https://ntrs.nasa.gov/search.jsp?R=19870013758 2020-03-20T10:38:22+00:00Z

(NASA-TM-89376) Ada STYLE GUIDE (VERSION 1.1) (NASA) 106 p Avail: NTIS HC A06/MF A01; single copies available from NASA/GSFC, Code 552, Greenbelt, Md. 20771 CSCL 09B

N87-23191

Unclas
G3/61 0076839

ADA® STYLE GUIDE (Version 1.1)

MAY 1987

ADA IS A REGISTERED TRADEMARK OF THE U.S. GOVERNMENT, ADA JOINT PROGRAM OFFICE.

NASA
National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/ Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (Systems Development and Analysis Branch)
The University of Maryland (Computer Sciences Department)
Computer Sciences Corporation (Flight Systems Operation)

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document. This document was prepared as a joint effort of representatives of the GSFC Ada User's Group. The principal author is

Edwin V. Seidewitz (GSFC)

Additional contributors to this document include William Agresti (Computer Sciences Corporation (CSC)), Daniel Ferry (CSC), David LaVallee (Ford), Paul Maresca (AdaSoft), Robert Nelson (GSFC), Kelvin Quimby (CSC), Jacob Rosenberg (GSFC), Daniel Roy (Century Computing), Allyn Shell (CSC), and J. T. Thompson (Ford).

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 552
NASA/GSFC
Greenbelt, Maryland 20771

PRECEDING PAGE BLANK NOT FILMED

INTENTIONALLY BLANK

ABSTRACT

Ada is a programming language of considerable expressive power. The Ada Language Reference Manual provides a thorough definition of the language. However, it does not offer sufficient guidance on the appropriate use of Ada's powerful features. For this reason, the Goddard Space Flight Center Ada User's Group has produced this style guide which addresses such "program style" issues. The guide covers three areas of Ada program style: the structual decomposition of a program, the coding and use of specific Ada features, and the textual formatting of a program.

PRECEDING PAGE BLANK NOT FILMED

PAGE IV INTENTIONALLY BLANK

CONTENTS

PRECEDING PAGE BLANK NOT FILMED

PAGE vi INTENTIONALLY BLANK

CHAPTER 1

INTRODUCTION

Ada is a programming language of considerable expressive power. The Ada Language Reference Manual [RM] provides a thorough definition of the language. However, it does not offer sufficient guidance on the appropriate use of Ada's powerful features. For this reason, the Goddard Space Flight Center Ada User's Group has produced this style guide which addresses such "program style" issues.

1.1 SCOPE OF THE GUIDELINES

Program source code serves two functions: to specify an algorithm to be performed on a computer, and to communicate this algorithmic design to other human readers. Program style relates to how well a program meets the second function. It is a consistent manner of using the features of a programming language to promote the readability and understandability of a program. This is a matter of the form of the final, delivered program source code, as opposed to the process of developing the code.

The guidelines in this document cover three areas of Ada program style:

- STRUCTURE: the structural decomposition of a program
- CODING: the coding and use of specific Ada features
- FORMAT: the textual formatting of a program which does not effect the meaning of the program

The guidelines are also classified as standards or recommendations. A conforming Ada program obeys all the standards in this document. Generally, recommendations should also be followed unless there is a good reason to do otherwise.

1.2 STRUCTURE OF THE DOCUMENT

This guide contains fourteen chapters corresponding to the fourteen chapters of the Ada LRM. This provides a standard frame of reference for the discussion of Ada features. Each chapter is subdivided into sections on STRUCTURE, CODING and FORMAT guidelines. Note that some chapters do not include all three of these subsections. Standards

are marked with a star (*), while recommendations are unmarked. Where appropriate, the guide includes examples, justifications and references for various guidelines. Some chapters have an EXAMPLES section giving additional longer examples for the chapter.

1.3 GOALS

While some of what constitutes "good style" is subjective and somewhat arbitrary, it is important that the style of a program be consistent throughout the program. The primary goal of this guide is to promote such consistent use of good style across a large number of Ada programs. The whole intent of "good style" is to increase the readability of these programs. Therefore, the guidelines under each of the areas of structure, coding and format follow from general principles of program readability and understandability discussed below.

In the structure area, a program should model the structure of the problem to be solved. The program should reflect the natural levels of abstraction in the problem domain, so that the reader can reasonably comprehend each level individually.

In the coding area, there are several features in Ada which are unfamiliar to many programmers. There is thus a tendency to either underuse these features or to use them inappropriately. A feature of Ada should generally not be ignored, but neither should it be used in excess. The coding guidelines highlight the proper use of Ada features.

Finally, the textual format of a program should be pleasing to the eye and promote the readability and understandability of the program. The format should highlight the structure of a program and the role of a program as a model of the problem domain. Just as the careful layout of a book can enhance written communication, the careful layout of a program can enhance the communication of algorithmic design to another human. The consistent use of formating style is especially important, because it allows readers to become accustomed to the familiar layout of program constructs. An automated formating program is particularly helpful, but even in the absence of such a tool, much can be gained from a common format style.

CHAPTER 2

LEXICAL ELEMENTS

2.1 CODING

2C1 The Package Standard

(a)* Language words with predefined meanings in package Standard should not be redefined.

2C2 Comments

(a)* Comments should be used to add information for the reader or to highlight sections of code, and should not merely paraphrase the code.

2.2 FORMAT

2F1 Indentation

(a)* The standard indentation is two spaces.

2F2 Character Set

(a)* Full use should be made of the ISO character set where available. Alternate character replacements should only be used when the corresponding graphical symbols are not available.

2F3 Upper / Lower Case

(a)* Reserved words and attributes should appear in lower case.

(b)* All identifiers except type, enumeration value and attribute identifiers should be in mixed upper and lower case. The first letter of each word in the identifier should be in upper case with other characters in lower case, unless a word is normally written in all upper case (eg. acronyms).

```
Display_Device
Number_Of_User_Names
Get_FHST_Data
Package_Name
```

(c)* Type and enumeration value identifiers should appear in all upper case.

```
LONG_INTEGER
AUTHORITY_LEVEL

(RED, GREEN, BLUE)
(ARMY, AIR_FORCE, NAVY, MARINES)
```

2F4 Identifiers

(a)* Identifier names should be meaningful and easily distinguishable from each other, except possibly for loop parameters, array indices and common mathematical variables, which may be as short as only one character.

(b)* Distinct words in identifiers should always be separated by underscores.

(c)* The use of abbreviations in identifiers should be avoided. When used, an abbreviation should be significantly shorter than the word it abbreviates, and its meaning should be clear. The same abbreviations should be used consistently throughout a project. [ACGE]

2F5 Spaces

(a)* Single spaces should be used consistently between lexical elements to enhance readability.

2F6 Blank Lines

(a)* Blank lines should be used to group logically related lines of text.

> A careful use of blank lines can greatly enhance readability by making the logical structure of a sequence of lines more textually obvious. However, the overuse of blank lines (e.g., "double spacing") defeats the very purpose of grouping and can actually reduce readability. Blank lines should thus always be used with grouping in mind and not just to increase white space.

(b)* A blank line should always follow a construct whose last line is not a the same indentation level as its first line.

```
type COMPLEX is
  record
    Real      : FLOAT;
    Imaginary : FLOAT;
  end record;

-- Followed by a blank line
```

2F7 Continuations

(a)* Statements extending over multiple lines should always be broken BEFORE reserved words, operator symbols or one of the following symbols:

```
:  |  =>  ..  :=
```

but they should be broken AFTER a comma (","). Unless otherwise specified in later guidelines, all the continuation lines should be indented at least two levels with respect to the original line they continue.

```
Corrected_Value := (1 + Sensor_Scale) * Raw_Value
    + Distortion_Factor * Distortion_Value + Sensor_Bias;
```

(b)* Long strings extending over more than one line should be broken up at natural boundaries, appropriate to the meaning of the contents of the string, if any.

```
"This is a rather long string, so it is likely that "
    & "it will extend over more than one line"
```

2F8 Comments

(a)* Comments should begin with the "--" aligned with the indentation level of the code that they describe, or to the right of the code, aligned with other such comments.

```
-- Check if the user has special authorization
if Authority = SPECIAL then
  Display_Special_Menu;     -- All operations are allowed
else
  Display_Normal_Menu;      -- Only normal operations allowed
end if;
```

CHAPTER 3

DECLARATIONS AND TYPES

3.1 CODING

3C1 Constants

(a)* An object should be declared constant if its value is intended not to change. [NW]

Declaring an object to be constant clearly signals both the human reader and the compiler the intention that its value will not change. This not only increases readability, it also increases reliability because the compiler will detect any attempt to tamper with the object. Also, it can result in some decrease in executable size and better run time efficiency.

(b)* Defining a constant object is preferable to using a numeric literal or expression with constant value, as long as the constant object has an intrinsic conceptual meaning. [NW]

There is no use to defining a constant object when a numeric literal is obviously more appropriate, for example using "One" instead of "1". However, the use of constant objects with intrinsic meaning (such as "Buffer_Size" or "Field_Of_View") can greatly increase the readability of code. Further, the code is more maintainable since a change in a value will be localized to the constant declaration.

(c)* A named number (i.e., a constant object with type universal-integer or universal-real) should be used only for values that are truly "universal" and "typeless". Other numeric constants should be declared with an explicit type. [NW]

Such constants as "Pi" and cardinal integers (e.g., a "number of things") should be named numbers. Note also that declaring a constant in terms of a predefined numeric type (INTEGER, FLOAT, etc.) has no advantage over a named number since these predefined types provide only range and accuracy constraints and no additional conceptual meaning. In fact, since the range and accuracy of predefined numeric types is implementation-defined, portability can be increased by using named numbers, in those cases where a constant of a user-defined type is not more

appropriate.

```
Number_Of_Sensors                    -- This is a named number
  : constant := 4;

Main_Sensor_Number
  : constant SENSOR_INDEX := 2;
```

3C2 Types

(a)* Separate types should be used for values that belong to logically independent sets, and for distinct concepts. [NW]

```
type X_COORDINATE is
  range 1 .. 640;

type Y_COORDINATE is
  range 1 .. 480;

type PIXEL_VALUE is
  range 0 .. 255;

type IMAGE_GRID is
  array (X_COORDINATE, Y_COORDINATE) of PIXEL_VALUE;
```

A data type characterizes a set of values and a set of operations applicable to objects of the type. In the above example, each coordinate has a type because coordinates are independent entities. Explicitly declaring these types makes the concepts more obvious to a human reader and also allows the compiler to detect mistakes such as:

```
  Image (Y, X) := Pixel;      -- Should be "(X, Y)"
```

The drawback of this kind of typing is that the following construct is illegal:

```
if X = Y then          -- ILLEGAL since X and Y have different types
  ...
```

A type conversion must be used:

```
if X = X_COORDINATE(Y) then
  ...
```

Note that, depending on context (and compiler quality), there may or may not be some run time penalty associated with type conversion (e.g., testing of range constraints).

3C3 Enumeration Types

(a)* An enumeration type should always be used in preference to an integer type, unless the logical nature of the concept to be modeled demands the other. [NW]

For example the type:

```
type DEVICE_MODE is
  (READ_ONLY, WRITE_ONLY, READ_WRITE);
```

is preferable to encoding DEVICE_MODE as an integer 0, 1 or 2.

3C4 Floating Types

(a) To enhance portability, the range and accuracy of a floating point type should generally be specified.

The precision for the predefined floating types (FLOAT, etc.) is implementation-dependent, though all implementations should provide at least 6 decimal digits of accuracy. Explicitly declaring floating point ranges can yield more reliable and more efficient as well as more portable code.

3C5 Record Types

(a)* A record type should be used instead of an array type even when all the record components have the same type, as long as each component can be sensibly named and the components do not need to be dynamically indexed. [NW]

For example, the definition:

```
type COMPLEX is
  record
    Real      : FLOAT;
    Imaginary : FLOAT;
  end record;
```

is preferable to defining COMPLEX as an array of two FLOATs.

(b) Overcomplicated record structures should be avoided by grouping related data into subrecord types. [NW]

```
type COORDINATE is
  record
    Row    : FLOAT;
    Column : FLOAT;
  end record;

type WINDOW is
  record
    Top_Left     : COORDINATE;
    Bottom_Right : COORDINATE;
  end record;
```

(c) Enumeration types should be used for discriminants of record variants whenever possible. A discriminant should generally have a default initialization only if the the discriminant value is intended to change over the lifetime of an object.

3C6 Access Types

(a) Generally, access types should not be used when static types and stack allocation would be sufficient.

Generally access types should be used only when it is necessary to have data structures with dynamic pointers or to dynamically create objects. However, access types may be needed for static objects if this leads to a more consistent programming style (e.g., so that similar static and dynamic objects are treated identically). For example, if linked lists are used in a program, there may be some lists which are constant, but which are still implemented as linked lists using access types. This would allow, for example, passing these constant lists to subprograms which also handle dynamic lists.

3C7 Object Declarations

(a)* Each object declaration should declare only one object.

For example, the following objects should be declared in separate declarations even though they are all of the same type:

```
Table_Size
  : TABLE_RANGE;

Table_Index
  : TABLE_RANGE;

Current_Entry
  : TABLE_RANGE;
```

(b)* An object should not be declared using an unnamed constrained array definition.

The unnamed array definition is the only case in Ada where an object can be declared to be of a type which does not have a name. Instead, the array type should be named in an array definition, and that name used in the object declaration, even if there is only one object declared of that type.

```
type POOL_TYPE is
  array (POOL_RANGE) of CHARACTER;

POOL
  : POOL_TYPE;
```

(c) Objects should generally be initialized. Where possible, objects should always be initialized by their declaration, rather than in later code.

```
Is_Found
  : BOOLEAN := FALSE;
```

3.2 FORMAT

3F1 Commenting

(a)* Type declarations (or groups of declarations) should be commented to indicate what is being defined, if that is not obvious from the type declaration itself.

```
type VELOCITY is      -- Inertial velocity relative to the Earth
  array (1..3) of FLOAT;
```

(b)* Object declarations should be commented if the object definition is unclear from the object and type identifiers alone. Note that those properties of an object obtained from its type should not be repeated in comments on the object declaration.

```
Spacecraft_Velocity  -- Spacecraft orbital velocity, assuming a
  : VELOCITY;        -- circular orbit
```

3F2 Indentation

(a)* All declarations in a single declaration part should begin at the same indentation level.

3F3 Type Definitions

(a)* Array type definitions should have one of the following formats:

```
type <type name> is
  array <index definition> of <subtype indication>;
```

```
type <type name> is
  array <index definition>
    of <subtype indication>;
```

(b)* Record type definitions should have one of the following formats:

```
type <type name> is
  record
    <component declaration>
    <component declaration>
  end record;
```

```
type <type name>
  ( <discriminant declaration>;
    <discriminant declaration> ) is
  record
    <component declaration>

    case <discriminant name> is
      when <choices> =>
        <component declaration>
        <component declaration>

    end case;

  end record;
```

All <component declarations> and <discriminant declarations> should be formatted like object declarations (guideline 3F4).

(c)* Other type definitions should be formatted as follows:

```
type <type name> is
  <type definition>;

subtype <type name> is
  <subtype indication>;
```

Long enumeration type definitions should be formatted into easily readable columns.

3F4 Object Declarations

(a)* Object declarations should have one of the following formats. The preferred formats are:

```
<object name>
  : <subtype indication> := <expression>;

<object name>
  : <subtype indication>
    := <expression>;
```

Declarations containing short identifiers may also be formatted all on one line:

```
<object name>  : <subtype indication>  := <expression>;
```

In this case, all such declarations textually grouped together or appearing as components in a single record definition or in a single parameter list should have their ":" and ":=" symbols aligned.

3.3 EXAMPLES

See also examples 6X5, 7X2, 9X3 and 14X1.

Example 3X1

```
type SENSOR_ARRAY is
  array (NATURAL range <>) of SENSOR;

UARS_Sensors                              -- Sensor configuration for the
  : SENSOR_ARRAY(1 .. Num_Sensors); -- UARS control system
```

Example 3X2

```
type COMPLEX is
  record
    Real        : FLOAT;
    Imaginary   : FLOAT;
  end record;
```

Example 3X3

```
type DEVICE is
  (PRINTER, DISK, DRUM);

type STATE is          -- Operational state of a
  (OPEN, CLOSED);      -- device.

type PERIPHERAL
  ( Unit : DEVICE := DISK ) is
  record
    Status
      : STATE;

    case Unit is
      when PRINTER =>
        Lines_Per_Page
          : INTEGER range 1 .. Page_Size;

      when DISK | DRUM =>
        Cylinder
          : CYLINDER_INDEX;
        Track
          : TRACK_NUMBER;

    end case;

  end record;
```

CHAPTER 4

NAMES AND EXPRESSIONS

4.1 CODING

4C1 Aggregates

(a)* Aggregates are preferable to individually setting all or most of the components of an array or record.

(b) Named aggregates should be used where possible. [NW]

(c) The "others" choice should not be used within aggregates without good reason. [NW]

4C2 Static Expressions

(a) Where possible, universal expressions are preferable to static (but not universal) expressions, which are in turn preferable to dynamic expressions. [NW]

Since they do not depend on run time dynamics, static expressions are easier for a human reader to understand. Also, universal expressions maximize accuracy and portability, and static expressions eliminate run time overhead.

4C3 Short-Circuit Control

(a) Short-circuit control forms should generally be used only to avoid evaluation of an undefined or illegal expression. Short circuit operators should not be used to optimize execution.

```
(N /= 0) and then (Total/N > Limit)

(Index = 0) or else User(Index).Not_Available
```

The short-circuit control forms should be used to signal to a human reader that the correctness of the second condition depends on the results of the first. They should not be used for micro-efficiency reasons, concerns better handled by an optimizing compiler. If efficiency considerations are substantially important, "if" statements should be used instead of the short-circuit forms with functions used to avoid repeated

code, if necessary. [ACGE]

4C4 Type Qualification

(a)* An explicit type conversion should not be used if a type qualified expression is meant. [NW]

```
Good:  LONG_FLOAT'(3.14159)
Bad:   LONG_FLOAT (3.14159)
```

A qualified expression is used to state explicitly the type, and possibly subtype, of a value. A type conversion, however, results in the dynamic conversion of a value to a target type. Sometimes a type conversion can be used to serve the purpose of a type qualification. However, if the operand is already of the desired base type, a conversion is not really necessary and a qualification should be used instead.

(b)* Situations where type qualification is necessary should be avoided if possible. Other than where absolutely necessary, type qualification may be justified only if it makes the program clearer to a reader. [NW]

The main case to avoid is when the type of an enumeration literal or aggregate is not known from context. For example:

```
type COLOR is
  (BLACK, RED, GREEN, BLUE, WHITE);

type LIGHT is
  (RED, YELLOW, GREEN);

procedure Set
  ( Color_Code : in COLOR );

procedure Set
  ( Color_Code : in LIGHT );

...

Set(COLOR'(RED));        -- Type qualification must be used here to

Set(LIGHT'(RED));        -- resolve the overloading of Set and RED
```

It would be better in this case to rename on of the Set procedures, or to at least give them different parameter names so the overloading could be resolved using named notation.

4.2 FORMAT

4F1 Names

(a)* The name for a type should be a common noun indicating the class of the objects it contains.

```
DEVICE
AUTHORITY_LEVEL
USER_NAME
PHONE_LIST
```

A type name may also end with the suffix "TYPE".

```
EMPLOYEE_TYPE
SCHEDULE_TABLE_TYPE
COLOR_TYPE
```

(a)* The names of non-BOOLEAN valued objects should be nouns, preferably more precise than the names of types.

```
Current_User
  : USER_NAME;

Output_Device
  : DEVICE;

Schedule_Table
  : SCHEDULE_TABLE_TYPE;

New_Employee
  : EMPLOYEE_TYPE;
```

BOOLEAN valued objects should have predicate-clause (e.g., "Is_Open") or adjective names.

```
User_Is_Available
List_Empty
Done
Not_Ready
Is_Waiting
```

4F2 Parentheses

(a) Syntactically redundant parentheses should generally be used to enhance the readability of expressions, especially by indicating the order of evaluation. [NW]

For example:

```
Variance := (Roll_Error ** 2) + ((Yaw_Error ** 2) / 2);
```

4F3 Aggregates

(a) When longer than two or three components, or whenever readability is improved, named aggregates should be formatted as indicated below, with one association per line and the "=>" arrows aligned.

```
Output_Device :=
  ( Device          => DISK,
    Status          => CLOSED,
    Cylinder        => 1,
    Track           => Startup_Track_Num );
```

(b) Aggregates for tabular data structures may instead be formatted in a tabular fashion, so as to enhance readability.

4F4 Continuation

(a) When a long expression is broken over more than one line, it should be broken near the end of the line before an operator symbol with the lowest reasonable precedence.

```
Corrected_Value := (1 + Sensor_Scale) * Raw_Value
    + Distortion_Factor * Distortion_Value + Sensor_Bias;
```

CHAPTER 5

STATEMENTS

5.1 CODING

5C1 Slice Statements

(a)* Array slice assignments should be used rather than loops, to copy all or part of an array.

This is more readable and less error prone, especially in the case of slices with overlapping ranges. [NW]

```
Client_List (Last_Client .. Number_Of_Clients)
    := New_Clients (1 .. Num_New_Clients)
```

5C2 If Statements

(a)* An "if" statement should not be used to create the effect of a "case" statement controlled by the value of an enumeration type other than BOOLEAN. [NW]

5C3 Case Statements

(a)* A "case" statement should not be controlled by a BOOLEAN value. [NW]

(b) When possible, the explicit listing of all choices on a "case" statement is preferable to the use of an "others" clause.

This makes it easier for a human reader to see that the proper actions are being taken in all cases. Further, if the enumeration type of the control expression is modified, the compiler will indicate overlooked alternatives. However, there are cases when an "others" clause makes sense. For example, if the control expression is of type character, then it is usually best to use an "others" clause to handle the "undesired characters" case.

5C4 Block Statements

(a)* Blocks should be used cautiously to introduce local declarations or to define a local exception handler. [NW]

To some extent, a block can be thought of as a procedure which is hard coded in-line. However, a procedure call contributes to readability precisely by not having its source code in-line (providing a "functional abstraction"). Therefore blocks should always be used cautiously and only for specific purposes. Thought should always be given to using a procedure call instead of a block to improve readability.

(b)* Declarations of objects used only within a block should be nested within the block. [NW]

5C5 Exit Statements

(a) "Exit" statements should be used cautiously, and only when they significantly enhance the readability of the code.

It is often more readable to use "exit" than to try to add BOOLEAN variables to a "while" loop condition to simulate exits from the middle of a loop. However, it can be difficult to understand a program where loops can be exited from multiple places. It is best to limit the use of "exit" statements to one per loop, if possible, and it is generally more readable to use "exit when". Use "if...then...exit; end if" when "last wishes" processing is needed.

5C6 Return Statements

(a) It is preferable to minimize the number of return points from a subprogram, as long as this does not distract from the natural structure or readability of the subprogram.

5C7 Goto Statements

(a)* Neither "goto" statements nor labels should ever be used.

Use of the "goto" makes the textual structure of code less reflective of its logical structure. Possible uses of the "goto" statement can always be handled by other constructs in Ada. Cases in Ada when the "goto" still seems appropriate almost always indicate poorly designed code. It is better to redesign the code than to use the "goto" statement.

5.2 FORMAT

5F1 Statement Sequences

(a)* Blank lines should be used liberally to break sequences of statements into short, meaningful groups (see also guideline 2F6).

```
    Put_Line ("Welcome to the Electronic Message System");

    Logon_User(Current_User);
    User_Directory.Lookup
      ( User_Name     => Current_User,
        Authority     => User_Authority );

    if User_Authority = SPECIAL then
      Put_Line ("** You have SPECIAL authorization **");
    end if;
```

5F2 If Statements

(a)* "If" statements should have the following format:

```
    if <condition> then
      <statement>
      <statement>
    elsif <condition> then
      <statement>
      <statement>
    else
      <statement>
      <statement>
    end if;
```

(b)* Multiple conditions in an "if" clause should be grouped together, placed on appropriate lines, and aligned so as to enhance clarity. [Gardner]

5F3 Case Statements

(a)* "Case" statements should have the following format:

```
    case <expression> is
      when <choices> =>
        <statement>
        <statement>
      when others =>
        <statement>
        <statement>
    end case;
```

5F4 Loop Statements

(a)* "Loop" statements should have one of the following formats:

```
<loop name>:
<iteration scheme> loop
  <statement>
  <statement>
end loop <loop name>;

<iteration scheme> loop
  <statement>
  <statement>
end loop;
```

(b) A loop should preferably have a loop identifier.

5F5 Block Statements

(a)* Block statements should have the following format:

```
<block name>:
declare
  <declaration>
  <declaration>
begin
  <statement>
  <statement>
exception
  when <exceptions> =>
    <statement>
    <statement>
end <block name>;
```

(b)* Blocks should always have a block identifier.

5.3 EXAMPLES

See also examples 9X2, 9X3, 9X4, 14X1 and 14X2.

Example 5X1

```
    if Security_Level = 0 then
      Message_Classification := UNCLASSIFIED;

    elsif Security_Level > User_Clearance then
      Message_Classification := PROTECTED;

    else
      Message_Classification := Classification (Security_Level);

    end if;
```

Example 5X2

```
    case Sensor is
      when ELEVATION =>
        Record_Elevation (Sensor_Value);
      when AZIMUTH =>
        Record_Azimuth (Sensor_Value);
      when DISTANCE =>
        Record_Distance (Sensor_Value);
    end case;
```

Example 5X3

```
    Read_File:
    loop

      Text_IO.Get(File1, Next_Record);
      Store_Record(Next_Record);

      exit when Text_IO.End_Of_File(File1);

    end loop Read_File;


    Compute_Total_Taxes:
    while Next /= Head loop

      Total_Taxes := Total_Taxes + Next.Pay_Period_Deductions;
      Next := Next.Successor;

    end loop Compute_Total_Taxes;
```

```
    Merge_Files:
    for N in 1 .. Max_Num_Files loop

      Get_Items:
      for J in 1 .. Max_Num_Items loop

        Get_New_Item(New_Item);
        Merge_Item(New_Item, Storage_File);

        exit Merge_Files when New_Item = Terminal_Item;

      end loop Get_Items;

    end loop Merge_Files;
```

Example 5X4

```
    Swap_Integers:
    declare

      Temp
        : constant INTEGER := U;

    begin  -- Swap_Integers;

      U := V;
      V := Temp;

    end Swap_Integers;


    Check_Entry:
    begin

      Int_IO.Get(Value);
      Update(Value);

    exception

      when Data_Error =>
        Text_IO.New_Line;
        Text_IO.Put_Line("Value entry error.");
        Entry_Error_Flag := TRUE;

    end Check_Entry;
```

CHAPTER 6

SUBPROGRAMS

6.1 STRUCTURE

6S1 Cohesion

(a)* A subprogram should perform a single, conceptual action (ie, should be "functionally cohesive"). [Myers] [YC]

> The use of a subprogram increases readability by hiding the details of how an action is performed and giving it a descriptive name. A subprogram should perform only a single conceptual action so that its use can be understood as independently as possible from its implementation details and it can be given a self-documenting name. Note that simply shortening a program by placing "repeated code" into subprograms must be considered a secondary goal. Thus it is quite acceptable to have subprograms which are only called at one place, so long as those programs define cohesive actions.

6.2 CODING

6C1 Parameters

(a) Subprograms with equivalent parameters should generally declare each parameter in the same position with the same identifier. [NW]

(b) Parameters with default expressions should usually be used only when they have very well known default values and/or they are defaulted most of the time and the default is only over-ridden in special circumstances. [NW]

(c) Parameters with default expressions should generally be placed at the end of the parameter list, so that they may be omitted if desired in calls using positional notation.

6C2 Recursion

(a) A recursive subprogram should generally be used only if it is conceptually simpler for a given problem than a corresponding iterative subprogram.

Many people have difficulty in understanding a program wich uses recursion extensively. However, there are many cases where a recursive solution is considerably simpler and clearer than an iterative one. This is especially true, for example, for traversing complicated data structures such as tree and graph structures.

6C3 Functions

(a) A subprogram without side-effects returning a single value should generally be written as a function. [NW]

Since functions can be called from within expressions, there is more freedom in how a function can be used. For example, if a function is to be called only once within some other subprogram, it can be used to initialize a constant object.

```
procedure Process_Sensor_Data is

  Main_Sensor_Data
    : constant SENSOR_DATA
      := Read_Sensor (Main_Sensor_Index);

begin
  ...
```

However, if this sort of freedom is specifically not desired, or if a subprogram has side effects, then use of a procedure should be considered instead of a function, even if the subprogram returns only a single value.

6C4 Overloading

(a)* Overloading of subprograms should not be used except in the following cases:

- widely used utility subprograms which perform identical or very similar actions on arguments of different types (eg, square-root of integer and real arguments)

- overloading of operator symbols

Note that this is not meant to cover subprograms with identical names in different packages, unless both subprograms are visible through "use" clauses for their packages.

(b)* Operator symbols should be overloaded only when the new operator definitions comply closely with the traditional meaning of the operator (eg, "+" for vector addition).

For example "+" might be used for vector addition, but should certainly not be used for vector dot product.

6.3 FORMAT

6F1 Subprogram Names

(a)* Except as indicated below, a subprogram name should be an imperative verb phrase describing its action.

```
Obtain_Next_Token
Increment_Line_Counter
Create_New_Group
```

Non-BOOLEAN valued function names may also be noun phrases.

```
Top_Of_Stack
X_Component
Successor
Sensor_Reading
```

BOOLEAN valued functions should have predicate-clause names.

```
Stack_Is_Empty
Last_Item
Device_Not_Ready
```

6F2 Subprogram Header

(a)* Each subprogram specification, body or stub should be preceded by a header comment block containing at least the subprogram name and the indication SPEC, BODY, SPEC & BODY, STUB or SUBUNIT.

```
--  ..........................
--  .                        .
--  .    Obtain_Next_Token   .    SPEC
--  .                        .
--  ..........................
```

6F3 Subprogram Declarations

(a)* Procedure declarations should have the following format:

```
procedure <procedure identifier>
  ( <parameter specification>;
    <parameter specification> );

--| <documentary comments>
```

Each <parameter specification> should be formatted like an object declaration (guideline 3F4). The documentary comments should follow guideline (d) below.

(b)* Function declarations should have the following format:

```
function <function designator>
  ( <parameter specification>;
    <parameter specification> )
  return <type mark>;

--| <documentary comments>
```

Each <parameter specification> should be formatted like an object declaration (guideline 3F4). The <documentary comments> should follow guideline (d) below.

(c)* Parameter mode indications should always be used in procedure specifications. In a function specification, mode indications should either be used for all of the parameters or none of the parameters.

(d)* Subprogram declarations should be followed by AT LEAST the following documentation:

```
--| Purpose
--| A description of the purpose and function of the subprogram.
--|
--| Exceptions
--| A list of all exceptions which may propagate out of the
--| subprogram, and a description of when each would be raised.
--|
--| Notes
--| Additional comments on the use of the subprogram.
```

The "Exceptions" and "Notes" headings should be included even if these sections are empty. An empty section may be indicated by placing the annotation "(none)" after the appropriate header. Only in the case that the subprogram declaration is a compilation unit, the following section should be added to the documentation:

```
--| Modifications
--| A list of modifications made to the subprogram DECLARATION.
--| Each entry in the list should include the date of the change,
--| the name of the person who made the change and a description
--| of the modification. The first entry in the list should
--| always be the initial coding of the subprogram declaration.
```

6F4 Subprogram Bodies And Stubs

(a)* Subprogram bodies should have the following format:

```
separate (<parent name>)
<subprogram specification> is

--| <documentary comments>

  <declaration>
  <declaration>

begin   -- <subprogram name>

  <statement>
  <statement>

exception
  when <exceptions> =>
    <statement>

end <subprogram name>;
```

The <subprogram specification> should be formatted as in a subprogram declaration (guideline 6F3). The <documentary comments> should follow guideline (b) below. Note that the "end" of a subprogram should always include the subprogram name.

(b)* Subprogram bodies should have AT LEAST the following documentation placed immediately after the subprogram header:

```
--| Notes
--| Comments on the design, implementation and use of the
--| subprogram.
```

The "Notes" heading should be included even if the section is empty. An empty section may be indicated by the comment "Notes (none)". Only in the case of a subprogram body which is a compilation unit, the following section should be added to the documentation:

```
--| Modifications
--| A list of modifications made to the subprogram BODY.  Each
--| entry in the list should include the date of the change,
--| the name of the person who made the change and a description
--| of the modification.  The description should identify exactly
--| where in the comilation unit that the change was made.  The
--| first entry in the list should always be the initial coding
--| of the subprogram body.
```

If there is no declaration or stub for a subprogram, then the subprogram body should also include all the documentation required for a subprogram declaration (guideline 6F3).

(c)* Subprogram stubs should have the following format:

```
<subprogram specification> is separate;
```

where the <subprogram specification> is formatted as in a subprogram declaration (guideline 6F3). If there is no previous declaration for a separate subprogram, then the subprogram stub should be followed by the same documentatary comments required for a subprogram declaration (guideline 6F3).

6F5 Named Parameter Association

(a) Named parameter association should generally be used for procedure calls of more than a single parameter. Positional parameters are generally prefered for function calls.

(b) Named and positional parameter associations should generally not be mixed in a single subprogram call.

(c) Named parameter associations should generally appear one to a line with formal parameters, "=>" symbols and actual parameters aligned.

```
Obtain_Next_Token
  ( File          => Current_Source_File,
    Position      => Current_Column,
    Token         => Next_Token );
```

6.4 EXAMPLES

See also examples 7X3, 9X3, 12X1, 12X3, 14X1 and 14X2.

Example 6X1

```
--  ..............................
--  .                            .
--  .    Obtain_Next_Token       .    SPEC
--  .                            .
--  ..............................

procedure Obtain_Next_Token

  ( File                                    -- Sequential text file.
      : in out Parser_Types.FILE;

    Token
      : out Parser_Types.TOKEN_TYPE;

    Position                                -- Column position of the
      : in Parser_Types.COL_NUM_TYPE        -- beginning of the next
        := 0);                              -- token

--| Purpose
--| This procedure scans the current input line from the point at
--| which it was last called and returns the next token.
--|
--| Exceptions
--| Source_File_Not_Open  --  Raised if the input file is not open
--|
--| Notes (none)
```

Example 6X2

```
--  ......................
--  .                    .
--  .    Decode_Token    .    SPEC
--  .                    .
--  ......................

function Decode_Token

  ( File                                    -- Sequential text file.
      : Parser_Types.FILE;

    Token
      : Parser_Types.TOKEN_TYPE )

  return Parser_Types.TOKEN_TYPE;

--| Purpose
--| This function returns the ordinal value of the decoded token.
--|
--| Exceptions
--| Illegal_Token  --  raised if the token is not legal
--|
--| Notes
--| This function will later be changed to a procedure.
```

Example 6X3

```
--  ..........................
--  .                        .
--  .    Obtain_Next_Token   .    STUB
--  .                        .
--  ..........................

procedure Obtain_Next_Token

  ( File                                    -- Sequential text file.
      : in out Parser_Types.FILE;

    Token
      : out Parser_Types.TOKEN_TYPE;

    Position                                -- Column position of the
      : in Parser_Types.COL_NUM_TYPE        -- beginning of the next
        := 0 ) is separate;                 -- token.
```

Example 6X4

```
--  ......................
--  .                    .
--  .    Decode_Token    .    STUB
--  .                    .
--  ......................

function Decode_Token

  ( File
      : Parser_Types.FILE;

    Token
      : Parser_Types.TOKEN_TYPE )

  return Parser_Types.TOKEN_TYPE is separate;
```

Example 6X5

```
--  ..............................
--  .                            .
--  .    Obtain_Next_Token       .    SUBUNIT
--  .                            .
--  ..............................

with Parser_Types,
     File_Handler;

separate (Lexical_Analyzer)
procedure Obtain_Next_Token

  ( File                                    -- Sequential text file.
      : in out Parser_Types.FILE;

    Token
      : out Parser_Types.TOKEN_TYPE;

    Position                                -- Column position of the
      : in Parser_Types.COL_NUM_TYPE        -- beginning of the next
        := 0 ) is                           -- token.

--| Notes (none)
--| Modifications
--| 7/4/85    Rebecca DeMornay   Initial version of the subunit
--| 9/6/85    R. DeMornay        Added the local function
--|                              "Increment_Line_Counter".

  type LINE_COUNT is                        -- A count of the number
    range 1 .. File_Handler.Max_Size;       -- of lines in a file.

  Line_Counter
    : LINE_COUNT := 1;

  --  ..................................
  --  .                                .
  --  .    Increment_Line_Counter      .    SPEC & BODY
  --  .                                .
  --  ..................................

  function Increment_Line_Counter

    ( File                                  -- Sequential text file.
        : Parser_Types.FILE;

      Line                                  -- Line number in "File"
        : LINE_COUNT )                      -- at the time of call

    return LINE_COUNT is
```

```
   --| Purpose
   --| This function increments the line counter from the point at
   --| which it was after the last call of this routine.
   --|
   --| Exceptions
   --| Source_File_Not_Open  --  Raised if "File" is not open.
   --| End_Of_File           --  Raised if the function is called and
   --|                           the end of the file has already been
   --|                           reached.
   --|
   --| Notes (none)

   begin    -- Increment_Line_Counter

     ...

   end Increment_Line_Counter;


begin    -- Obtain_Next_Token

   ...

exception

   when File_Handler.FILE_ERROR =>
     Token := Parser_Types.NONE;
     raise Source_File_Not_Open;

end Obtain_Next_Token;
```

CHAPTER 7

PACKAGES

7.1 STRUCTURE

7S1 Use

(a)* A package should fulfill one or more of the following:

- model an abstract entity (or data type) appropriate to the domain of a problem. [NW] [Booch]

- collect related type and object declarations which are used together (this kind of package should generally be used only to provide a common set of declarations for two or more library units) [NW] [Booch]

- to group together program units for essential configuration control or visibility reasons (packages fulfilling this role alone should be used sparingly)

The roles above are listed in order of decreasing desirability. The first role, modeling a problem domain entity, is the strongest use of packages for structuring a program. It corresponds to the requirement of functional cohesion for subprograms (guideline 6S1) and contributes to the goal of making the structure of a program reflect the structure of its problem domain.

The second kind of package, a collection of related declarations, should generally be used only to provide a common set of declarations for two or more library units. Further, it is better to minimize the declaration of variables in these packages. Overuse of packages of variables results in a FORTRAN COMMON block style program decomposition which defeats the abstraction and information hiding properties of packages (see also guideline 7C2).

Finally the last type of package, a grouping of units for configuration reasons, should be used sparingly since it gives no additional information to a human reader on the structure of the program. This type of package might, for example, be used to divide a large program at the top level into subsystems to be

developed by separate teams. It would be best, however, if these subsystem packages fufilled, in addition, one of the other two roles.

(b)* Packages should NOT be designed based on the procedural structure of the code which calls them.

For example, a group of procedures should not be packaged simply because they are all called at system initialization, or because they are always called in a certain sequence. Such a package is closely coupled to the context in which it is used and is not very understandable, reusable or maintainable as a unit.

(c)* A logical hierarchy of packages should be used to reflect or model levels of abstraction. [NW]

7S2 Nesting

(a)* Nested package bodies should be separate subunits.

(b) Subprogram bodies within a package should generally be separate subunits (when this is possible).

(c) Packages should generally not be nested within subprograms, except within the main procedure.

A possible exception to this recommendation is when a package has objects of variable size which can be allocated when a procedure is called. For example, suppose some data processing uses a Buffer package which implements a buffer area of a user-specified size:

```
procedure Process_Data
  ( Buffer_Size : POSITIVE ) is

...

  package body Buffer is

    type BUFFER_TYPE is
      array (INTEGER range <>) of DATUM;

    Buffer_Area : BUFFER_TYPE (1..Buffer_Size);

    ...

  end Buffer;

...
```

Note, however, that the nested package cannot be reused outside the context of the procedure. An alternative would be to allocate the buffer using an access type. This would require careful handling of allocation and deallocation, but would result in a more self-contained package.

(d) Nesting of a package specification inside another package specification should generally be avoided.

When a package provides a good abstraction, it hides the details of its implementation. Generally, nesting one package specification inside another either exposes too much of the internal details of the outer package, or indicates that the outer package does not provide a good abstraction in the first place. It is usually better to nest the package specification within the body of the outer package. Certain of the inner package operations can then be called on by outer package operations which are at the approriate level of abstraction for the outer package.

7.2 CODING

7C1 Initialization

(a) Calls from the initialization statements of a package to subprograms outside the package should be avoided. [NW]

7C2 Visible Variables

(a) Variable declarations in package specifications should be minimized.

The use of variables in a package specification generally reduces the abstraction and information hiding properties of that package. For example, a variable cannot provide protection against being changed by units other than the package. Therefore it is generally better to use a function rather than a variable to read data from a package. It is also generally better to use a procedure rather than a variable to give data to a package, since a variable cannot trigger any package operations and a variable declaration often exposes some internal data representation details of the package.

(a)* The private part of a package specification should only be used to supply the full definitions of private types and deferred constants; all other declarations should be put in the package body. [NW]

(b) Objects of private type should be initialized by default, if possible. [NW]

7.3 FORMAT

7F1 Package Names

(a)* A package name should be a noun phrase describing the abstract entity modeled by the package, or simply whatever is being packaged.

```
Stack_Handler
Vehicle_Controller
Terminal_Operations
Parser_Types
Utilities_Package
```

7F2 Package Header

(a)* Each package specification, body or stub should be preceded by a header comment block containing at least the package name and the indication SPEC, BODY, STUB or SUBUNIT.

```
--       ****************************
--       *                          *
--       *    Lexical_Analyzer      *    SPEC
--       *                          *
--       ****************************
```

7F3 Package Specifications

(a)* Package specifications should have the following format:

```
package <package identifier> is

--| <documentary comments>

  <declaration>
  <declaration>

private  -- <package identifier>

  <declaration>
  <declaration>

end <package identifier>;
```

The <documentary comments> should follow guideline (b) below. Note that the <package identifier> should always be repeated at the "end" of the package specifications.

(b)* A package specification should include AT LEAST the following documentation immediately after the package header:

```
--| Purpose
--| A description of the purpose and function of the package.
--|
--| Initialization Exceptions
--| A list of all exceptions which may propagate out of the
--| package INITIALIZATION PART and a description of when each
--| would be raised.
--|
--| Notes
--| Additional comments on the use of the package.
```

The "Initialization Exceptions" and "Notes" headers should be included even if these sections are empty. An empty section may be indicated by placing the annotation "(none)" after the appropriate header. Only in the case of a package specification which is a compilation unit, the following section should be added to the documentation:

```
--| Modifications
--| A list of modifications made to the package SPECIFICATION.
--| Each entry in the list should include the date of the change,
--| the name of the person who made the change and a description
--| of the modification. The description should indicate exactly
--| where in the compilation unit that the change was made. The
--| first entry in the list should always be the initial coding
--| of the package specification.
```

(c) In a declarative part, all package specifications should appear before any package or task bodies.

7F4 Package Bodies And Stubs

(a)* Package bodies should have the following format:

```
separate (<parent name>)
package body <package identifier> is

--| <documentary comments>

  <declaration>
  <declaration>

begin  -- <package identifier>

  <statement>
  <statement>

exception
  when <exceptions> =>
    <statement>

end <package identifier>;
```

The <documentary comments> should follow guideline (b) below. Note that the <package identifier> should always be repeated at the "end" of the package body.

(b)* A package body should have at least the following documentation placed immediately after the package header:

```
--| Notes
--| Comments on the design, implementation and use of the
--| package.
```

The "Notes" header should be included even if the section is empty. An empty section may be indicated by the comment "Notes (none)". Only in the case of a package specification which is a compilation unit, the following section should be added to the documentation:

```
--| Modifications
--| A list of modifications made to the package BODY.  Each
--| entry in the list should include the date of the change,
--| the name of the person who made the change and a
--| description of the modification.  The description should
--| indicate exactly where in the compilation unit that the
--| change was made.  The first entry in the list should always
--| be the initial coding of the package body.
```

(b)* Package stubs should have the following format:

```
package body <package identifier> is separate;
```

7.4 EXAMPLES

See also example 12X3.

Example 7X1

```
--       ****************************
--       *                          *
--       *    Lexical_Analyzer      *    SPEC
--       *                          *
--       ****************************

with Basic_Types,
     Parser_Types;

package Lexical_Analyzer is

--| Purpose
--| The routines in this package read the source program, one
--| character at a time, to generate a stream of tokens.  As each
--| token is produced it is passed to the package "Parser".  The
--| legal tokens are defined in the Language Reference Manual.
--|
--| Initialization Exceptions
--| Diana_File_Non_Existent  --  Raised if the file "DIANA.ADA"
--|                              does not exist
--|
--| Notes
--| Tokens are limited to 32 characters in length.  Also, only
--| sequential text files can be operated on by the parser.
--|
--| Modifications
--| 6/14/85  Rebecca DeMornay   Initial version of spec.
--| 8/26/85  C. Royale          Added "Decode_Token" function.


  Diana_File_Non_Existent
    : exception;

  Source_File_Not_Open
    : exception;

  Illegal_Token
    : exception;
```

```
  --  ..............................
  --  .                            .
  --  .    Obtain_Next_Token       .    SPEC
  --  .                            .
  --  ..............................

  procedure Obtain_Next_Token

    (
      ...
          );

  ...

  --  ..........................
  --  .                        .
  --  .    Decode_Token        .    SPEC
  --  .                        .
  --  ..........................

  function Decode_Token

    (
      ...
          )

    return Parser_Types.TOKEN_VALUE_TYPE;

  ...

end Lexical_Analyzer;
```

Example 7X2

```
--       ****************************
--       *                          *
--       *    Lexical Analyzer      *    BODY
--       *                          *
--       ****************************

with Text_IO,
     File_Handler;

package body Lexical_Analyzer is

--| Notes
--| The package "Lexical_Analyzer" will later be changed to a task,
--| so that the "Parser" task (now a package) can make an entry
--| call to "Lexical_Analyazer" when it needs the next token.
--|
--| Modifications
--| 6/14/85  Charity Royale   Initial version of body.
--| 8/26/85  C. Royale        Added "Decode_Token" function.
--|                           Added instantiation of "Enumeration_IO".


  --       ******************
  --       *                *
  --       *    Char_IO     *    SPEC
  --       *                *
  --       ******************

  package Char_IO is
    new Text_IO.Enumeration_IO (Enum => Character);

  --| Purpose
  --| Used to read the input text file character by character.
  --|
  --| Initialization Exceptions (none)
  --| Notes (none)


  --  ..........................
  --  .                        .
  --  .    Obtain_Next_Token   .    STUB
  --  .                        .
  --  ..........................

  procedure Obtain_Next_Token

    (
      ...
          ) is separate;
```

```
  --  ......................
  --  .                    .
  --  .    Decode_Token    .    STUB
  --  .                    .
  --  ......................

  function Decode_Token

    (
      ...
          )

    return Parser_Types.TOKEN_VALUE_TYPE is separate;

  ...

begin  -- Lexical_Analyzer

  ...

exception

  when File_Handler.File_Error =>
    raise Diana_File_Non_Existent

end Lexical_Analyzer;
```

Example 7X3

```
--          **********************
--          *                    *
--          *        Disk        *     SPEC
--          *                    *
--          **********************

generic

  type SPECIFIC_DATA_TYPE is    -- The type of data to be
    ( <> );                     -- stored on disk

package Disk is

--| Purpose
--| This package defines an abstract data type to simplify
--| the I/O interface to disk files.
--|
--| Initialization Exceptions (none)
--| Notes (none)
--| Modifications
--| 9/10/86    Ada Users Group    Initial version
--|

  type FILE_TYPE is
    private;

  End_Of_File
    : exception;

  Open_Error
    : exception;

  Mode_Error
    : exception;

  subtype FILE_MODE is
    (IN_FILE, OUT_FILE);
```

```
-- ..................
-- .                  .
-- .    Create        .    SPEC
-- .                  .
-- ..................

function Create

  ( Name
      : STRING;

    Mode
      : FILE_MODE := IN_FILE )
  return FILE_TYPE;

--| Purpose
--| This function creates a FILE_TYPE data object to
--| represent the disk file with the given name and mode.
--|
--| Exceptions (none)
--| Notes
--| This function does not actually open the file.

-- ..................
-- .                  .
-- .     Close        .    SPEC
-- .                  .
-- ..................

procedure Close

  ( Disk_File
      : in out FILE_TYPE );

--| Purpose
--| This procedure closes a disk file if it is open.  If
--| the file is already closed it has no effect.
--|
--| Exceptions (none)
--| Notes (none)

-- ..................
-- .                  .
-- .     Read         .    SPEC
-- .                  .
-- ..................

procedure Read

  ( Disk_File
      : in out FILE_TYPE;

    Data
      : out SPECIFIC_DATA_TYPE );
```

```
--| Purpose
--| This procedure reads the next record from a file,
--| opening the file if necessary.
--|
--| Exceptions
--| End_Of_File    - raised if no more elements can be
--|                  read from the file
--| Open_Error     - if the file cannot be opened
--| Mode_Error     - if the file mode is not IN_FILE
--|
--| Notes (none)

-- .................
-- .               .
-- .      Write    .    SPEC
-- .               .
-- .................

procedure Write

  ( Disk_File
      : in out FILE_TYPE;

    Data
      : in SPECIFIC_DATA_TYPE );

--| Purpose
--| This function writes a record to a file,
--| opening the file if necessary.
--|
--| Exceptions
--| Open_Error     - if the file cannot be openned
--| Mode_Error     - if the file mode is not OUT_FILE
--|
--| Notes (none)


private   -- Disk

  --        **********************
  --        *                    *
  --        *      Disk_IO       *    SPEC
  --        *                    *
  --        **********************

  package Disk_IO is
    new Sequential_IO (SPECIFIC_DATA_TYPE);

  --| Purpose
  --| This package provides the basis for the representation
  --| of disk files.
  --|
  --| Initialization Exceptions (none)
  --| Notes (none)
```

```
    File_Name_Length
      : constant := 40;

    type FILE_TYPE is
      record
        File_Name
          : STRING(1..File_Name_Length) := (others => ' ');
        File
          : Disk_IO.FILE_TYPE;
        Mode
          : FILE_MODE := Disk_IO.IN_FILE;
      end record;

end Disk;
```

CHAPTER 8

VISIBILITY

8.1 STRUCTURE

8S1 Scope

(a)* The scope of identifiers should not extend further than necessary. Where a scope is extended by "with" clauses, these clauses should cover as small a region of text as possible. [NW]

> For example, "with" clauses should be placed only on the subunits that really need them, not on their parents. This promotes information hiding and reduces coupling. It can also result in faster recompilation (due to the dependency rules).

8S2 The Package Standard

(a)* The package **STANDARD** should not be named in a "with" clause.

8.2 CODING

8C1 The Use Clause

(a)* The "use" clause should be used only in the following cases:

- for packages of commonly known utility operations used throughout a program (eg, MATHLIB)
- to make overloaded operators visible, so that they may be used in infix notation
- for predfined input/output packages (eg, Text_IO, instantiations of Integer_IO, etc.)
- to make enumeration constants visible so that they can be named without using the dot notation

Note that even when a "use" clause is used, the dot notation should still be used in cases other than those listed above.

8C2 Renaming Declarations

(a) For a name with a large number of package qualifications, a renaming declaration may be used to define a new shorter name. The new identifier should still reflect the complete meaning of the full name.

(b) For a function which can be appropriately represented by an operator symbol name (see 6C4), a renaming declaration may be used to give it such a name.

For example a Matrix_Multiply function could be renamed "*".

8C3 Redefinition

(a)* Items from the package STANDARD should not be redefined or renamed.

(b) Redefinition of an identifier in different declarations should be avoided. [NW]

CHAPTER 9

TASKS

9.1 STRUCTURE

9S1 Use

(a)* A task should fulfill one or more of the following:

- model a concurrent abstract entity appropriate to the problem domain
- serve as an access-controlling or synchronizing agent for other tasks, or otherwise act as an interface between asynchronous tasks
- serve as an interface to asynchronous entities external to the program (eg, asynchronous I/O, devices, interrupts, etc.)
- define concurrent algorithms for faster execution on multiprocessor architectures
- perform an activity which must wait a specified time for an event or have a specific priority

[NW] [Cherry]

Just as for packages (guideline 7S1), it is best to have tasks which model problem domain entities. However, in the case of tasks it is also necessary to have some tasks which solely provide interfaces between other tasks and which handle the other issues of concurrency and parallelism mentioned above. The program should generally be structured, however, around the tasks which represent problem-domain entities.

9S2 Nesting

(a) Tasks should generally not be nested within tasks or subprograms, except for the main procedure.

Note that a subprogram containing a task cannot return until the task has terminated.

(b)* Nested task bodies should be separate subunits, unless they are quite small.

9S3 Visibility

(a) When only certain entries of a task are intended to be called by program components outside on enclosing package, it is generally preferable to hide the task specification in the package body, introducing package procedures which in turn call the actual entries. [NW]

This helps to promote information hiding and strengthens the abstraction of the enclosing package (see also guideline 7S2d). It also hides the use of tasking within the package. Note, however, that special care must be taken if the task entries are to be called using conditional or timed entry calls. In this case either the outer package must provide special procedures or procedure parameters or this guideline should not be followed.

9.2 CODING

9C1 Task Types

(a)* A task type should be used only when multiple instances of that type are required. Otherwise a directly named task should be used. [NW]

(b)* Identical tasks should be derived from a common task type. [NW]

(c)* Static task structures should be used whenever they are sufficient. Access types to task type should be used only when it is essential to create and destroy tasks dynamically, or to be able to change the names with which they are associated. [NW]

9C2 Task Termination

(a)* A task nested within the main program must terminate by reaching its "end", or must have a selective wait with a terminate alternative.

All tasks nested within program must terminate before the program can terminate. Therefore, if this guideline is not followed, it will be impossible for the main program to ever terminate other than by aborting all nested tasks. However, "abort" statements are to be avoided (see guideline 9C7).

(b)* Tasks dependent on library units should not use the "terminate" alternative of a select statement. Therefore, other provision should be made for the graceful termination of such tasks at system close down. [NW]

Tasks which are dependent on library units will not terminate due to a "terminate" alternative [RM]. Therefore a library unit task should have an entry which forces termination. If it does not, an "abort" statement in the main program may be used to terminate

the task. However, "abort" statements are to be avoided (see guideline 9C7).

9C3 Entries And Accept Statements

(a)* Only those actions should be included in the "accept" statement which must be completed before the calling task is released from its waiting state. [NW]

(b)* A task should never call its own entries, even by indirection.

This would result in a deadlock.

(c) Conditional entry calls should be used sparingly to avoid unnecessary busy waiting. [NW]

9C4 Delay Statement

(a)* A "delay" statement should be used whenever a task must wait for some known duration. A "busy wait" loop should never be used for this purpose.

It is important to remember that "delay t" provides a delay of at least t seconds, but possibly more. A program should not rely on any upper bound for this delay, especially when tasks are used (since tasks must compete for CPU time). The following example shows how to alleviate this problem in a periodic activity:

```
...
Next_Time := Calendar.Clock + Required_Period;

Periodic_Activity:
while Still_Time loop

  -- Perform activity
  ...

  -- Correct for delay statement incertitude

  Period := Next_Time - Calendar.Clock;

  if Period < 0.0 then              -- Processing was too slow
    Next_Time := Calendar.Clock     -- Avoid cumulative effect
  end if;
  Next_Time := Next_Time + Required_Period;

  delay Period;

end loop Periodic_Activity;
```

(b) The "delay" statement should normally only be used to manage interaction with some external process which works in real time, or to create a task which behaves in a well-defined manner in real time. [NW]

9C5 Task Synchronization

(a)* Knowledge of the execution pattern of tasks (eg, fixed, known time pattern, etc.) should not be used to avoid the use of explicit task synchronization. [NW]

9C6 Priorities

(a) Only a small number of priority levels should be used. The priority levels used should be spread over the range made available to type PRIORITY in the implementation. Names should be given to the priority levels by declaring constants of predefined type PRIORITY and grouping these declarations into a single package.

Using only a small number of priority levels makes the interaction of the various prioritized tasks easier to understand. On the other hand, spreading the levels across the available range allows easy insertion of a new level between existing levels if this later becomes necessary. As with other literal numbers, the use of names is more readable than the use of the literals. Further, for priorities, the allowable range of levels is implementation dependent. Naming priority levels by constant declarations grouped into a single package restricts the implementation dependency to that package. For example:

```
with System;
package Priority_Levels is

  Lowest
    : constant System.PRIORITY := System.PRIORITY'first;
  Highest
    : constant System.PRIORITY := System.PRIORITY'last;
  Number
    : constant := Highest - Lowest + 1;
  Average
    : constant System.PRIORITY := Number / 2;

  Idle
    : constant System.PRIORITY := Lowest;
  Background
    : constant System.PRIORITY := Average - 20;
  User
    : constant System.PRIORITY := Average - 10;
  Foreground
    : constant System.PRIORITY := Average + 10;

end Priority_Levels;
```

(b) For any group of related tasks, such as those declared within the same program unit, priorities should be specified either for all, or for none of them. [NW]

This avoids confusion about the scheduling of tasks with undefined priorities.

9C7 Abort Statements

(a) Abortion of tasks should generally be avoided.

Aborting a task can produce unpredictable results. In particular, do not assume anything ahout the moment at which an aborted task becomes terminated [NW]. The "abort" statement should generally be used only in case of unrecoverable failure.

9C8 Shared Variables

(a)* Tasks should not directly share variables unless only one of them can possibly be running at any one time.

(b)* Any task which uses shared variables should identify in its documentary comments all the shared variables that it uses.

9C9 Local Exception Handling

(a) To allow the handling of local exceptions without task termination, a task should generally have a block statement with an exception handler coded within its main loop.

```
begin  -- Some_Task

  Main_Loop:
  loop

    Local:
    begin

      -- Task code
      ...

    exception  -- Local
      ... handle local exceptions ...

    end Local;

  end Main_Loop;

exception
  ... handle fatal exceptions ...

end Some_Task;
```

9.3 FORMAT

9F1 Task And Entry Names

(a)* A task name should be a noun phrase describing the task function or abstract entity modeled by the task.

```
Sensor_Interface
Status_Monitor
```

```
Event_Handler
Message_Buffer
```

(b)* Entry names should follow the same guidelines as for procedure names (guideline 6F1).

9F2 Task And Entry Headers

(a)* Each task or task type specification or body and each entry specification should be preceded by a header comment block containing at least the unit name and the indication SPEC, BODY or STUB.

```
--       **********************
--       *                    *
--       *       Buffer       *    SPEC
--       *                    *
--       **********************

task Buffer is

  -- .....................
  -- .                   .
  -- .       Read        .    SPEC
  -- .                   .
  -- .....................

  entry Read
```

9F3 Task Specifications

(a)* Task specifications should have the following format:

```
task <task identifier> is

--| <documentary comments>

  <declaration>
  <declaration>

end <task identifier>;
```

The <documentary comments> should be AT LEAST as required for a procedure declaration (guideline 6F3), except for the "Exceptions" section. Note that the <task identifier> should always be repeated at the "end" of the task specification.

(b)* A task type specification should be formatted the same as a task specification, with the exception of including "task type" in the header.

(c)* Entry declarations should have the following format:

```
entry <entry identifier> (<family range>)
  ( <parameter specification>;
    <parameter specification> );

--| <documentary comments>
```

Each <parameter specification> should be formatted like an object declaration (guideline 3F4). The <documentary comments> should be AT LEAST as required for a procedure declaration (guideline 6F3).

(d)* Parameter mode indications should always be used in entry declarations.

(e) In a declarative part, all task specifications should appear before any task or package bodies.

9F4 Task Bodies And Stubs

(a)* Task bodies should have the following format:

```
separate (<parent>)
task body <task identifier> is

--| <documentary comments>

  <declaration>
  <declaration>

begin  -- <task identifier>

  <statement>
  <statement>

exception
  when <exceptions> =>
    <statement>

end <task identifier>;
```

The <documentary comments> should be AT LEAST as required for a procedure body (guideline 6F4). Note that the <task identifier> should always be repeated at the "end" of the task body.

(c)* Task stubs snould have the following format:

```
task body <task identifier> is separate;
```

9F5 Accept Statements

(a)* "Accept" statements should have one of the following formats:

```
accept <entry identifier> (<entry index>);

accept <entry identifier> (<entry index>)
  ( <parameter specification>;
    <parameter specification> )
do
  <statement>
  <statement>
end <entry identifier>;
```

Each <parameter specification> should be formatted like an object declaration (guideline 3F4). Note that the <entry identifier> should always be repeated at the "end" of the "accept" (if there is an "end").

(b)* Parameter mode indications should always be used in accept statements.

9F6 Select Statements

(a)* Selective wait statements should have the following format:

```
select
  <statement>
  <statement>
or
  <statement>
  <statement>
or
  when <condition> =>
    <statement>
    <statement>
else
  <statement>
  <statement>
end select;
```

This format is consistent with the indentation style of other statements. In addition, the added level of indentation especially highlights guarded sections of code.

(b)* Conditional and timed entry calls should have the following format:

```
select
  <entry call>
  <statement>
else
  <statement>
  <statement>
end select;
```

9F7 Pragma Priority

(a) The priority pragma should appear in task specifications before any entry declarations, and in the main program before any declarations. [NW]

9.4 EXAMPLES

Example 9X1

```
--        **********************
--        *                    *
--        *       Buffer       *    SPEC
--        *                    *
--        **********************

task Buffer is

--| Purpose
--| This task provides a character buffer to smooth variations
--| between the speed of output of a producing task and the speed
--| of input of a consuming task.
--|
--| Exceptions (none)
--| Notes (none)

   -- ..................
   -- .                  .
   -- .       Read       .    SPEC
   -- .                  .
   -- ..................

   entry Read
     ( Output   : out Character);

   --| Purpose
   --| This entry reads a character from the buffer.
   --| If the buffer is empty, the entry will wait
   --| until a character is written into the buffer.
   --|
   --| Exceptions (none)
   --| Notes (none)


   -- ..................
   -- .                  .
   -- .      Write       .    SPEC
   -- .                  .
   -- ..................

   entry Write
     ( Input    : in Character);

   --| Purpose
   --| This entry writes a character into the buffer.
   --| If the buffer is full the entry will wait
   --| until a character is read from the buffer.
   --|
   --| Exceptions (none)
   --| Notes (none)
```

```
end Buffer;
```

Example 9X2

```
--        **********************
--        *                    *
--        *       Buffer       *    BODY
--        *                    *
--        **********************

separate (Buffer_Package)
task body Buffer is

--| Notes
--| This task contains an internal pool of characters processed
--| in a round-robin fashion.
--|
--| Modifications
--| 7/2/86  Fred Blah   Initial version.
--|

  Pool_Size
    : constant := 100;

  subtype POOL_RANGE is
    INTEGER range 1..Pool_Size;

  type POOL_TYPE is
    array (POOL_RANGE) of CHARACTER;

  Pool
    : POOL_TYPE;

  Count                                  -- The number of characters in
    : INTEGER range 0..Pool_Size         -- the pool.
      := 0;

  In_Index                               -- The space for the next input
    : POOL_RANGE := 1;                   -- character.

  Out_Index                              -- The space for the next output
    : POOL_RANGE := 1;                   -- character.
```

```
begin   -- Buffer

  loop
    select
      when Count < Pool_Size =>
        accept Write
          ( Input  : in Character )
        do
          Pool(In_Index) := Input;
        end Write;

        In_Index := In_Index mod Pool_Size + 1;
        Count := Count + 1;

    or
      when Count > 0 =>
        accept Read
          ( Output : out Character)
        do
          Output := Pool(Out_Index);
        end Read;

        Out_Index := Out_Index mod Pool_Size + 1;
        Count := Count - 1;

    or
      terminate;

    end select;

  end loop;

end Buffer;
```

Example 9X3

```
-- ..................
-- .                  .
-- .    Shellsort     .    BODY
-- .                  .
-- ..................

procedure Shellsort

  ( List                    -- This list will be sorted
     : in out ITEM_LIST;    -- in place.

    Number_Of_Items
     : in NATURAL );

--| Notes
--| This sorting procedure implements the Shell sort by
--| seperating the n-sorts into multiple Ada tasks.
--| This algorithm is designed for parallel processing
--| of the tasks and is not necessarily an efficient
--| method on a single processor.
--|
--| Modifications
--| 9/5/86   A. Shell        Initial version
--|

  Increment                 -- Increment of an n-sort
    : NATURAL;

  Number_Of_Sorts           -- Number of paralled sorts
    : NATURAL;              -- for a single pass

  Number_Of_Tasks
    : NATURAL;

  --        **********************
  --        *                    *
  --        *    SORTER_TASK     *    SPEC
  --        *                    *
  --        **********************

  task type SORTER_TASK is

  --| Purpose
  --| Tasks of this type perform the n-sort for the Shell sort.
  --|
  --| Notes
  --| A SORTER_TASK terminates itself when it is no longer
  --| needed for the sort.
```

```
   -- ...................
   -- .                 .
   -- .      Sort       .    SPEC
   -- .                 .
   -- ...................

   entry Sort
     ( First : in INTEGER;
       Step  : in NATURAL );

   --| Purpose
   --| This entry signals a sorter task to perform a new
   --| n-sort.  Elements are sorted in place in List,
   --| starting with the element at index First and
   --| including subsequent elements at the indicated Step.
   --|
   --| Exceptions (none)
   --| Notes (none)

 end SORTER_TASK;

 --         **********************
 --         *                    *
 --         *    SORTER_TASK     *    STUB
 --         *                    *
 --         **********************

 task body SORTER_TASK is separate;

 type SORTER_ARRAY is
   array (INTEGER range <>) of SORTER_TASK;

begin   -- Shellsort

 if Number_Of_Items < 2 then
   return;
 end if;

 -- Determine the first n-sort increment.

 Increment := 1

 while Increment < Number_Of_Items
   Increment := 3*Increment + 1;
 end loop;

 Increment := Increment / 3;
 if Increment < 1 then
   Increment := 1;
 end if;

 -- Determine the number of tasks required to perform
 -- the sort.

 if Number_Of_Items / Increment = 1 then
```

```
      Number_Of_Sorts := Number_Of_Items mod Increment;
      if Increment/3 > Number_Of_Sorts then
        Number_Of_Tasks := Increment / 3;
      else
        Number_Of_Tasks := Number_Of_Sorts;
      end if;

    else
      Number_Of_Sorts := Increment;
      Number_Of_Tasks := Number_Of_Sorts;

    end if;

    -- Perform the sort

    Task_Block:
    declare

      Sort_List
        : SORTER_ARRAY (1 .. Number_Of_Tasks);

    begin

      while Increment > 0 loop

        for K in 1 .. Number_Of_Sorts loop

          Sort_List(K).Sort
            ( First    => List'first + K - 1,
              Step     => Increment );

        end loop;

        Increment := Increment / 3;
        Number_Of_Sorts := Increment;

      end loop;

    end Task_Block;

end Shellsort;
```

Example 9X4

```
--        **********************
--        *                    *
--        *    SORTER_TASK     *    SUBUNIT
--        *                    *
--        **********************

separate (Shellsort)
task body SORTER_TASK is

--| Notes
--| This task body implements a task type.
--|
--| Modifications
--| 9/5/86    A. Shell        Initial version
--|

-- Global variables
-- List              --  An array of all items to be
--                       sorted by Shellsort.
-- Number_Of_Items   --  The number of items in the list.

  Start         : INTEGER;
  Increment     : NATURAL;

  A             : INTEGER;
  B             : INTEGER;
  First_B       : INTEGER;

  Temp          : INTEGER;

begin   -- SORTER_TASK

  loop

    accept Sort
      ( First     : in INTEGER;
        Step      : in NATURAL )
    do
      Start := First;
      Increment := Step;
    end Sort;

    First_B := Start + Increment;
    while First_B <= List'first + Number_Of_Items - 1 loop

      B := First_B;
      A := B - Increment;

      Find_Position:
      while A >= Start loop
        exit when not (List(A) > List(B));
```

```
            Temp := List(A);
            List(A) := List(B);
            List(B) := Temp;

            B := A;
            A := B - Increment;

          end loop Find_Position;

          First_B := First_B + Increment;

        end loop;

        -- Terminate if task is not needed for n-sort
        exit when Increment/3 < Start - List'first + 1;

      end loop;

    end SORTER_TASK;
```

CHAPTER 10

PROGRAM STRUCTURE AND COMPILATION ISSUES

10.1 STRUCTURE

10S1 Program Units

(a)* Library units should be used in the following cases:

- to allow configuration control of the high level functional subsystems of a program

- for general purpose, reusable program units

[ACGE]

(b)* Nested program units should be used in the following cases:

- to allow direct access to objects declared in an enclosing scope

- to increase the structural hiding of the internal implementation details of an enclosing program unit

[ACGE]

(c)* Bodies of nested program units should be made separate unless they are small enough not to effect the readability of the enclosing unit.

(d) Library units which are packages are generally preferable over library units which are subprograms. Library units providing services to the main program should be packages.

10S2 With Clauses

(a)* No unit should have a "with" clause for a unit it does not need to see directly.

(b) If only a small part of a given unit needs access to a library unit, then it should generally appear as a subunit and have its own "with" clause for that library unit (see also guideline 8S1) [NW].

10S3 Program Unit Dependencies

(a) Excessive dependencies between compilation units should be avoided, especially the use of complicated networks of "with" clauses.

(b) It is preferable to limit program unit dependencies to a tree structure whenever possible [NW].

10.2 FORMAT

10F1 Compilation Units

(a)* Each compilation unit should be in a separate file, except possibly in the case of a generic procedure specification and its body.

CHAPTER 11

EXCEPTIONS

11.1 STRUCTURE

11S1 Exception Propagation

(a) Exceptions propagated by a program unit should be considered part of the abstraction or function represented by that unit. Therefore, it should generally only propagate exceptions which are appropriate to that level of abstraction. If necessary, an exception which cannot be handled by a unit at one level of abstraction should be converted into an exception which can be explicitly recognized by the next higher level.

> For example, a Stack package, should provide a Stack_Full exception instead of propagating a Constraint_Error. Similarly, a Matrix_Inverse function should raise a Matrix_Is_Singular exception rather than propagating Numeric_Error.

11.2 CODING

11C1 Use

(a)* An exception should be used only for one or more of the following reasons:

- it reports an irregular event which is outside the normal operation of a program unit or is in some sense an error
- it is used where it can be argued that it is safer (more defensive) than the alternative, in particular to guard against omissions of error checking code for especially harmful errors
- it reports an event for which it is inconvenient or unnatural to test at the point of cause/occurence and thus use of the exception enhances readability.

> Exceptions declared in package specifications are really part of the abstraction defined by that package. Therefore their use should be integral to the design of the package (see also guideline 10S1).

Also, note that the predefined exceptions should be used with care. Due to allowable implementation differences, they should not be relied upon to indicate particular circumstances. [NW]

(b)* Exceptions should not be used as a means of returning normal state information. [NW]

For example, a Stack package may have Stack_Full and Stack_Empty exceptions which are raised by its Push and Pop subprograms. However, these subprograms should NOT be used solely to raise exceptions to test if the appropriate conditions are true. Instead, the package should provide BOOLEAN functions such as Full and Empty to test for these state conditions.

11C2 Exception Handlers

(a)* The exception handler choice "others" should be used only if it is necessary to ensure that no UNANTICIPATED exception can be propagated or if some special action must be taken before propagation.

For example important tasks should generally have an "others" clause in a local exception handler (see guideline 9C9) to prevent them from terminating due to unanticipated exceptions. However, in the case when it can be expected that a certain exception may sometimes occur, than that exception should always be explicitly named in the exception handler.

(b)* Recursion should not be used within an exception handler.

(c) Exception handlers on block statements should be used sparingly.

One of the advantages of using exceptions is that it separates the error handling code from the more often executed normal-processing code. Excessive use of exception handlers in block statements can defeat this advantage.

11C3 Raise Statements

(a)* Exceptions declared in the specification of a package which represents a problem domain entity should not be raised outside that package.

Exceptions declared in a package specification should be considered part of the abstraction defined by that package. These exceptions provide special "signals" from the package operations, and should thus not be raised outside of the package.

(b)* Exceptions raised within a task should always be handled within that task.

Note that in the case of an exception raised during a rendezvous the exception will also be propagated back to the point of the entry call.

(c) The predefined exceptions should generally not be explicitly raised.

11C4 Exception Propagation

(a)* Exceptions should not be allowed to propagate outside their own scope. [NW]

An exception may be allowed to propagate to any point where it can be named in an exception handler. Note that this includes the case where an exception is defined in a package specification and has its scope "expanded" by a "with" clause. What must be avoided are cases such as the following:

```
...
procedure Raise_Exception is
  Hidden_Exception  : exception;
begin
  raise Hidden_Exception;
end Raise_Exception;

begin
  Raise_Exception;
  -- "Hidden_Exception" CANNOT be named at this point
  ...
```

11C5 Suppressing Checks

(a)* Checks should not be suppressed except for essential efficiency or timing reasons in thoroughly tested program units.

11.3 FORMAT

11F1 Exception Declarations

(a)* Exception declarations should be formatted like object declarations (guideline 3F4).

11.4 EXAMPLES

See examples 5X4, 6X5, 7X2 and 14X2.

CHAPTER 12

GENERIC UNITS

12.1 STRUCTURE

12S1 Use

(a)* Generics should not be used in situations in which normal programming constructs are equivalent. [NW]

(b)* A generic program unit should fulfill one or more of the following:

- provide logically equivalent operations on objects of different type
- parameterize a program unit by a subprogram value
- provide a data abstraction required at many points in a program, even if no parameterization is required [NW]
- provide parameters which are particularly appropriate to be fixed at declaration or elaboration time.

12S2 Generic Library Units

(a) Generic units should generally be library units. [NW]

12S3 Generic Instantiation

(a) The most commonly used generic instantiations should generally be placed in library units.

(b) Generic instantiations should be used cautiously within generic units.

12.2 CODING

12C1 Generic Formal Subprograms

(a)* The actual subprograms associated with the formal subprogram parameters of a generic unit should be consistent with the conceptual meanings of the formal parameters (e.g., only functions which are conceptually "adding operations" should be associated with a formal parameter named "plus"). [NW]

(b) Operator symbol function generic parameters should generally be provided with a box default body ("is <>").

```
with function "<"
  ( X : ITEM;
    Y : ITEM )
  return BOOLEAN is <>;
```

12C2 Use Of Attributes

(a) In writing generic bodies, attributes should be used as much as possible to generalize the code produced. [NW]

12.3 FORMAT

12F1 Generic Declarations

(a)* Generic declarations should have the following format:

```
generic
  <declaration>
  <declaration>
<program unit specification>;

--| <documentary comments>
```

Each <declaration> should be formatted like its non-formal counterpart (guidelines 3F3 and 3F4), except for formal subprograms which should be formatted as in (b) below. The <program unit specification> should be formatted as for non-generic units (guidelines 6F3 and 7F3).

(b)* A generic formal parameter subprogram declaration should have one of the following formats:

```
with <subprogram specification>;
--| <purpose>

with <subprogram specification> is <>;
--| <purpose>

with <subprogram specification> is <default name>;
--| <purpose>
```

The <subprogram specification> should be formatted as for a subprogram declaration (guideline 6F3). However, generally the only documentation needed on formal subprograms is the "Purpose".

(c)* A generic declaration should be preceeded by the appropriate unit header block (guidelines 6F2 and 7F2).

12F2 Generic Instantiations

(a)* Generic instantiations should have one of the following formats:

```
    <unit header> is
      new <generic name> (<generic argument>, <generic argument>);

    --| <documentary comments>
```

or:

```
    <unit header> is
      new <generic name>
        ( <generic parameter> => <generic argument>,
          <generic parameter> => <generic argument> );

    --| <documentary comments>
```

Note that in the second form the arrows ("=>") should be kept aligned. The <documentary comments> should be the same as those required for a specification of the appropriate kind of unit (guidelines 6F3 and 7F3).

(b)* Generic instantiations should have the same kind of header comment block as for a specification of the appropriate kind of unit (guidelines 6F2 and 7F2).

12.4 EXAMPLES

See also examples 7X2 and 7X3.

Example 12X1

```
-- .....................
-- .                     .
-- .    Shellsort        .    SPEC
-- .                     .
-- .....................

generic

  type ITEM is
    private;                        -- The type of items sorted

  type ITEM_LIST is                 -- The type of the item list
    array (INTEGER range <>) of ITEM;

  with function "<"
    ( Left  : ITEM;
      Right : ITEM )
    return BOOLEAN is <>;

  --| Purpose
  --| This function defines the ordering used when the
  --| items are sorted.

procedure Shellsort

  ( List                            -- This list will be sorted
      : in out LIST_TYPE;           -- in place.

    N                               -- The number of items in
      : in NATURAL );               -- the list.

--| Purpose
--| This procedure sorts the items in List using a Shell
--| sort algorithm designed for parallel processing.
--|
--| Exceptions (none)
--| Notes
--| (This is a generic declaration for the procedure
--|  body given in example 9X3.)
--|
--| Modifications
--| 9/5/86    A. Shell          Initial version
--|
```

Example 12X2

```
-- ....................
-- .                  .
-- .    Name_Sort     .    SPEC
-- .                  .
-- ....................

procedure Name_Sort is
  new Shellsort
    ( ITEM       => NAME,
      LIST_TYPE  => NAME_LIST );
```

Example 12X3

```
--        *************************
--        *                       *
--        *    Unit_Statistics    *   SPEC
--        *                       *
--        *************************

with TEXT_IO;
generic

  Unit_Name                    -- Name of the unit for which
    : STRING;                  -- statistics are to be kept.

  type ELEMENT_TYPE is         -- Enumeration type of the elements
    (<>);                      -- to be counted.

package Unit_Statistics is

--| Purpose
--| This package provides operations to keep counts for the
--| various elements of a program unit.  These counts can be
--| incremented or printed out in a report.
--|
--| Initialization Exceptions (none)
--| Notes
--| This package is based on the generic package "Task_Statistics"
--| written by Dan Roy.
--|
--| Modifications
--| 8/18/86   Ed Seidewitz    Initial version

  -- .........................
  -- .                       .
  -- .    Number_Of_Lines    .    SPEC
  -- .                       .
  -- .........................

  function Number_Of_Lines
    return POSITIVE;

  --| Purpose
  --| This function returns the number of lines printed by
  --| procedure Report since the last call to Number_Of_Lines.
  --|
  --| Exceptions (none)
  --| Notes (none)
```

```
-- ...................
-- .                 .
-- .    Count_Of     .    SPEC
-- .                 .
-- ...................

function Count_Of
  ( Element
      : ELEMENT_TYPE )
  return NATURAL;

--| Purpose
--| This function returns the current count for the specified
--| element.
--|
--| Exceptions (none)
--| Notes (none)

-- ....................
-- .                  .
-- .    Increment     .    SPEC
-- .                  .
-- ....................

procedure Increment
  ( Element
      : in ELEMENT_TYPE;
    By_Amount
      : in INTEGER := 1 );

--| Purpose
--| This procedure increments the count for the specified
--| element by a certain amount.  By default, this amount
--| is one.
--|
--| Exceptions (none)
--| Notes (none)
```

```
  -- .................
  -- .               .
  -- .    Report     .    SPEC
  -- .               .
  -- .................

  procedure Report
    ( Report_File
        : Text_IO.FILE_TYPE );

  --| Purpose
  --| This procedure prints a report of all statistics for
  --| this unit to the specified text file.
  --|
  --| Exceptions (none)
  --| Notes (none)

end Unit_Statistics
```

Example 12X4

```
--        ************************************
--        *                                  *
--        *    Telemetry_Reader_Statistics   *    SPEC
--        *                                  *
--        ************************************

package Telemetry_Reader_Statistics is
  new Unit_Statistics
    ( Unit_Name       => "Telemetry_Reader",
      ELEMENT_TYPE    => READER_ELEMENTS );

--| Purpose
--| This package collects statistics on elements of the
--| Telemetry_Reader.
--|
--| Initialization Exceptions (none)
--| Notes (none)
```

CHAPTER 13

REPRESENTATION CLAUSES AND IMPLEMENTATION-DEPENDENT FEATURES

13.1 STRUCTURE

13S1 Encapsulation

(a)* Representation clauses and implementation dependent features should, if possible be hidden inside packages which present implementation independent interfaces to users. [NW]

13.2 CODING

13C1 Use

(a)* Machine dependent and low-level Ada features should not be used except when absolutely necessary.

(b)* Representation clauses and implementation-dependent features should only be used for one of the following:

- to increase efficiency (when absolutely necessary)

- for interrupt handling

- for interfacing to hardware, foreign code or foreign data

- to specify task storage size

Further, address clauses should be used with entries only to associate them with hardware interrupts.

(c)* Representation clauses should not be used to change the meaning of a program. [NW]

13C2 Interrupts

(a) Interrupt routines should be kept as short as possible. [NW]

13.3 FORMAT

13F1 Representaion Clauses

(a)* Representation clauses should be placed near to the objects they affect.

CHAPTER 14

INPUT-OUTPUT

14.1 STRUCTURE

14S1 Encapsulation

(a)* Use of the Low_Level_IO procedures should always be encapsulated in packages or tasks.

(b) Use of the Low_Level_IO procedures should generally be encapsulated in task objects associated with each item of controlled equipment. [NW]

(c) File management and textual input-output software should generally be encapsulated in specialized packages with simple interfaces. [NW]

> This should include file interface code, textual formatting code and user interface code. User interface encapsulation can be especially useful when a system must accomodate increasing levels of user interface sophistication or changing user needs over its lifetime. In these cases it is crucial that details of the implementation of the user interface be hidden so that changes can be made to it without affecting the rest of the system.

14.2 CODING

14C1 Text Formatting

(a)* Line and page formatting should be done using the New_Line and New_Page subprograms, rather than explicitly writing end-of-line or end-of-page characters.

14C2 Low-Level Input-Output

(a)* Use of package Low_Level_IO should be avoided unless absolutely necessary.

14C3 Form Parameter

(b) Use of the Form parameter of the Open and Create procedures should generally be avoided.

The "Form" parameter on the file Open and Create procedures specifies system-dependent file characteristics. This can reduce both readability and portability, and so should only be used if absolutely necessary.

14.3 EXAMPLES

See also examples 5X3, 5X4 and 7X3.

Example 14X1

```
-- ................
-- .              .
-- .    Report    .    SUBUNIT
-- .              .
-- ................

separate (Unit_Statistics)
procedure Report is
  ( Report_File
      : in Text_IO.FILE_TYPE ) is

--| Notes
--| This example is based on Task_Statistics.Report
--| by Dan Roy.
--|
--| Modifications
--| 8/18/86   Ed Seidewitz     Initial version

  Unit_Name_Column
    : constant := 10;

  Value_Column
    : constant := 40;

  use Text_IO;          -- For output operations.

begin   -- Report

  -- Print header
  New_Line (Report_File);
  Set_Col (Report_File, To => Unit_Name_Column);
  Put_Line (Report_File,
      "Statistics for " & STRING(Unit_Name));
  New_Line (Report_File);
  Number_Lines_Printed := Number_Lines_Printed + 2;

  -- Print "element name    element value" for all elements
  for Element in Statistics_Array'range loop
    Put (Report_File, ELEMENT_TYPE'image (Element));
    Set_Col (Report_File, To => Value_Column);
    Put_Line (Report_File,
        INTEGER'image (Statistics_Array(Element)));
    Number_Lines_Printed := Number_Lines_Printed + 1;
  end loop;

end Report;
```

Example 14X2

```
-- ..................
-- .                  .
-- .      Read        .     SUBUNIT
-- .                  .
-- ..................

separate (Disk)
procedure Read

  ( Disk_File
      : in out FILE_TYPE;

    Data
      : out SPECIFIC_DATA_TYPE ) is

--| Notes
--| (This is the body of procedure Read in example 7X3)
--|
--| Modifications
--| 9/10/86    Ada User's Group        Initial version
--|

begin   -- Read

  if not Disk_IO.Is_Open(Disk_File.File) then
    Open_File(Disk_File);
  end if;

  Disk_IO.Read
    ( File  => Disk_File.File,
      Item  => Data );

exception

  when Disk_IO.End_Error =>
    Disk_Io.Close (Disk_File.File);
    raise End_Of_File;

  when Disk_IO.Name_Error | Disk_IO.Use_Error =>
    raise Open_Error;

  when Disk_IO.Mode_Error =>
    raise Mode_Error;

end Read;
```

REFERENCES

[ACGE] Ausnit, Cohen, Goodenough and Fanes. Ada in Practice. Springer Verlag, 1985.

[Booch] Booch, Grady. Software Engineering with Ada. Benjamin-Cummings, 1983.

[Cherry] Cherry, George. PAMELA. Course Notes, 1985.

[Gardner] Gardner, et al. Intellimac Ada Style Manual (2nd edition). Intellimac, June 1983.

[Myers] Myers, G. J. Reliable Systems through Composite Design. Van Nostrand, 1975.

[NW] Nissen and Wallis (ed). Portability and Style in Ada. Cambridge University Press, 1984.

[QA] Quimby and Agresti. Ada Style Guide. CSC, February 12, 1986.

[RM] Reference Manual for the Ada Programming Language. ANSI/MIL-STD-1815A-1983

[YC] Yourdon and Constantine. Structured Design. Prentice-Hall, 1979.

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-004, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977

SEL-77-005, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-78-302, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3), W. J. Decker and W. A. Taylor, July 1986

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

C-2

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve As a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1, P. Lo and D. Wyckoff, July 1983

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-106, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, May 1985

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-203, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo, June 1984

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-003, Software Engineering Laboratory (SEL) Data Base Reporting Software User's Guide and System Description, P. Lo, August 1983

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1), W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-406, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, Q. L. Jordan, and F. E. McGarry, November 1986

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

SEL-84-002, Configuration Management and Control: Policies and Procedures, Q. L. Jordan and E. Edwards, December 1984

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, Proceedings From the Ninth Annual Software Engineering Workshop, November 1984

SEL-85-001, A Comparison of Software Verification Techniques, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team, R. Murphy and M. Stark, October 1985

SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985

SEL-85-004, Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics, R. W. Selby, Jr., May 1985

SEL-85-005, Software Verification and Testing, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-85-006, Proceedings From the Tenth Annual Software Engineering Workshop, December 1985

SEL-86-001, Programmer's Handbook for Flight Dynamics Software Development, R. Wood and E. Edwards, March 1986

SEL-86-002, General Object-Oriented Software Development, E. Seidewitz and M. Stark, August 1986

SEL-86-003, Flight Dynamics System Software Development Environment Tutorial, J. Buell and P. Myers, July 1986

SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986

SEL-86-005, Measuring Software Design, D. N. Card, October 1986

SEL-86-006, Proceedings From the Eleventh Annual Software Engineering Workshop, December 1986

SEL-87-001, Product Assurance Policies and Procedures for Flight Dynamics Software Development, S. Perry et al., March 1987

SEL-87-002, Ada Style Guide (Version 1.1), E. Seidewitz et al., May 1987

SEL-RELATED LITERATURE

Agresti, W. W., Definition of Specification Measures for the Software Engineering Laboratory, Computer Sciences Corporation, CSC/TM-84/6085, June 1984

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," Proceedings of the First Pan-Pacific Computer Conference, September 1985

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?", Journal of Systems and Software, February 1981, vol. 2, no. 1

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," Proceedings of the International Computer Software and Applications Conference, October 1985

[4]Basili, V. R., and D. Patnaik, A Study on Fault Prediction and Reliability Assessment in the SEL Environment, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics, March 1981

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P--A Prototype Expert System for Software Engineering Management," Proceedings of the IEEE/MITRE Expert Systems in Government Symposium, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE Computer Society Press, 1979

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environments's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland, Technical Report TR-1501, May 1985

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," IEEE Transactions on Software Engineering, July 1986

[2]Basili, V.R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: IEEE Computer Society Press, 1978

[3]Card, D. N., "A Software Technology Evaluation Program," Annais do XVIII Congresso Nacional de Informatica, October 1985

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, February 1986

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," ACM Software Engineering Notes, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: IEEE Computer Society Press, 1983

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," Proceedings of the Hawaiian International Conference on System Sciences, January 1985

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," IEEE Transactions on Software Engineering, February 1985

[1]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (proceedings), November 1982

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

NOTES:

[1]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.

[2]This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[3]This article also appears in SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985.

[4]This article also appears in SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986.