances their programs' readability and thus aids those who must read, understand, and perhaps modify their programs.

We recognize that our nest-free program style for Ada does not provide for an absolutely precise description of control flow and data flow. In particular, it does not offer a general facility for selectively denying access to program units or data objects. It is easy to see, however, that nesting also fails in this regard. Indeed, completely general control of accessibility can be obtained only through additional language constructs (e.g., import/export lists) or through mechanisms in a suitable programming environment. In the absence of such mechanisms and based on the current design of Ada, we believe our nest-free program style provides a degree of control over accessibility that is superior to that provided by nesting. Furthermore, it provides a more readable and maintainable program structure, which is more easily adapted to top-down methods of program development.

---

*Of course, in cases where a program unit's body does not textually precede the first reference to that unit, the Ada rules governing order of compilation require that a compilation unit consisting of a specification of that unit be placed ahead of the reference, as illustrated in Figure 9. While we do not consider this a significant limitation on program organization, removing this unnecessary restriction would be more in keeping with the spirit of our proposal.

4. Conclusion

We have proposed a style for programming in Ada that precludes the use of nesting and thereby avoids nesting's negative impact on program organization and readability. The nest-free program style has been justified by detailing a number of nesting's shortcomings and by showing how this program style overcomes them without requiring a single change to the Ada language. Although we believe that a nest-free program organization would benefit any programming methodology, it is especially conducive to top-down programming since it allows the textual ordering of the units in a program developed in a top-down fashion to more closely correspond to the order in which they were generated. Thus, as contrasted with the convoluted organization imposed by nested program structures, the nest-free style allows a programmer to directly record a program's development history and logical structure within the organization of the text itself.

References

[1] ANSI X3.9 - 1966 (USA Standard FORTRAN).

[2] Buxton, J.N., Requirements for Ada Programming Support Environments, ("Stoneman"), United States Department of Defense, February 1980.

[3] Ichbiah, J.D., et al., Reference Manual for the Ada Programming Language, United States Department of Defense, July 1980.

[4] Liskov, B.H. and Zilles, S.N., "Specification Techniques for Data Abstractions", IEEE Transactions on Software Engineering, SE-1, 1 (March 1975), pp. 7-18.

[5] Naur, P. (ed.), "Revised Report on the Algorithmic Language ALGOL 60", Communications of the ACM, 6, 1 (January 1963), pp. 1-17.

[6] Wortman, D.B. (ed.), "Proceedings of an ACM Conference on Language Design for Reliable Software", SIGPLAN Notices, 12, 3 (March 1977).