J3/02-277r1

WORKING DRAFT
ISO IEC TECHNICAL REPORT 19767

ISO/IEC JTC1 WG5 PROJECT 1.22.02.01.01.01

Enhanced Module Facilities

in

Fortran

An extension to IS 1539-1

7 November 2002

Contents

Foreword

[General part to be provided by ISO CS]

This technical report specifies an extension to the module program unit facilities of the programming language Fortran. Fortran is specified by the international standard ISO/IEC 1539-1. This document has been prepared by ISO/IEC JTC1/SC22/WG5, the technical working group for the Fortran language.

It is the intention of ISO/IEC JTC1/SC22/WG5 that the semantics and syntax specified by this technical report be included in the next revision of the Fortran standard (ISO/IEC 1539-1) without change unless experience in the implementation and use of this feature identifies errors that need to be corrected, or changes are needed to achieve proper integration, in which case every reasonable effort will be made to minimize the impact of such changes on existing implementations.

0 Introduction

The module system of Fortran, as standardized by ISO/IEC 1539-1, while adequate for programs of modest size, has shortcomings that become evident when used for large programs, or programs having large modules. The primary cause of these shortcomings is that modules are monolithic.

This technical report extends the module facility of Fortran so that program developers can optionally encapsulate the implementation details of module procedures in **submodules** that are separate from but dependent on the module in which the interfaces of their procedures are defined. If a module or submodule has submodules, it is the **parent** of those submodules.

The facility specified by this technical report is compatible to the module facility of Fortran as standardized by ISO/IEC 1539-1.

0.1 Shortcomings of Fortran's module system

The shortcomings of the module system of Fortran, as specified by ISO/IEC 1539-1, and solutions offered by this technical report, are as follows.

0.1.1 Decomposing large and interconnected facilities

If an intellectual concept is large and internally interconnected, it requires a large module to implement it. Decomposing such a concept into components of tractable size using modules as specified by ISO/IEC 1539-1 may require one to convert private data to public data.

Using facilities specified in this technical report, such a concept can be decomposed into modules and submodules of tractable size, without exposing private entities to uncontrolled use.

Decomposing a complicated intellectual concept may furthermore require circularly dependent modules, but this is prohibited by ISO/IEC 1539-1. It is frequently the case, however, that the dependence is between the implementation of some parts of the concept and the interface of other parts. Because the module facility defined by ISO/IEC 1539-1 does not distinguish between the implementation and interface, this distinction cannot be exploited to break the circular dependence. Therefore, modules that implement large intellectual concepts tend to become large, and therefore expensive to maintain reliably.

Using facilities specified in this technical report, complicated concepts can be implemented in submodules that access modules, rather than modules that access modules, thus reducing the possibility for circular dependence between modules.

0.1.2 Avoiding recompilation cascades

Once the design of a program is stable, most changes in modules occur in the implementation of those modules – in the procedures that implement the behavior of the modules and the private data they retain and share – not in the interfaces of the procedures of the modules, nor in the specification of publicly accessible types or data entities. Changes in the implementation of a module have no effect on the translation of other program units that access the changed module. The existing module facility, however, draws no structural distinction between interface and implementation. Therefore, if one changes any part of a module, most language translation systems have no alternative but to conclude that a change might have occurred that could affect other modules that access the changed module. This effect cascades into modules that access modules that access the changed module, and so on. This can cause a substantial expense to retranslate and recertify a large program. Recertification can be severals orders of magnitude more costly than retranslation.

Using facilities specified in this technical report, implementation details of a module can be encapsulated in submodules. Submodules are not accessible by use association, and they depend on their parent module, not vice-versa. Therefore, submodules can be changed without implying that other modules must be translated differently.

If a module is used only in the implementation of a second module, a third module accesses the second, and one changes the interface of the first module, utilities that examine the dates of files have no alternative but to conclude that a change may have occurred that could affect the translation of the third module.

Modules can be decomposed using facilities specified in this technical report so that a change in the interface of a module that is used only in a submodule has no effect on the parent of that submodule, and therefore no effect on the translation of other modules that use the second module. Thus, compilation cascades caused by changes of interface can be shortened.

0.1.3 Packaging proprietary software

If a module as specified by international standard ISO/IEC 1539-1 is used to package proprietary software, the source text of the module cannot be published as authoritative documentation of the interface of the module, without either exposing trade secrets, or requiring the expense of separating the implementation from the interface every time a revision is published.

Using facilities specified in this technical report, one can easily publish the source text of the module as authoritative documentation of its interface, while witholding publication of the source text of the submodules that contain the implementation details, and the trade secrets embodied within them.

0.1.4 Easier library creation

Most Fortran translator systems produce a single file of computer instructions and data, called an *object file*, for each module. This is easier than producing an object file for the specification part and one for each module procedure. It is also convenient, and conserves space and time, when a program uses all or most of the procedures in each module. It is inconvenient, and results in a larger program, when only a few of the procedures in a general purpose module are needed in a particular program.

Modules can be decomposed using facilities specified in this technical report so that is easier for each program unit's author to control how module procedures are allocated among object files.

0.2 Disadvantage of using this facility

Translator systems will find it more difficult to carry out inter-procedural optimizations if the program uses the facility specified in this technical report. When translator systems become able to do inter-procedural optimization in the presence of this facility, it is likely that requesting inter-procedural optimization will cause compilation cascades in the first situation mentioned in section 0.1.2, even if this facility is used. Although one advantage of this facility could perhaps be nullified in the case when users request inter-procedural optimization, it would remain if users do not request inter-procedural optimization, and the other advantages remain in any case.

Information technology – Programming Languages – Fortran

Technical Report: Enhanced Module Facilities

1 General

1.1 Scope

This technical report specifies an extension to the module facilities of the programming language Fortran. The current Fortran language is specified by the international standard ISO/IEC 1539-1 : Fortran. The extension allows program authors to develop the implementation details of concepts in new program units, called **submodules**, that cannot be accessed directly by use association. In order to support submodules, the module facility of international standard ISO/IEC 1539-1 is changed by this technical report in such a way as to be upwardly compatible with the module facility specified by international standard ISO/IEC 1539-1.

Clause 2 of this technical report contains a general and informal but precise description of the extended functionalities. Clause 3 contains detailed editorial changes that would implement the revised language specification if they were applied to the current international standard.

1.2 Normative References

The following standards contain provisions that, through reference in this text, constitute provisions of this technical report. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. Parties to agreements based on this technical report are, however, encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referenced applies. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 1539-1 : *Information technology - Programming Languages - Fortran*

2 Requirements

The following subclauses contain a general description of the extensions to the syntax and semantics of the current Fortran programming language to provide facilities for submodules, and to separate subprograms into interface and implementation parts.

2.1 Summary

This technical report defines a new entity and modifications of two existing entities.

The new entity is a program unit, the *submodule*. As its name implies, a submodule is logically part of a module, and it depends on that module. A new variety of interface bodies, a *module interface body*, and a new variety of procedure, a *separate module procedure*, are described below.

By putting a module interface body in a module and its corresponding separate module procedure in a submodule, program units that access the module interface body by use association do not depend on the procedure's body. Rather, the procedure's body depends on its interface body.

2.2 Submodules

A **submodule** is a program unit that is dependent on and subsidiary to a module or another submodule. A module or submodule may have several subsidiary submodules. If it has subsidiary submodules, it is the **parent** of those subsidiary submodules, and each of those submodules is a **child** of its parent. A submodule accesses its parent by host association.

An **ancestor** of a submodule is that submodule, or an ancestor of its parent. A **descendant** of a module or submodule is that program unit, or a descendant of a child of that program unit.

A submodule is introduced by a statement of the form `SUBMODULE (` *parent-name* `)` *submodule-name*, and terminated by a statement of the form `END SUBMODULE` *submodule-name*. The *parent-name* is the name of the parent module or submodule.

Identifiers in a submodule are effectively PRIVATE, except for the names of separate module procedures that correspond to public module interface bodies in the parent module. It is not possible to access entities declared in the specification part of a submodule by use association because a USE statement is required to specify a module, not a submodule. ISO/IEC 1539-1 permits PRIVATE and PUBLIC declarations only in a module, and this technical report does not propose to change that specification.

In all other respects, a submodule is identical to a module.

2.3 Separate module procedure and its corresponding module interface body

A **module interface body** is different from an interface body defined by ISO/IEC 1539-1 in three respects. First, it has a MODULE prefix in the subroutine statement or function statement that introduces the interface body. Second, in addition to specifying a procedure's characteristics and dummy argument names, a module interface body specifies that its corresponding procedure body is in a descendant of the module or submodule in which it appears. Third, unlike an ordinary interface body, it accesses the module or submodule in which it is declared by host association.

If a module procedure has the same name as a module interface body declared in an ancestor module or submodule, the procedure is a **separate module procedure** that corresponds to the module interface body. Its characteristics and dummy argument names are declared by its corresponding interface body. The procedure is accessible if and only if its interface body is accessible.

Some or all of the characteristics and dummy argument names may be redeclared in the module subprogram that defines the separate module procedure. If any dummy arguments are redeclared, all shall be redeclared, and shall have the same names and characteristics as in the interface body. Any other characteristics of the module procedure that are declared in the module subprogram shall be the same as those declared in its interface body.

If the procedure is a function, the result variable name is determined by the declaration of the module subprogram, not by the module interface body. If the module interface body declares a result variable name different from the function name, that declaration is ignored, except for its use in specifying the result variable characteristics.

2.4 Examples of modules with submodules

The example module POINTS below declares a type POINT and an interface body for a module function POINT_DIST. Because the interface body includes the MODULE prefix, it accesses the scoping unit of the module by host association, without needing an IMPORT statement. The declaration of the result variable name DISTANCE serves only as a vehicle to declare the result characteristics; the name is otherwise ignored.

```
MODULE POINTS
  TYPE :: POINT
    REAL :: X, Y
  END TYPE POINT

  INTERFACE
    MODULE FUNCTION POINT_DIST ( A, B ) RESULT ( DISTANCE )
      TYPE(POINT), INTENT(IN) :: A, B ! Accessed by host association
      REAL :: DISTANCE
    END FUNCTION POINT_DIST
  END INTERFACE
END MODULE POINTS
```

The example submodule POINTS_A below is a submodule of the POINTS module. The scope of the type name POINT extends into the submodule; it cannot be redefined in the submodule. The characteristics of the function POINT_DIST can be redeclared in the module function body, or taken from the module interface body in the POINTS module. The fact that POINT_DIST is accessible by use association results from the fact that there is a module interface body of the same name in the ancestor module.

```
SUBMODULE ( POINTS ) POINTS_A
  CONTAINS
    REAL FUNCTION POINT_DIST ( P, Q ) RESULT ( HOW_FAR )
      TYPE(POINT), INTENT(IN) :: P, Q
      HOW_FAR = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
    END FUNCTION POINT_DIST
END SUBMODULE POINTS_A
```

An alternative declaration of the example submodule POINTS_A shows that it is not necessary to redeclare the characteristics of the module procedure POINT_DIST. The result variable name is POINT_DIST, even though the module interface body specifies a different result variable name.

```
SUBMODULE ( POINTS ) POINTS_A
  CONTAINS
    FUNCTION POINT_DIST
      TYPE(POINT), INTENT(IN) :: P, Q
      POINT_DIST = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
    END FUNCTION POINT_DIST
END SUBMODULE POINTS_A
```

2.5 Relation between modules and submodules

Public entities of a module, including module interface bodies, can be accessed by use association. The only entities of submodules that are accessible by use association are separate module procedures for which there is a corresponding publicly accessible module interface body.

A submodule accesses the scoping unit of its parent module or submodule by host association.

3 Required editorial changes to ISO/IEC 1539-1

The following editorial changes, if implemented, would provide the facilities described in foregoing sections of this report. Descriptions of how and where to place the new material are enclosed between square brackets.

[After the third right-hand-side of syntax rule R202, at 9:12+, insert:]

or *submodule*

[After syntax rule R1104, at 9:34+, add the following syntax rule. This is a quotation of the "real" syntax rule in subclause 11.2.3.]

```
submodule                    is  submodule-stmt
                                     [ specification-part ]
                                     [ module-subprogram-part ]
                                     end-submodule-stmt
```

[In the second line of the first paragraph of subclause 2.2, at 11:42, insert ", a submodule" after "module".]

[In the fourth line of the first paragraph of subclause 2.2, at 11:44, insert a new sentence:]

A submodule is an extension of a module; it may contain the definitions of procedures declared in a module or another submodule.

[In the sixth line of the first paragraph of subclause 2.2, at 11:46, insert ", a submodule" after "module".]

[In the penultimate line of the first paragraph of subclause 2.2, at 11:48, insert "or submodule" after "module".]

[Replace the second sentence of 2.2.3.2, at 12:27-29, by the following sentence.]

A module procedure may be invoked from within any scoping unit that accesses its declaration (12.3.2.1) or definition (12.5).

[Insert the following note at the end of 2.2.3.2, at 12:30+.]

NOTE 2.2$\frac{1}{2}$

The scoping unit of a submodule accesses the scoping unit of its parent module or submodule by host association.

[Insert a new subclause at 13:17+:]

2.2.5 Submodule

A **submodule** contains definitions (12.5) for procedures whose interfaces are declared (12.3.2.1) in its parent module or submodule. It may also contain declarations and definitions of entities that are accessible to and used by descendant submodules. An entity declared in a submodule is not accessible by use association, but a procedure that is declared in a module and defined in one of that module's submodules is accessible by use association.

[In the second line of the first row of Table 2.1 on page 14, insert ", SUBMODULE" after "MODULE".]

[On page 14, change the heading of the third column of Table 2.2 from "Module" to "Module or Submodule".]

[In the second footnote to Table 2.2 on page 14, insert "or submodule" after "module" and change "the module" to "it".]

[In the last line of 2.3.3, at 15:3, insert ", *end-submodule-stmt*" after "*end-module-stmt*".]

[In the first line of the second paragraph of 2.4.3.1.1, at 17:4, insert ", submodule" after "module".]

[At the end of 3.3.1, immediately before 3.3.1.1 on page 28, add "END SUBMODULE" to the list of adjacent keywords where blanks are optional.]

[In the third line of the first paragraph of 4.5.1.8, at 50:22, replace "itself" by "and all of its descendant submodules".]

[In the last line of the second paragraph of 4.5.1.8, at 50:28, after "definition" add "and all of its descendant submodules".]

[In the last line of the fourth paragraph of 4.5.1.8, at 51:6, after "definition" add "and all of its descendant submodules".]

[In the last line of the first paragraph after Note 4.34, at 51:8, after "definition" add "and all of its descendant submodules".]

[In the last line of Note 4.37 on page 51, after "module" add "and all of its descendant submodules".]

[In the last line of Note 4.38 on page 51, after "defined" add ", and all of its descendant submodules".]

[In the last line of Note 4.39 on page 52, after "definition" add "and all of its descendant submodules".]

[In the third line of the second paragraph of 4.5.10.1, at 60:19, insert "or submodule" after "module". In the third and fourth line, replace "referencng the module" by "that has access to that program unit".]

[In the first line of the second paragraph of Note 4.58, on page 61, insert "or submodule" after "module".]

[In constraint C531, at 69:33, insert "or submodule" after "module".]

[In the first line of the second paragraph of 5.1.2.12, at 81:26, insert ", or any of its descendant submodules" after "attribute".]

[In the first line of the second paragraph of 5.1.2.13, at 82:9, insert "or any of its descendant submodules" after "module".]

[In constraint C558, at 85:10, insert "or submodule" after "module".]

[After the second paragraph after constraint C580, at 91:7+, insert the following note.]

[In the third line of the penultimate paragraph of 6.3.1.1, at 111:15, replace "or a subobject thereof" by "or submodule, or a subobject thereof,".]

[In the first line of the first paragraph after Note 6.22, at 113:9, insert "or submodule" after "module".]

[In the fourth item in the list in 6.3.3.2, at 115:10, insert "or submodule" after the first "module".]

[In the second line of the first paragraph of Section 11, at 245:3, insert ", a submodule" after "module".]

[In the first line of the second paragraph of Section 11, at 245:4, insert ", submodules" after "modules".]

[In constraint C1105, at 246:20, insert "or submodule" after "module".]

[In constraint C1106, at 246:22, insert "or submodule" after "module".]

[In constraint C1107, at 246:24, insert "or submodule" after "module".]

[Within the first paragraph of 11.2.1, at its end at 247:4, insert the following sentence:]

A submodule shall not reference its ancestor module by use association, either directly or indirectly.

[Then insert the following note:]

NOTE 11.6$\frac{1}{2}$

It is possible for submodules of different modules to access each others' ancestor modules.

[After constraint C1109, at 247:36+, insert an additional constraint:]

Constraint: If the USE statement appears within a submodule, *module-name* shall not be the name of the ancestor module of the submodule.

[Insert a new subclause immediately before 11.3, at 249:6-:]

11.2.3 Submodules

A **submodule** is a program unit that depends on a module or another submodule. It may provide definitions for module procedures that are declared in or accessible by host association in the module or submodule on which it depends, and declarations and definitions of other entities that are accessible by host association in submodules subsidiary to it.

```
submodule                    is  submodule-stmt
                                     [ specification-part ]
                                     [ module-subprogram-part ]
                                     end-submodule-stmt

submodule-stmt               is  SUBMODULE ( parent-name ) submodule-name

end-submodule-stmt           is  END [ SUBMODULE [ submodule-name ] ]
```

Constraint: The *parent-name* shall be the name of a submodule or a nonintrinsic module.

Constraint: The *submodule-name* shall not be the same as *parent-name*.

Constraint: If a *submodule-name* is specified in the *end-submodule-stmt*, it shall be identical to the *submodule-name* specified in the *submodule-stmt*.

The program unit on which a submodule depends is its **parent** module or submodule; its parent is specified by the *parent-name* in the *submodule-stmt*. A submodule is a **child** of its parent. An **ancestor** of a submodule is that submodule or an ancestor of its parent. A **descendant** of a module or submodule is that program unit or a descendant of one of its child submodules.

A submodule accesses the scoping unit of its parent module or submodule by host association.

NOTE 11.12$\frac{1}{2}$

A procedure in a module or submodule has access to every entity in its ancestor program units. Even if no other program unit has access to the module or submodule, there may be an active procedure invoked by way of a procedure pointer or by means other than Fortran that has access to it. This may affect finalization (4.5.10.1) or undefinition (6.3.3.2, 16.4.2.1.3, 16.5.6).

[In the third line of the second paragraph of 12.3, at 253:15, replace ", but" by ". If the dummy arguments are redeclared in a separate module procedure body (12.5.2.5) they shall have the same names as in the corresponding module interface body (12.3.2.1); otherwise".]

[Add a new constraint after Constraint C1211 on page 255, at 255:26+.]

Constraint: (R1209) An IMPORT statement shall not appear within a module procedure interface body.

[After the third paragraph after constraint C1211 on page 255, at 255:35+, insert the following paragraph and note.]

A **module interface body** is an interface body in which MODULE appears in the *prefix* in its introductory *function-stmt* or *subroutine-stmt*. It declares the interface for a separate module procedure (12.5.2.5). A separate module procedure is accessible by use association if and only if its module interface body is accessible by use association. If the definition of its procedure body does not appear within the *module-subprogram-part* of the program unit in which the module interface body is declared, or one of its descendant submodules (11.2.3), the interface may be used but the procedure shall not be used in any way.

NOTE 12.3$\frac{1}{2}$

A module interface body shall not appear except within a specific or generic interface block within the *specification-part* of a module or submodule.

[In the first sentence of the fourth paragraph after constraint C1211 on page 255, at 255:36, insert ", that is not a module interface body," after "block".]

[In the first paragraph on page 256, move the sentence "An interface for a procedure named by an ENTRY statement may be specified by using the entry name as the procedure name in the interface body" to be a paragraph in its own right after Note 12.4.]

[In the first paragraph after Note 12.6 on page 257, at 257:3-4, delete the sentence "The characteristics of module procedures are not given in interface blocks, but are assumed from the module subprograms."]

[After the first paragraph after Note 12.7 on page 258, at 258:2+, insert the following paragraph.]

The characteristics of a module procedure may be specified by a module interface body, by the module procedure's declaration, or both.

[Add a fourth right-hand side for syntax rule R1228 on page 275, at 275:34+.]

or MODULE [(*module-or-submodule-name*)]

[Add the following constraint after constraint C1243 on page 275, at 275:38+.]

Constraint: (R1227) MODULE shall not appear in a *prefix* except within *subroutine-stmt* or *function-stmt* that introduces an interface body that is directly within a specific or generic interface block that is in turn directly within the *specification-part* of a module.

[Insert a new paragraph and a note immediately before Note 12.38 on page 276, at 276:33+]

The optional *module-or-submodule-name* in the MODULE prefix shall be the name of the module or submodule in which the separate module procedure definition appears (12.5.2.5).

NOTE 12.37$\frac{1}{2}$

The purpose of the optional *module-or-submodule-name* in the MODULE prefix is to allow processors to inline module procedures that have separate bodies. It is also useful to those maintaining a program. If it is not specified, the separate module procedure definition may not yet exist, it may be in the same program unit as the interface body, or it may be in an unspecified submodule – which makes it difficult to find.

[Insert a new subclause before 12.5.2.5 on page 280, at 280:14-, and renumber succeeding subclauses appropriately.]

12.5.2.5 Separate module procedure definition

A **separate module procedure** is a module procedure for which the interface is declared by a module interface body (12.3.2.1) in the *specification-part* of a module or submodule and the procedure body is defined by a module subprogram in a descendant of the program unit in which the module interface body is declared.

NOTE 12.40$\frac{1}{3}$

A separate module procedure can be accesseed by use association if and only if its module interface body can be accessed by use association. A separate module procedure that is not accessible by use association might still be accessible by way of a procedure pointer, a dummy procedure, or a type-bound procedure.

A module subprogram that defines a separate module procedure may respecify any of the characteristics declared in its module interface body. If any dummy arguments are respecified in that module subprogram, they shall all be respecified, in the same order and each with the same name and characteristics as specified in its module interface body. If a characteristic is specified in the module interface body but it is not specified in the module subprogram, the module procedure nonetheless has that characteristic. A characteristic that is specified in the module subprogram that defines a separate module procedure shall be specified in its module interface body.

NOTE 12.40$\frac{2}{3}$

As specified in 12.3.2.1, specifications within an interface body that do not specify characteristics or dummy argument names have no effect. Therefore, if a separate module procedure is to be recursive, or it is to have a result name different from the function name, these properties must necessarily be specified within the module subprogram.

[In the first line of the first paragraph after syntax rule R1236 in 12.5.2.6, at 280:27, insert ", submodule" after "module",]

[In the first line of the first paragraph of 16.4.1.3, at 400:32, insert ", a module interface body" * after "module subprogram". In the second line, insert "that is not a module interface body" after "interface body".]

[In the second line after the seventeen-item list in 16.4.1.3, at 401:28, insert "that does not define a separate module procedure" after "subprogram".]

[In item 2 of 16.5.6, at 411:30, insert "or submodule" after "module".]

[In item 4c of 16.5.6, at 411:38-39, insert "or submodule" after the first "module" and replace the second "module" by "that scoping unit".

[Replace Note 16.18 by the following.]

NOTE 16.18

A module subprogram inherently references the module or submodule that is its host. Therefore, for processors that keep track of when modules or submodules are in use, one is in use whenever any procedure in it or any of its descendant submodules is active, even if no other active scoping units reference its ancestor module; this situation can arise if a module procedure is invoked via a procedure pointer or by means other than Fortran.

[In item 4d of 16.5.6, at 411:40-41, insert "or submodule" after the first "module" and replace the second "module" by "that scoping unit".

[Insert the following definitions into the glossary in alphabetical order:]

ancestor (11.2.3) : A module or submodule, or an ancestor of the parent of that module or submodule. 415:12+

child (11.2.3) : A submodule, when considered in its relation to the module or submodule upon which it depends. 416:40+

descendant (11.2.3) : A module or submodule, or a descendant of a child of that module or submodule. 418:22+

parent (11.2.3) : A module or submodule, when considered in its relation to the submodules that depend upon it. 422:32+

submodule (2.2.5, 11.2.3) : A program unit that depends on a module or another submodule; it extends the program unit on which it depends. 425:14+

[Insert a new subclause immediately before C.9, at 465:33+:]

C.8.3.9 Modules with submodules

Each submodule specifies that it is the child of exactly one parent module or submodule. Therefore, a module and all of its descendant submodules stand in a tree-like relationship one to another.

If a module interface body that is specified in a module has public accessibility, and its separate module procedure body is defined by a subprogram in a descendant of that module, the procedure can be accessed by use association. No other entity in a submodule can be accessed by use association. Each program unit that accesses a module by use association depends on it, and each submodule depends on its ancestor module. Therefore, one can change a procedure body in a submodule without any possibility of changing the interface of the procedure. If a tool for automatic program translation is used, and even if it exploits the relative modification times of files as opposed to comparing the result of translating the module to the result of a previous translation, modifying a submodule cannot result in the tool deciding to reprocess program units that access the module by use association.

This is not the end of the story. By constructing taller trees, one can put entities at intermediate levels that are shared by submodules at lower levels, and have no possibility to affect anything that is accessible from the module by use association. Developers of modules that embody large complicated concepts can exploit this possibility to organize components of the concept into submodules, while preserving the privacy of entities that ought not to be exposed to users of the module and preventing cascades of reprocessing.

The following example illustrates a module, `color_points`, with a submodule, `color_points_a`, that in turn has a submodule, `color_points_b`. Public entities declared within `color_points` can be accessed by use association. Except for the characteristics and dummy argument names of separate module procedures that have module interface bodies that are accessible by use association, the submodules `color_points_a` and `color_points_b` can be changed without causing the appearance that the module `color_points` might have changed.

The module `color_points` does not have a *contains-part*, but a *contains-part* is not prohibited. It could be published as definitive specification of the interface, without revealing trade secrets contained within `color_points_a` or `color_points_b`. Of course, a similar module without the `module` prefix in the interface bodies would serve equally well as documentation – but the procedures would be external prodcedures. It wouldn't make any difference to the consumer, but the developer would forfeit all of the advantages of modules.

```
module color_points

  type color_point
    private
    real :: x, y
    integer :: color
  end type color_point

  interface      ! Interfaces for procedures with separate
                 ! bodies in the submodule color_points_a
    module subroutine color_point_del ( p ) ! Destroy a color_point object
      type(color_point) :: p
    end subroutine color_point_del
    ! Distance between two color_point objects
    real module function color_point_dist ( a, b )
      type(color_point) :: a, b
    end function color_point_dist
    module subroutine color_point_draw ( p ) ! Draw a color_point object
      type(color_point) :: p
    end subroutine color_point_draw
    module subroutine color_point_new ( p ) ! Create a color_point object
      type(color_point) :: p
    end subroutine color_point_new
  end interface

end module color_points
```

The only entities within `color_points_a` that can be accessed by use association are separate module procedures for which module interface bodies are provided in `color_points`. If the procedures are changed but their interfaces are not, the interface from program units that access them by use association is unchanged. If the module and submodule are in separate files, utilities that examine the time of modification of a file would notice that changes in the module could affect the translation of its submodules or of program units that access the module by use association, but that changes in submodules could not affect the translation of the parent module or program units that access it by use association.

The variable `instance_count` is not accessible by use association of `color_points`, but is accessible within `color_points_a`, and its submodules.

```
  submodule ( color_points ) color_points_a ! Submodule of color_points

    integer, save :: instance_count = 0

    interface                   ! Interface for a procedure with a separate
                                ! body in submodule color_points_b
     module subroutine inquire_palette ( pt, pal )
        use palette_stuff       ! palette_stuff, especially submodules
                                ! thereof, can access color_points by use
                                ! association without causing a circular
                                ! dependence because this use is not in the
                                ! module.  Furthermore, changes in the module
                                ! palette_stuff are not accessible by use
                                ! association of color_points
        type(color_point), intent(in) :: pt
        type(palette), intent(out) :: pal
      end subroutine inquire_palette

    end interface

  contains ! Invisible bodies for public interfaces declared in the module

    subroutine color_point_del ! ( p )
      instance_count = instance_count - 1
      deallocate ( p )
    end subroutine color_point_del
    function color_point_dist result(dist) ! ( a, b )
      dist = sqrt( (b%x - a%x)**2 + (b%y - a%y)**2 )
    end function color_point_dist
    subroutine color_point_new ! ( p )
      instance_count = instance_count + 1
      allocate ( p )
    end subroutine color_point_new

  end submodule color_points_a
```

The subroutine `inquire_palette` is accessible within `color_points_a` because its interface is declared therein. It is not, however, accessible by use association, because its interface is not declared in the module, `color_points`. Since the interface is not declared in the module, changes in the interface cannot affect the translation of program units that access the module by use association.

```
  submodule ( color_points_a ) color_points_b ! Subsidiary**2 submodule

  contains ! Invisible body for interface declared in the parent submodule
    subroutine color_point_draw ! ( p )
    ! Its interface is defined in an ancestor.
      type(palette) :: MyPalette
      ...; call inquire_palette ( p, MyPalette ); ...
    end subroutine color_point_draw
    subroutine inquire_palette
    ! "use palette_stuff" not needed because it's in the parent submodule
      ... implementation of inquire_palette
    end subroutine inquire_palette
    subroutine private_stuff ! not accessible from color_points_a
      ...
    end subroutine private_stuff

  end submodule color_points_b

  module palette_stuff
    type :: palette ; ... ; end type palette
  contains
    subroutine test_palette ( p )
    ! Draw a color wheel using procedures from the color_points module
      type(palette), intent(in) :: p
      use color_points ! This does not cause a circular dependency because
                       ! the "use palette_stuff" that is logically within
                       ! color_points is in the color_points_a submodule.
      ...
    end subroutine test_palette
  end module palette_stuff
```

There is a `use palette_stuff` in `color_points_a`, and a `use color_points` in `palette_stuff`. The `use palette_stuff` would cause a circular reference if it appeared in `color_points`. In this case it does not cause a circular dependence because it is in a submodule. Submodules are not accessible by use association, and therefore what would be a circular appearance of `use palette_stuff` is not accessed.

```
  program main
    use color_points
    ! "instance_count" and "inquire_palette" are not accessible here
    ! because they are not declared in the "color_points" module.
    ! "color_points_a" and "color_points_b" cannot be accessed by
    ! use association.
    interface ( draw ) ! just to demonstrate it's possible
      module procedure color_point_draw
    end interface
    type(color_point) :: C_1, C_2
    real :: RC
    ...
    call color_point_new (c_1)        ! body in color_points_a, interface in color_points
    ...
    call draw (c_1)                   ! body in color_points_b, specific interface
                                      ! in color_points, generic interface here.
    ...
    rc = color_point_dist (c_1, c_2)  ! body in color_points_a, interface in color_points
    ...
    call color_point_del (c_1)        ! body in color_points_a, interface in color_points
    ...
  end program main
```

Multilevel submodule systems can be used to package and organize a large and interconnected concept without exposing entities of one subsystem to other subsystems.

Consider a `Plasma` module from a Tokomak simulator. A plasma simulation requires attention at least to fluid flow, thermodynamics, and electromagnetism. Fluid flow simulation requires simulation of subsonic, supersonic, and hypersonic flow. This problem decomposition can be reflected in the submodule structure of the `Plasma` module:

```
                                            Plasma module
                                                  |
                           |---------------------|---------------------|
                           |                     |                     |
                      Flow submodule      Thermal submodule     Electromagnetics
                           |                                           Submodule
        |------------------|-------------------|
        |                  |                   |
    Subsonic           Supersonic          Hypersonic
```

Entities can be shared among the `Subsonic`, `Supersonic`, and `Hypersonic` submodules by putting them within the `Flow` submodule. One then need not worry about accidental use of these entities by the `Thermal` or `Electromagnetics` modules, or the development of a dependency of correct operation of those subsystems upon the representation of entities of the `Flow` subsystem as a consequence of maintenance.