

AD-A140 818

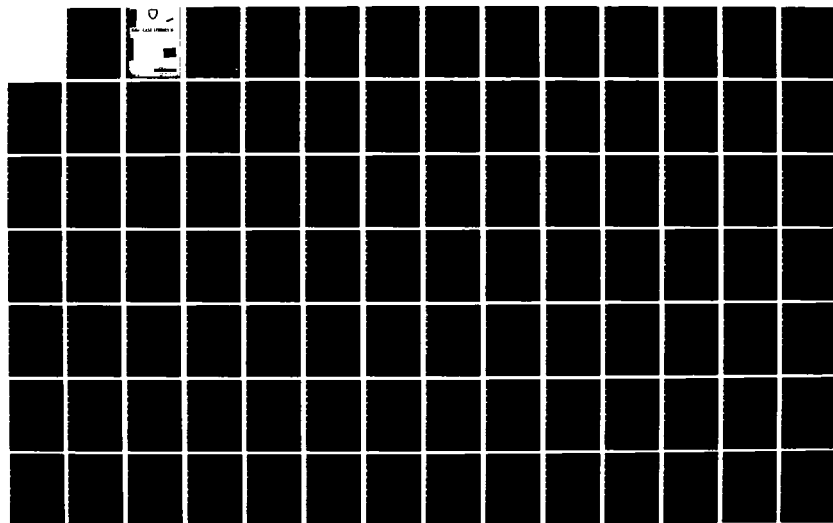
ADA (TRADEMARK) CASE STUDIES II(U) SOFTECH INC WALTHAM
NR JAN 84 DAB07-83-C-K514

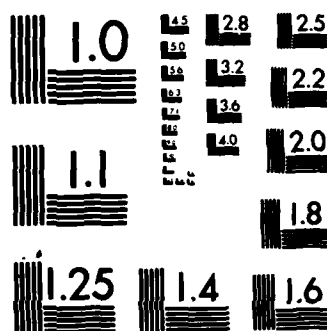
1/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

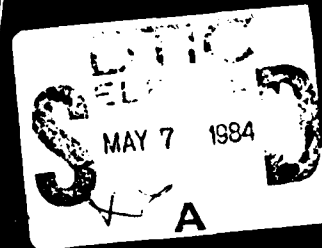
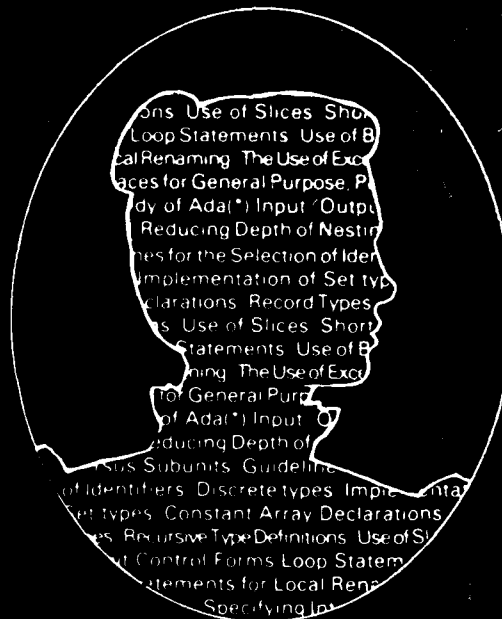
AD-A140 818



JANUARY 1984

Ada CASE STUDIES II

DTIC FILE COPY



Center For Tactical Computer Systems
(CENTACS)

U.S. Army Communications-Electronics Command
(CECOM)

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

Contract DAAB07-83-C-K514

84 04 30 046

Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

Approved For Public Release; Distribution Unlimited

(1)

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
1	INTRODUCTION	1-1
1.1	Background	1-1
1.2	Approach	1-2
1.3	Overview	1-3
2	CASE STUDIES	2-1
2.1	Naming Conventions	2-1
2.1.1	Guidelines for the Selection of Identifiers	2-3
2.2	Types	2-27
2.2.1	Discrete Types	2-29
2.2.2	Implementation of Set Types	2-41
2.2.3	Constant Array Declarations	2-53
2.2.4	Record Types	2-69
2.2.5	Recursive Type Definitions	2-79
2.3	Coding Paradigms	2-89
2.3.1	Use of Slices	2-91
2.3.2	Short Circuit Control Forms	2-97
2.3.3	Loop Statements	2-105
2.3.4	Use of Block Statements for Local Renaming	2-121
2.4	Exceptions	2-133
2.4.1	The Use of Exceptions	2-135

(1)

A

(1)

Accession For
Distribution/
Availability C. list
Avail and for
Special
List

A-1

(1)

S
D
A

DTIC
ELECTE
MAY 7 1984

2.5	Program Structure	2-179
2.5.1	Specifying Interfaces for General Purpose, Portable Software: A Study of Ada (*) Input/Output	2-181
2.5.2	Information Hiding	2-191
2.5.3	Reducing Depth of Nesting	2-201
2.5.4	Library Units Versus Subunits	2-229
3	CONCLUSIONS	3-1
3.1	Ada Life Cycle Design Methodology	3-1
3.1.1	Problem Analysis	3-3
3.1.2	Requirements Definition	3-3
3.1.3	High-Level Design	3-4
3.1.4	Low-Level Design	3-5
3.1.5	Coding	3-6
3.1.6	Unit Testing	3-7
3.1.7	Integration Testing	3-8
3.1.8	Acceptance Testing	3-8
3.1.9	Maintenance	3-9
3.2	Research Topics Addressed	3-10
3.2.1	Information Hiding - Data Abstraction	3-10
3.2.2	Postponing Design Details	3-11
3.2.3	Guidelines on Packages	3-11
3.2.4	Tutorial on Exceptions	3-12
3.2.5	Dot Notation, Naming and Readability	3-13
3.2.6	Standard Coding Style	3-13
3.2.7	Simple Coding Paradigms	3-13
3.2.8	Handling Impossible States	3-14
3.2.9	Hardware Error Detection	3-15
3.2.10	Reusable Software Modules	3-15
3.3	Areas for Future Research	3-17
3.3.1	Design Issues	3-17
3.3.1.1	Guidelines for Preserving Imple- mentation Freedom When Designing Packages	3-17
3.3.1.2	Recognizing the Opportunity for Generic Solutions	3-17

* Ada is a registered trademark of the Department of Defense (Ada Joint Program Office) OUSDRE (R&AT)

3.3.1.3	Fault-Tolerant Program Design	3-17
3.3.1.4	Introduction of Tasks During Step-wise Refinement of a Design	3-18
3.3.1.5	Alternative I/O Packages	3-18
3.3.2	Data Abstraction Issues	3-18
3.3.2.1	Specific Versus General Types	3-18
3.3.2.2	Abstract Data Types Implemented as Access Types	3-18
3.3.2.3	Non-Access "Pointer" Types	3-19
3.3.2.4	Data Structures Involving External Files	3-19
3.3.2.5	Use of Character Types	3-19
3.3.3	Additional Naming Conventions	3-20
3.3.4	Additional Coding Paradigms	3-20
3.3.4.1	When to Use <u>use</u> With <u>with</u>	3-20
3.3.4.2	Justifiable Uses of the Goto Statement	3-20
3.3.4.3	Use of Procedures for Local Renaming	3-21
3.3.4.4	Replication of Tasks	3-21
3.3.4.5	Subprograms with Variable Types of Parameters	3-21
3.3.5	Operational Issues	3-22
3.3.5.1	Implementation and Optimization Issues	3-22
3.3.5.2	Definition of Standard Functional Subsets for I/O	3-22
3.3.5.3	Identification of Ada Programming Tools	3-23

Section 1

INTRODUCTION

1.1 Background

→ This report presents a set of case studies on different aspects of the Ada language. The work which led to this volume was performed by SofTech, Inc. under the Ada Design Methods Training Support contract, (No. DAAB07-83-C-K514) for the U.S. Army Communication - Electronics Command (CECOM). This effort is a continuation of the case studies work which originated under the Ada Software Design Methods Formulation (ASDMF) contract (No. DAAK80-81-C-0187) with CECOM.

→ The objective of this report is threefold:

- to expand on Ada issues which surfaced in the ASDMF contract cited above
- to explore new Ada issues
- to gain insight into the characteristics of a life cycle design methodology that would promote effective use of Ada

This report contains fifteen case studies which illustrate different areas of the Ada language:

- naming conventions,
- types,
- coding paradigms,
- exceptions,
- program structure. ←

They provide insights into Ada usage and style, addressing both issues that arise in embedded computer systems and general programming and design practice. A substantial portion of the examples originate in the code written for a message switch by one of the contractors observed during the ASDMF contract. Many of the topics discussed in these case studies were first raised during the Technical Interchange Meetings that occurred under ASDMF. They were documented in [Sof82], some as areas for future research. The correlation between these topics and the current collection of case studies is discussed further in Section 3.2 of this report.

1.2 Approach

↙ The case studies share a common format, consisting of the following sections: background, detailed example(s), and an epilogue, where appropriate. The background section is subdivided into three parts: objective, problem, and discussion. The objective states succinctly the purpose of the case study. The problem poses the questions facing the designer or programmer from an abstract viewpoint. The discussion amplifies on the problem and its causes, motivating the specific examples given in the subsequent section. ↘

The second section of a case study presents one or more detailed examples of the problem exposed in the background section. Depending on the topic, there are variations in the actual format of its development. In the first variation, there is a single problem statement and multiple solutions. In the second variation, there are multiple examples, each having its own solution. In the third variation, the example problem statements stand alone because their solutions are beyond the scope of the case study. This variation serves as a medium for an in-depth discussion of some of the design decisions behind the Text_IO package described in Chapter 14 of [DoD83]. A solution to the limitations of the input/output specifications is clearly too complex and unconstrained an undertaking for a case study.

In the third section, the epilogue, issues related to the detailed example(s), if any, are discussed. This section is not intended to be a summary but rather a discussion of the implications of the examples and solutions of the earlier sections. The epilogue may raise new questions, in effect pointing to new areas of research. This section may also suggest exercises for the reader.

1.3 Overview

The case studies presented in Section 2 of this report are partitioned into five areas: naming conventions, types, coding paradigms, exceptions, and program structure. These general categories constitute Sections 2.1 through 2.5 respectively of this report. The ordering of these sections is deliberate, progressing from simpler, local issues to complex, global issues. This organization differs from the order in which these subjects are applied in the program life cycle. During design, program structure is the first consideration. Section 3.1 discusses approaches to program structure and development in depth.

The topics covered by some of the case studies are related and it is worth noting these relationships. The following table lists the fifteen case study topics and the section number in which the case study is found:

<u>Topic</u>	<u>Section</u>
Guidelines for the selection of identifiers	2.1.1
Discrete types	2.2.1
Set types	2.2.2
Constant array declarations	2.2.3
Record types	2.2.4
Recursive types	2.2.5
Array slices	2.3.1
Short circuit control forms	2.3.2
Loops	2.3.3
Block statements	2.3.4
Exceptions	2.4.1
Reusable packages: I/O	2.5.1
Information hiding	2.5.2
Reducing depth of nesting	2.5.3
Library units	2.5.4

Ada allows identifiers of any length, but it also requires identifiers to name a wide variety of entities. Case study 2.1.1, "Guidelines for the Selection of Identifiers," discusses solutions to the problem of naming all these entities in a descriptive and consistent way. This case study establishes the naming conventions followed in the subsequent case studies.

Ada's most powerful abstraction tool may be the ability to declare new types. Section 2.2 is devoted to various aspects of Ada type declarations. Case study 2.2.1, "Discrete Types," describes the use of enumeration types in situations where previous languages required use of integers. One of these uses is as an array index. Case study 2.2.2, "Implementation of Set Types," uses discrete values as arrays indexed to implement a generic package providing Pascal-like set types. A quite different use of arrays, as tables holding fixed data is described in

case study 2.2.3, "Constant Array Declarations." That case study deals with several questions of basic Ada usage. Case study 2.2.4, "Record Types," discusses the role of records with and without discriminants in data modeling. Case study 2.2.5, "Recursive Type Definitions," illustrates first the conventional mechanism for defining a type in terms of itself and then a more unusual form of recursive type involving links among direct access file elements.

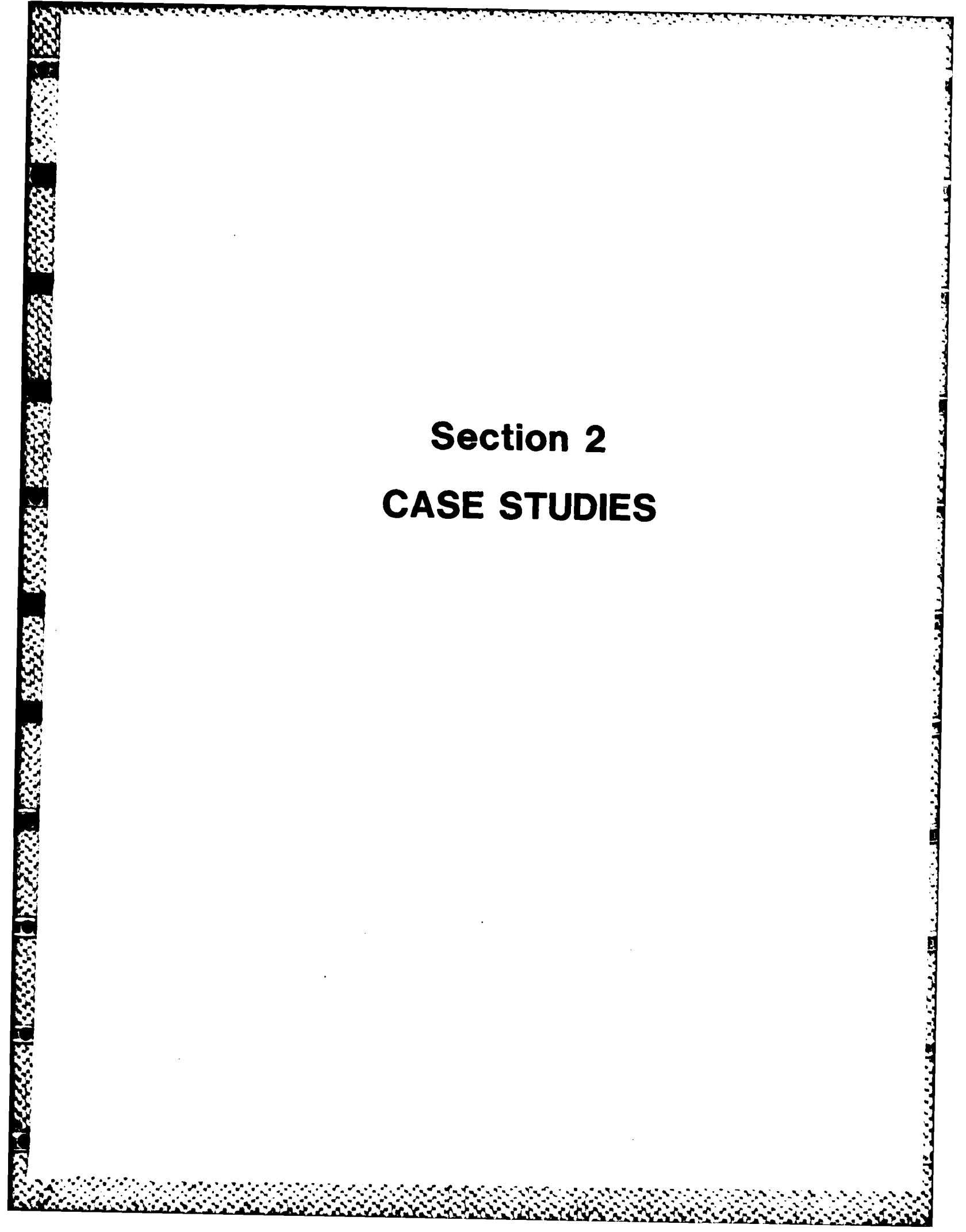
Section 2.3 is concerned with distinctive aspects of Ada arising in the coding of statements and expressions. Case study 2.3.1, "Use of Slices," describes how loops processing arrays one element at a time can often be rewritten in Ada as single assignment statements processing an entire slice. Case study 2.3.2, "Short Circuit Control Forms," explains the intended use of the and then and or else operations -- not to optimize the evaluation of Boolean expressions, but to allow Boolean expressions in which one operand might not be defined in cases where the value of that operand cannot affect the ultimate result anyway. One of the examples illustrating the proper use of short circuit control forms, a loop to search an array for a particular value, also figures prominently in case study 2.3.3, "Loop Statements." That case study compares various alternatives for writing loops and specifying how loops terminate. Case study 2.3.4, "Use of Block Statements for Local Renaming," shows how a block statement containing renaming declarations can be used in much the same way as a Pascal with statement.

Exceptions are a powerful feature of Ada, but there is little agreement about how they should be used, or whether they should be used at all. Case study 2.4.1, "The Use of Exceptions," addresses this issue. It presents six different uses of exceptions. Some of these uses are localized and can be considered coding paradigms. Others involve large portions of a program and raise difficult design questions. Because exceptions do not fall entirely into the purview of section 2.3, "Coding Issues," or section 2.5, "Program Structure Issues," this case study was placed in a section of its own.

Whereas the earlier sections are primarily concerned with the elements of Ada programs, section 2.5 is concerned with how the elements are put together to form a program. This section deals with issues of program structure and design. One of Ada's goals is to promote the development of generally reusable software components. This issue is addressed in case studies 2.2.2 and 2.4.1, but case study 2.5.1, "Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output," discusses how difficult it is to meet this goal. Another central theme in Ada is information hiding -- the distinction between interfaces and implementations. This theme is pervasive, playing an important role in case studies 2.2.2, 2.2.5, 2.4.1, 2.5.1, and 2.5.4. It is the primary subject of case study 2.5.2, "Information Hiding." Case study 2.5.3, "Reducing Depth of Nesting," explains why deeply nested programs are difficult to understand and offers some strategies for avoiding deep nesting. In contrast to case study 2.3.4, which shows how the introduction of block statements may be useful, this

case study illustrates how they may be removed to make programs more readable. Another depth-reducing strategy discussed in case study 2.5.4 is the use of body stubs, which allow nested program text to be placed in a separate subunit without affecting the logical structure of the program. The use of subunits raises some fundamental issues concerning the modular structure of programs. These issues are discussed in case study 2.5.4, "Library Units Versus Subunits," which discusses the relative merits of using library units and subunits as the basic tools for programming in the large. Issues of module-level program structure, coupling and cohesion of modules, scope of declaration, and recompilation costs all affect the choice between library units and subunits.

THIS PAGE INTENTIONALLY LEFT BLANK



Section 2

CASE STUDIES

2.1 Naming Conventions

This section contains the following case study:

Guidelines for the Selection of Identifiers

THIS PAGE INTENTIONALLY LEFT BLANK

2.1.1 GUIDELINES FOR THE SELECTION OF IDENTIFIERS

1. BACKGROUND

Case Study Objective

To explore the requirements for good identifiers in a large Ada program, to describe the problems encountered when trying to formulate good Ada identifiers, and to provide illustrative guidelines for selecting identifiers.

Problem

One of Ada's goals is to promote the writing of easily maintained programs. A large Ada program contains an extraordinary number of entities that must be named. These names must be meaningful if the program is to be maintainable. Guidelines are needed for the formulation of identifiers, both to make programs easier to read and to constrain the choices programmers have to make. Without such guidelines, a significant amount of the programmer's time may be spent devising one name for a type, another for the one declared object belonging to that type, another for a record component belonging to the type, and still another for a formal parameter of the type. Guidelines would free a programmer to concentrate on more fundamental issues.

Discussion

Descriptive identifiers can substantially increase the readability and maintainability of a program. Many older languages severely restrict the length and form of identifiers. Ada allows the use of identifiers of any length (provided, of course, that the identifier fits on a line). It is important to make effective use of this power, for at least three reasons. First, there may be several different kinds of entities with closely related meanings in an Ada program, and it is desirable to give each of these a name that describes it uniquely. Second, the large number of reserved words eliminates many good candidates for identifiers. Third, context clauses allow large numbers of entities declared in separately compiled units to be used without local declaration; it is incumbent upon the identifiers naming these entities to document their purpose.

2. DETAILED EXAMPLE

Example Problem Statement

The case study is based on a message switch program that provides an excellent illustration of the issues involved in selecting identifiers. The program is tens of thousands of lines long and contains hundreds of identifiers. For the most part, it is not necessary to understand the role of each component of this program in order to appreciate the naming issues. The large variety of entities requiring identifiers, the impact of reserved words, and the effect of context clauses will be considered in turn.

The Large Variety of Entities To Be Named

Among the entities named by identifiers are objects (i.e., variables and constants), named numbers, types, subtypes, non-character enumeration literals, record components (including discriminants), packages, subprograms, tasks, entries, subprogram and entry formal parameters, generic templates, generic instantiations, generic formal parameters, exceptions, loop parameters, named loops, named blocks, and labels. In addition, the language and implementation use identifiers as attribute designators, pragma names, and pragma argument names. It is no wonder that it is difficult to find identifiers appropriate for all these purposes.

Even different kinds of entities can have meanings so closely related that they "compete" for the same small collection of descriptive names. In the message switch program, the type MSGID points to a VERSION_HEADER record, whose components include the line number of the "logical output line." The package MESSAGE_OPS contains subprograms to read and set components of specified VERSION_HEADER records. These subprograms provide evidence of this competition for names.

For instance, the procedure SET_LOGICAL_LINE takes two parameters -- a MSGID and a logical line number -- and sets the appropriate component of the VERSION_HEADER designated by the MSGID value to the specified logical line number. The programmer requires appropriate names for the type whose values are logical line numbers, for the VERSION_HEADER component containing the logical output line, and for the formal parameter of SET_LOGICAL_LINE specifying the logical output line. The names chosen were LOGICAL_LINE, LOGICAL_OUTPUT_LINE, and LINE_NUMBER, respectively. However, these choices are almost completely interchangeable. (LOGICAL_OUTPUT_LINE would not be an appropriate name for the type, since values in the type can also identify logical input lines. Each of the remaining four permutations is feasible.)

The `VERSION_HEADER` record also contains a component indicating whether the message is being transmitted in the ASCII or ITA character set. This component is set by the procedure `SET_CHAR_TYPE`. The enumeration type containing one value for each character set is named `CHAR_SET_TYPE`, the corresponding component of the `VERSION_HEADER` record is named `CHAR_TYPE`, and the formal parameter is named `SET`. Again, the three names chosen are virtually synonymous, and the apparent scarcity of good names leads to choices that are less than ideal. (The type name `CHAR_SET_TYPE` is easily confused with the procedure name `SET_CHAR_TYPE`, the word `SET` used to name the parameter has several other meanings that come to mind more readily, and the component name `CHAR_TYPE` sounds like the name of a type.)

In other procedures in package `MESSAGE_OPS`, the scarcity of distinct, appropriate names is even more apparent. In `SET_PRECEDENCE`, the type of the value being passed is named `PRECEDENCE`, and the formal parameter and record component are both named `PREC`. In `SET_FORMAT`, the type of the value being passed is named `FORMAT`, and the formal parameter and record component are both named `FMT`.

These choices seem to reflect policy that, when good names are scarce, good type names are more important than good record component or formal parameter names. This is borne out by the message switch program's record type `LOG_REQUEST`. The identifiers `CANCEL_REASON`, `SCRUB_REASON`, and `REJECT_REASON` name enumerated types. The identifiers `CAN_REASON`, `SCR_REASON`, and `REJ_REASON`, respectively, name record components belonging to these types. Throughout one package, the identifier `SEG` is used for formal parameters pointing to values in a type named `SEGMENT`. The record type `SEGMENT` itself includes a component named `CLASS` belonging to a type named `SECURITY_CLASSIFICATION`, a component named `PART` belonging to a type named `PART_NAME`, a component named `TIME` belonging to a type named `DATE_TIME`, and even a component named `SEG_NO` belonging to a type named `SEGMENT_NUMBER`.

It is easy to justify this policy: type names may appear by themselves throughout a very large system. Record component names appear by themselves in the very limited context of a record type declaration, but elsewhere they appear only as part of a selected component or aggregate. Subprogram formal parameter names appear by themselves in the relatively limited context of a subprogram specification or body, but elsewhere they appear only within a named parameter association in a subprogram call. Thus, when a compromise has to be made, it makes sense to use less descriptive identifiers for record components and formal parameters, taking advantage of the explanatory context in which these identifiers will appear. Of course it would be better if there were no need to compromise.

In languages like Ada that support encapsulation of data types, the programmer has to come up with yet another set of identifiers -- names for the subprograms that implement the abstract operations of the data type. Typically, the encapsulated data type is implemented as a record,

and there are subprograms to set and retrieve the values of various components. The package LINE_TBL_OPS has a private type LOGICAL_LINE, implemented as a record. Among the components of the record are FORMAT, PREFERRED_LMF, LEVEL, COMMUNITIES, XTS, MAX_RIS, CHARACTER_SET, and SPEC_TERM. All these components are set by the procedure SET_LOGICAL and retrieved by functions named READ_FORMAT, READ_PREFERRED_LMF, READ_LEVEL, READ_COMMUNITY, READ_XTS, READ_MAX_RIS, READ_CHARACTER_SET, and READ_SPEC_TERM. Similar conventions are followed for the private type PHYSICAL_PORT, also declared in LINE_TBL_OPS. The type VERSION_HEADER encapsulated in package MESSAGE_OPS has components named CATEGORY, CHAR_TYPE, CLASS, FMT, LOGICAL_OUTPUT_LINE, PREC, RI_COUNT, and TIME_OF_RECEIPT. Access to these components is through procedures named SET_CATEGORY, SET_CHAR_TYPE, SET_CLASS, SET_FORMAT, SET_LOGICAL_LINE, SET_PRECEDENCE, SET_NUM_RIS, and SET_TIME_OF_RECEIPT and functions named READ_CATEGORY, READ_CHAR_TYPE, READ_CLASS, READ_FORMAT, READ_LOGICAL_LINE, READ_PRECEDENCE, READ_NUM_RIS, and READ_TIME_OF_RECEIPT. This is a very consistent and descriptive naming convention, but it makes calls on the inquiry functions read awkwardly, as evidenced by the following excerpts from the message switch code:

Excerpt 1:

```

while LINE_PTR /= null loop
  if READ_PRECEDENCE (MESSAGE) >
    LINE_TBL_OPS.READ_PRECEDENCE (LINE_PTR.PHYS_LINE) then
    if PREEMPT_PTR = null then
      PREEMPT_PTR := LINE_PTR;
    elsif LINE_TBL_OPS.READ_PRECEDENCE (PREEMPT_PTR.PHYS_LINE) >
      LINE_TBL_OPS.READ_PRECEDENCE (LINE_PTR.PHYS_LINE) or else
      ( LINE_TBL_OPS.READ_PRECEDENCE (PREEMPT_PTR.PHYS_LINE) =
        LINE_TBL_OPS.READ_PRECEDENCE (LINE_PTR.PHYS_LINE) and
        LINE_TBL_OPS.READ_START_TIME (PREEMPT_PTR.PHYS_LINE) >
        LINE_TBL_OPS.READ_START_TIME (LINE_PTR.PHYS_LINE) ) then
        PREEMPT_PTR := LINE_PTR;
      end if;
    end if;
    LINE_PTR := LINE_PTR.NEXT;
  end loop;

```


Excerpt 2:

```
-- Check linkage for all new segments.
-- Items checked for sameness are part, classification, and ASN.
-- Segment number is checked for one more than the previous one.
if READ_PART (SEG => NEXT) /=
  READ_PART (SEG => WHERE_FROM.SEG)
  or else
  READ_CLASS (SEG => NEXT) /=
  READ_CLASS (SEG => WHERE_FROM.SEG)
  or else
  READ_EASN (SEG => NEXT).ASN /=
  READ_EASN (SEG => WHERE_FROM.SEG).ASN
  or else
  READ_SEGMENT_NUMBER (SEG => NEXT) /=
  READ_SEGMENT_NUMBER (SEG => WHERE_FROM.SEG) + 1 then
  STATUS := LINKAGE_ERROR; -- return error code
  exit;
end if;
```

These excerpts would read more naturally if the READ_ prefix were dropped from the function names. However, the function names would then clash with the record component names in many cases. (This would not make the program illegal, just confusing.)

Named parameter associations complicate the selection of formal parameter names because the names chosen must be meaningful at two levels of abstraction -- within the subprogram body and in calls on the subprogram. It is possible to name parameters in a way that makes subprogram calls extremely readable. For example, a hypothetical package implementing stacks could name parameters in such a way as to allow calls like PUSH(X, ON_TO=>S) and POP(X, OFF_OF=>S). Within the procedures PUSH and POP, however, ON_TO and OFF_OF are very unnatural names for stacks. In contrast, a name like LIST_HEAD for a stack parameter can make a subprogram body easy to understand, by documenting that the stack is implemented as a linked list; but it forces named subprogram calls to reflect the implementation of the subprogram.

The contractor for the message switch professed difficulty in coming up with distinct meaningful names both for tasks and their entries, expressing dissatisfaction with a task named FREE_VER having a single entry named FREE_VERSION. In addition, there is a task type GEN_SVC with a single entry GENERATE_SVC_MESSAGE. Two tasks named TRANS have entries named MIT. Calls on these entries read TRANS.MIT(C), but ACCEPT statements for entry MIT appear quite enigmatic.

In at least one case, there was difficulty coming up with distinct meaningful identifiers for both a loop name and a label. There is a task body containing a loop statement that is executed between messages. It is necessary to identify the loop both with a loop name so that the

loop can be exited and with a label so that the loop can be branched to. Since two identifiers are needed to refer to the same loop statement, there is a scarcity of appropriate names. The resulting code reads as follows:

```
<<BETWEEN>>
  B_TWEEN:
    loop
      .
      .
      .
      exit B_TWEEN;
    end loop B_TWEEN;
    .
    .
    .
    goto BETWEEN ;
```

There are also clashes of program unit names with other identifiers. For instance, ASN names both the package in which type SER_NO_TYPE is declared and the SER_NO_TYPE component of the EXT_SERIAL_NO record type. The procedure CANTRAN_SEQ contains a string constant named CANTRAN_SEQUENCE. The package PROCESS_MSG contains a task named PROCESS_MESSAGE.

Reserved Words

Sixty-three of the most likely candidates for identifiers are ruled out as reserved words. If the reserved words had been selected randomly from a dictionary, their impact on the pool of identifiers would be minimal. Unfortunately, the reserved words are words descriptive of actions and situations that arise when executing a computer program. Words like ACCEPT, ACCESS, BEGIN, CONSTANT, DELAY, DIGITS, ENTRY, EXIT, NEW, RANGE, RECORD, REVERSE, SELECT, TERMINATE, and TYPE are descriptive identifiers for a number of applications. Given the high demand for descriptive identifiers, the reserved words restrict the supply of identifiers significantly.

The message switch program contains a task type with two entries, named INITIATE and TERM. Considerations of uniformity and clarity suggest that TERMINATE would be a better name for the second entry, but that word is reserved for use in SELECT statements. The enumeration type SPEC_TERM_TYPE declared in package LINE_TBL_OPS has a literal named ACCES because the intended name, ACCESS, is reserved. Of course it is irrelevant that the programmer would have used these words in a totally different sense from the language.

The Effect of Context Clauses

A context clause consists of WITH clauses and, optionally, USE clauses. The WITH clause makes it possible to write expanded names referring to entities declared in separately compiled packages. (The expanded names have the form package-name.entity-name.) To see how these entities are declared, one must go back to the package named in the expanded name. A USE clause makes it possible to refer to an entity by its simple name rather than an expanded name.

In a large program, context clauses introduce a multitude of identifiers that are not declared locally. USE clauses can make it extremely difficult to track down the declaration of an identifier in some other unit. In such an environment, descriptive identifiers are not just a stylistic nicety — they are essential for readability. It is not enough that an identifier suggest what the entity it names is related to. The identifier must precisely indicate the meaning of the entity. In many cases the reader will be relying on identifiers to convey much of the information normally found in declarations.

The visible part of the package LINE_TBL_OPS contains the following type declaration:

```
type COMMUNITIES_SERVED is (R, U, RU, Y, RY, UY, RUY);
```

If this type were meant only for local use within LINE_TBL_OPS, these enumeration literals might be adequate. In the context of the entire list of enumeration literals, it is clear that each individual literal refers to a subset of the set {R,U,Y}. However, any unit containing a WITH clause and a USE clause for LINE_TBL_OPS (and there are several such units) can refer to these identifiers. Without the context of the type declaration, the identifiers are quite cryptic.

Context clauses do not really present additional difficulties in the selection of good identifiers. However, they drastically increase the importance of descriptive names, and make it essential that the problems discussed earlier are overcome.

Solution Outline

Effective Ada programs must take advantage of the ability to write long identifiers. Identifiers that describe their own meaning act simultaneously as identifiers and comments, and help make a program self-documenting. In order not to impose a cognitive burden on the reader, an identifier should be pronounceable and easy to associate with the underlying meaning. A programming project should have standard conventions for naming different kinds of entities. For instance, these conventions might involve different suffixes or different parts of speech for different kinds of entities. It is often possible to use abbreviations effectively to produce concise yet readily understood

names, but the set of abbreviations to be used by a project should be carefully controlled and documented. Finally, consideration should be given to making identifiers more expressive by writing them in mixed upper and lower case.

Detailed Solution

Unfortunately, the restrictions in earlier languages have led to a habit among programmers of using short, cleverly encoded names. This habit persists even when programmers are given the opportunity to write longer names. Thus the message switch program includes names like Q_HEAD rather than QUEUE_HEAD, ERR_STAT rather than ERROR_STATUS, F_STRING rather than FRAME_STRING, TRLR_LGTH rather than TRAILER_LENGTH, HSTAT rather than HISTORY_STATUS, SERNO rather than SERIAL_NUMBER, GENCC rather than GENERATED_CONTROL_CHARACTER_TASK, and RECVCC rather than RECEIVED_CONTROL_CHARACTER_TASK. The first step in writing good Ada identifiers is to break this habit. This does not mean that all identifiers must be long. Excessively long identifiers can also decrease readability. However, a programmer must not eliminate an identifier from consideration simply because of its length.

Self-Documenting Identifiers -- The Reader's Perspective

Good identifiers serve as comments, making programs largely self-explanatory. Self-documenting identifiers are written as descriptions meant to enlighten the reader rather than as codes meant to jog the memory of the writer. The short identifiers listed above are probably adequate to remind the writer of the purpose he has in mind for each entity named; they are inadequate only because they fail to help a reader.

The message switch program contains a package named LINE_TBL_OPS that provides subprograms manipulating logical and physical communications lines. To a person familiar with the capabilities provided by the package, the names of the subprograms clearly identify which capabilities are provided by which subprograms. There are, among others, subprograms to start and stop using a logical line's alternate physical lines, to check whether a logical line's alternate physical lines are currently in use, to mark a physical line as available or unavailable, to set the characteristics of a logical line, and to set, reset, or test the "kill flag" associated with a physical line. The names of the subprograms providing these capabilities are:

```

BEGIN_ALT
END_ALT
IS_ALT
MAKE_AVAILABLE
NOT_AVAILABLE
SET_LOGICAL
SET_KILL
RESET_KILL
IS_KILL

```

It is easy to remember which subprograms provide which capabilities -- if one is thoroughly familiar with the capabilities. However, the names are not helpful to a reader trying to understand calls on these subprograms in separately compiled units. The following names would make the purpose and effect of the calls much clearer:

```

START_USING_ALTERNATES
STOP_USING_ALTERNATES
IS_USING_ALTERNATES
MAKE_AVAILABLE
MAKE_UNAVAILABLE
SET_LOGICAL_LINE_CHARACTERISTICS
SET_KILL_FLAG
RESET_KILL_FLAG
KILL_FLAG_IS_SET

```

These names serve as comments, making it unnecessary to look up the actual comments in the LINE_TBL_OPS package to understand what each subprogram does.

The package QUEUE implements a queue of messages as a header record pointing to the first and last elements of a doubly-linked list. The visible part of the package declares two types, named HEAD and SET. An object of type HEAD is the header record of a list, and is the external manifestation of a queue. An object of the type SET is a two-dimensional array of HEAD objects, indexed by message categories and precedence levels. The names HEAD and SET have obvious meanings within the package QUEUE, but they make it difficult to understand a package that uses QUEUE. HEAD describes the implementation of a queue, not the fact that what it represents is indeed a queue. (The reader of a package using QUEUE is not concerned with the implementation of a queue, since it is implemented as a private type.) The name SET is too general, because the writer should not expect the reader to know what the elements of the set are. Names like MESSAGE_QUEUE and SET_OF_MESSAGE_QUEUES would make things clearer to the reader.

As a final illustration of the distinction between the writer's perspective and the reader's perspective, consider the task type used for making log entries to record message switch activities. The writers paid careful attention to decoupling the recording of the log entry from message processing, so that these two activities could proceed in

parallel. Unfortunately, this concern is evident in the name of the logging task type — DECOUPLE. The statements used to make a new log entry are as follows:

```
DECOUP := new DECOUPLE;  
DECOUP.LOG ([actual parameter to entry call]);  
DECOUP := null;
```

The reader's first concern is not how logging is implemented, but simply that these statements have the effect of making a log entry. This would be better conveyed by different names:

```
DECOUPLED_LOGGING_TASK := new LOGGING_TASK;  
DECOUPLED_LOGGING_TASK.LOG([actual parameter]);  
DECOUPLED_LOGGING_TASK := null;
```

These names continue to document that the creation of a new logging task decouples logging activity from other processing. However, the main message conveyed is that a log entry is being made. The definition of the logging task itself is more readily understood because the name explains what the task accomplishes rather than hinting at how it will be activated.

The Mental Image Of An Identifier

Adults cannot remember more than five to nine random letters at once. We are able to use longer identifiers only because we think of them as a sequence of pronounceable syllables, a sequence of words, or an abstract notion rather than as a sequence of letters. Even an identifier consisting of a few letters is more easily remembered on an abstract level than as a sequence of letters.

Consequently, identifiers should be pronounceable. The identifier SCTY_MISMATCH is not pronounceable. Neither are the identifiers SVC_MSG_TYPE and SVC_MSG_INFO (in which SVC is used as an abbreviation for SERVICE). Furthermore, since identifiers are often remembered by their pronunciation, distinct identifiers should have distinct pronunciations. The message switch program has a record type with one component named RIS (meant to be pronounced as "R.I.'s" -- the plural of R.I.) and another named RI_S. Earlier, we described a label BETWEEN adjacent to a loop name B_TWEEN and a package named PROCESS_MSG containing a task named PROCESS_MESSAGE. More subtly, there is a task body containing both an ACCEPT statement with a parameter REQ (for request) and a block statement with a local variable REC (for record). For the same reason that distinct identifiers should have distinct pronunciations, it is a bad idea to use a misspelled word as an identifier when the correct spelling is reserved, as with ACCES.

Ideally, different identifiers should refer to different underlying abstract notions. Just as it is easier to remember a sequence of syllables than the sequence of letters spelling the syllables, so it is easier to remember an abstract concept than the sequence of syllables describing the concept. The message switch program contains a task body with one variable named STATUS and another named STAT. Even though these two identifiers have distinct pronunciations, they have the same underlying meaning, and are easily confused. The reader is forced to think in terms of identifiers' pronunciations rather than their meanings. As mentioned earlier, the LOG_REQUEST record type has components named CAN_REASON, SCR_REASON, and REJ_REASON, belonging to types named CANCEL_REASON, SCRUB_REASON, and REJECT_REASON. Corresponding component and type names have distinct pronunciations, but identical underlying meanings. Similarly, the program has a variable named PHYS_LINE belonging to a type named PHYSICAL_LINE. If the type were renamed PHYSICAL_LINE_TYPE, the underlying meanings would be different.

Following these principles allows the reader to concentrate on the meaning of an identifier rather than on how that meaning is expressed. Ease of transcribing the identifiers in one part of the program to another is both a benefit of following the principles and a yardstick for measuring how well the principles have been followed. Ease of transcription is enhanced when there are standard, easily-learned conventions for naming identifiers. It should be easy to predict the composition of an identifier given a description of its meaning. In particular, succinct, abbreviated identifiers should not be used in the same place and for the same purpose as verbose, fully-spelled identifiers. For example, one package in the message switch declares an enumeration type with the fully-spelled name SECURITY_CLASSIFICATION, and with enumeration literals like RESTRICTED, CONFIDENTIAL, and TOP_SECRET. Thus the use of the abbreviated literal UNCLASS in place of UNCLASSIFIED is unexpected. In the same package, a type named PRECEDENCE has enumeration literals named ROUTINE, PRIORITY, and IMMEDIATE, but CRITIC rather than CRITICAL. The enumeration literals of type SVC_MSG_TYPE are a mixture of expansive identifiers like ILLEGAL_EXCHANGE, INVALID_BLOCK_COUNT, SUSPECTED_STRAGGLER, TRAFFIC_CHECK, and SUSPENDED_TRANSMISSION, and terse cryptic identifiers like HI_PREC_ACC, INVALID_EOM_REJ, and TWO_CONSEC_SOM.

Conventions

A convention for naming identifiers can help provide a large supply of descriptive identifiers, ensure that different identifiers have distinct underlying meanings, and make the composition of identifiers predictable. As one writer has noted [Car82], in the absence of conventions, "not only does each programmer have to learn a new 'language' (the language of the identifiers) each time someone else's code is read, but programmers also spend a lot of time inventing (and trying to remember) new names for old things." A good convention will be

easy to learn and will serve as a tool rather than an obstacle to the programmer. It will not consist merely of a set of restrictions, but will provide help in generating good identifiers.

There is no one "correct" convention. Still, there are certain common-sense rules that will probably be part of any convention for Ada identifiers. These include rules on the use of underscores, hiding, and abbreviations.

Whenever an identifier consists of two or more English words or abbreviations, the components of the identifier should be separated by underscores. For example, the message switch program contains a record type with components named GENCC and RECVCC. Rewriting these names as GEN_CC and RECV_CC makes it more likely that a reader will perceive the names as abbreviations for "generate control character" and "receive control character" rather than a hodgepodge of letters. There is also a task body containing procedures named HPJANAP and HPACP. Nested within HPACP are procedures named HPHEADER and COMPLETE_HEADER. JANAP and ACP are recognized abbreviations for two communication protocols, and HP is a more esoteric abbreviation for high precedence. These abbreviations would be more recognizable if the identifiers were written as HP_JANAP, HP_ACP, HP_HEADER, and COMPLETE_HP_HEADER. Besides making identifiers more readable in most cases, a strict adherence to the use of underscores also makes the composition of identifiers more predictable. Thus the NUM_INTRANSIT component of the AUDIT_RECORD type should be renamed NUM_IN_TRANSIT, even though the improvement in readability is minimal. Among the other components of type AUDIT_RECORD are NUM_IN_OVERFLOW and NUM_IN_INTERCEPT, so there is a significant difference in predictability.

Like most block-structured languages, Ada allows a declaration in an inner scope to hide a declaration of the same identifier in an outer scope. However, Ada is unique in allowing the outer entity to be made visible in the inner scope with an expanded name. For example, the task MANAGE_OUTGOING has an entry declared as follows:

```
entry INIT (LOG_ID : LOGICAL_LINE);
```

The declarative part of the MANAGE_OUTGOING task body has a declaration for a variable named LOG_ID, and the sequence of statements includes the following statement:

```
accept INIT (LOG_ID : LOGICAL_LINE) do
  MANAGE_OUTGOING.LOG_ID := LOG_ID;
end INIT;
```

Identifiers that hide other identifiers can be confusing, and ought to be avoided. This is especially true when it is necessary to refer to the hidden entity in the inner scope. It would be better to rename the entry parameter, so that the ACCEPT statement can be written as follows:


```

accept INIT(LOG_ID_PARAMETER : LOGICAL_LINE) do
  LOG_ID := LOG_ID_PARAMETER;
end INIT;

```

Perhaps the greatest impediment to readable identifiers is excessive and inconsistent abbreviation. Any convention for identifiers must include rules for forming abbreviations. This topic is treated more fully in the next section.

A useful guideline in choosing identifiers is that the statements of the resulting program should read well in English. It may be possible to achieve this through conventions calling for certain English grammatical constructs to be used to name certain kinds of entities.

Imperative clauses are appropriate for procedure and entry names (such as SET_COUNTING_FORMAT, APPEND_LINE, TRANSMIT_CONTROL_SEQUENCE, and TRANSLATE_MESSAGE). Often, the procedure's parameters can act as direct objects or objects of prepositions in these clauses. For instance, the procedure call MAKE_AVAILABLE(PHYS_LINE=>L) can be read as "Make physical line L available," and DROP_LINE(PHYS_LINE=>P, LOG_LINE=>L) can be read as "Drop physical line P from logical line L." A natural extension is to use prepositions as formal parameter names, so that calls with named parameter associations more closely resemble the imperative clauses they are based on. For instance, changing the LOG_LINE parameter of DROP_LINE to FROM allows us to write DROP_LINE(P, FROM=>L). Such a practice will make the body of the procedure read very unnaturally, however, unless the body contains a renaming declaration for the parameter named as a preposition. Another problem is that five of the most common prepositions -- at, for, in, of, and with -- are reserved words.

Noun phrases are often appropriate for identifiers naming variables, constants, numbers, and functions -- the entities that appear in expressions, denoting values. Of course the noun phrase should describe the value being named. Examples are a variable named MESSAGE, a constant named TRAILER, and functions named HARDWARE_CLOCK_READING and NEXT_SEGMENT. Function parameters can act as objects of prepositions in the noun phrase (so that NEXT_SEGMENT(X) can be read as "the next segment after X").

An exception should probably be made for Boolean-valued variables and functions. These are probably best named by declarative clauses (such as READING_IS_COMPLETE) or elliptic fragments that clearly stand for declarative clauses (such as LINE_AVAILABLE, which clearly stands for "The line is available."). The reason for this exception is that Boolean expressions most often appear as conditions in IF, WHILE, EXIT, and SELECT statements, following a subordinate conjunction (if, while, or when). A clause following the subordinate conjunction would make the statement (or at least that part of the statement) read like an English sentence. (Relational expressions like A<B and X in 1..10, which also

frequently appear in these contexts, can also be read as declarative clauses.) Parameters of Boolean functions tend to serve as subjects rather than objects of the clause. Thus `VALID_PHYS_LINE(L)` means "L is a valid physical line," and "`IS_AVAILABLE(L)`" means "L is available." Interestingly, a predicate in the mathematical sense -- a function returning a Boolean value -- is often named well by a predicate in the grammatical sense -- the part of a clause that expresses something about the subject.

Since a task is an object that works to accomplish some assigned duty, it is often appropriate to name a task with a noun formed by adding the `-er` suffix to a verb. This approach resolves the dilemma of how to name both tasks and their entries. Entry `FREE_VER.FREE_VERSION` can be named `VERSION_FREER.FREE_VERSION`, entry `TRANS.MIT` can be named `TRANSMITTER.TRANSMIT`, and the task type `GEN_SVC` whose single entry is `GENERATE_SVC_MESSAGE` can be renamed `SVC_MESSAGE_GENERATOR`. In general, entry calls will be of the form actor.action (parameters).

The criterion that an Ada statement should read like an English sentence, and a set of objective grammatical guidelines for choosing names that advance this end, make it easier to understand why certain names seem unsatisfactory. Earlier, it was observed that names like `READ_PART`, `READ_CLASS`, and `READ_SEGMENT_NUMBER` for functions retrieving components of a private type cause function calls to read awkwardly. The problem is that the names are based on imperative clauses rather than noun phrases. Names like `PART_OF`, `CLASS_OF`, and `SEGMENT_OF`, which cause the parameterized function call to read like a noun phrase, result in more English-like code:

```

if PART_OF (NEXT) /= PART_OF (WHERE_FROM.SEG) or else
   CLASS_OF (NEXT) /= CLASS_OF (WHERE_FROM.SEG) or else
   SEGMENT_NUMBER_OF (NEXT) /=
      SEGMENT_NUMBER_OF (WHERE_FROM.SEG) + 1 then
   STATUS := LINKAGE_ERROR;
   exit;
end if;

```

One subprogram of the message switch program contains a constant array holding, for each possible character set, the starting position of a channel sequence number (CSN) in that character set's "transmission identifier". The array is named `START_CSN`, but a better name would be `START_OF_CSN`. The word `of` makes the identifier read naturally as a noun phrase and also prevents its being mistaken for an imperative clause. A procedure named `NOT_AVAILABLE` marks a physical line as being unavailable. A better name would be `MAKE_UNAVAILABLE`, wince `MAKE_UNAVAILABLE(P)` reads like an imperative clause, while `NOT_AVAILABLE(P)` can be read as an ellipsis for the declarative clause "P is not available."

Grammatical considerations like these are not a complete solution to the problem of naming different kinds of entities with closely related meanings. We have not yet addressed how to name types and record components. Since variables, parameters, types, and record components often compete for the same names, one approach is to require distinctive suffixes on identifiers naming different kinds of entities. A convention might require that all type names end with _TYPE and all record component names with _PART, _FIELD, or _COMPONENT. Variable and constant names constitute the bulk of a program, and should read naturally as noun phrases describing the values held by the named objects, so they should probably be named without suffixes. Parameters, which are used as ordinary variables or constants inside a subprogram body or ACCEPT statement, should follow the same rule. There is also merit in requiring package names to end with _PACKAGE. As an alternative or supplement to the -er rule for tasks, a convention might require task type names to end with _TASK_TYPE, since task types can be used for data abstraction and such a name highlights the type's implementation rather than its behavior.)

Distinctive suffixes allow names of closely related entities to be derived from the same root. This makes it easier to generate all the required identifiers. For instance, LOGICAL_LINE_TYPE could name the type whose values are logical line numbers, LOGICAL_LINE_PART could name the component of a VERSION_HEADER record belonging to that type, and LOGICAL_LINE could name the parameter of SET_LOGICAL_LINE specifying the value to which the LOGICAL_LINE_PART component of a given record should be set.

The resulting identifiers are largely self-documenting. They accurately describe the entities they name and also convey much of the information one normally finds in declarations. Because of the suffixes required on the names of other entities, there is a larger pool of good names available for subprograms, variables, constants, and parameters.

There is a fairly common circumstance in which these conventions lead to a difficulty. Often, an enumeration type is introduced exclusively to serve as the discriminant of a record type with variants. If XXX is a word describing the kind of object represented by the record, then XXX_TYPE is a natural name for the enumeration type: Each enumeration value stands for a different "type" of XXX. But XXX_TYPE is just the name that the conventions would use for the data type whose values are XXX's, the record type! For example, package HISTORY_OPS includes the following declarations:

```
type ENTRY_TYPE is (LOG, RI_SET, SEGMENT);
```

```

.
.
.
type HISTORY_ENTRY (ENT : ENTRY_TYPE := SEGMENT; ...; ...) is
  record
    case ENT is
      when LOG =>
        ...
      when RI_SET =>
        ...
      when SEGMENT =>
        ...
    end case;
  end record;
```

Under the suffix conventions, the name HISTORY_ENTRY should be changed to HISTORY_ENTRY_TYPE, but this will lead to confusion with the name ENTRY_TYPE, which has the same underlying meaning. A solution is to change the enumeration type name ENTRY_TYPE to ENTRY_CLASS_TYPE or ENTRY_VARIANT_TYPE, or better yet to HISTORY_ENTRY_CLASS_TYPE or HISTORY_ENTRY_VARIANT_TYPE. (These names sound redundant, but are really not. HISTORY_ENTRY_CLASS_TYPE names the type whose values are classes of history entries.) Then the declarations would appear as follows:

```

type HISTORY_ENTRY_CLASS_TYPE is (LOG, RI_SET, SEGMENT);
.
.
.
type HISTORY_ENTRY_TYPE(HISTORY_ENTRY_CLASS_PART:
                        HISTORY_ENTRY_CLASS_TYPE := SEGMENT;
                        ...;
                        ...) is
  record
    case HISTORY_ENTRY_CLASS_PART is
      when LOG =>
        ...
      when RI_SET =>
        ...
      when SEGMENT =>
        ...
    end case;
  end record;
```

The name HISTORY_ENTRY_CLASS would be reserved for objects and parameters of type HISTORY_ENTRY_CLASS_TYPE. (A possibly more elegant alternative is to suspend the _TYPE suffix for enumeration type names, and use the suffix _CHOICE instead. Then HISTORY_ENTRY_CLASS_TYPE becomes HISTORY_ENTRY_CLASS_CHOICE.)

A naming convention can address more specific properties than the class of entities being named. We have already suggested names like `PART_OF`, `CLASS_OF`, and `SEGMENT_NUMBER_OF` for the retrieval functions of a private type. (An alternative to suffixing with `_OF` is prefixing with the root describing the private type. This leads to function calls like `SEGMENT_PART(X)` and `SEGMENT_CLASS(X)`. Of course `SEGMENT_NUMBER(X)` would be more reasonable than `SEGMENT_SEGMENT_NUMBER(X)`.) There can also be special rules for enumeration literals, access type names, and the data types used to construct list data structures, among others.

A good case can be made for requiring enumeration literals to have suffixes identifying the type to which they belong, particularly if the enumeration literals appear in separately compiled units. For instance, the declaration

```
type CHAR_SET_TYPE is (ANY, ASC, ITA);
```

in package `GLOBAL_TYPE` would become

```
type CHAR_SET_TYPE is (ANY_CHAR_SET, ASCII_CHAR_SET, ITA_CHAR_SET);
```

and the declaration

```
type PART_NAME is (MCB, HEADER, MSG_BODY_TRAILER);
```

in the same package would become

```
type PART_NAME_TYPE is (MCB_PART_NAME, HEADER_PART_NAME,  
MSG_BODY_PART_NAME, TRAILER_PART_NAME);
```

The obvious approach to access type names is to base the name of the access type on the name of the type it designates. If the designated type has a name of the form `XXX_TYPE`, the access type can be named `XXX_POINTER_TYPE`. However, this rule should be applied flexibly, to accommodate cases in which the access type is an abstraction in its own right. For instance, messages in the message switch program are handled as values of type `MSGID`. `MSGID` values are access values designating records of type `VERSION_HEADER`. `VERSION_HEADER` is a descriptive name for the record type, but the name `VERSION_HEADER_POINTER_TYPE` fails to convey that pointers to these records should be thought of as the identities of messages.

Another abstraction of access values is the linked list. Linked lists arise so often that they may merit special naming rules. In various circumstances, the object thought of as "the value of the list" may be a special header record, a pointer to a special header record, the first cell of the list, or a pointer to the first cell of the list. The type of the object playing this role can be given a name ending with `_LIST_TYPE`. The other types involved can be given names with suffixes like `_LIST_CELL_TYPE`, `_LIST_CELL_POINTER_TYPE`, and `_LIST_HEADER_TYPE`, as

appropriate.

Abbreviation

Abbreviated identifiers tend to make programs difficult to read. Before he can begin to understand what a program does, a reader must decipher the meanings of the identifiers. Abbreviations tend to be much less obvious to the reader than to the person who devised them. The LOG in LOG_LINE might stand for logical, or it might be an unabbreviated verb. (In fact, LOG is used in both senses in different parts of the program. Another interpretation for LOG, though not in the message switch program, is logarithm.) The identifier CAT might be an abbreviation for category, catalog, or catenate. The RET in RET_MESSAGE might stand for return, retarget, or retry. PREC might mean precedence, precision, or preceding. CLASS, used as an abbreviation for classification, could really mean class. The use of SVC as an abbreviation for service is confusing because it is well known among programmers that SVC stands for supervisor call.

Many abbreviations appear to be the outgrowth of habits acquired using other languages, but there are legitimate uses of abbreviation in Ada. Some abbreviations are so immediately and universally recognizable that they can almost be considered words in their own right. Examples are CPU, IO (used even in the names of predefined Ada packages), and MAX. In addition, each application area and system design has a jargon with its own abbreviations. Among the standard abbreviations used in the message switch program are ACP and JANAP (names of communications protocols), CSN (Channel Sequence Number), EOM (End of Message), LMF (Line Media Format), MCB (Message Control Block), RI (Routing Indicator), and TI (Transmission Identifier). It is also appropriate to use abbreviations in a restrained and disciplined way simply so expressions and statements can fit on one line. Identifiers that are too long make programs difficult to format, possibly decreasing readability. Readability is best served by a tradeoff yielding identifiers long enough to understand yet short enough to format easily.

What is needed is not a ban on abbreviations, but a discipline for producing understandable abbreviations. Among the guidelines suggested by one PL/I programmer [Car82] are the following:

- There should be a consistent way of writing each English word used in identifiers. The identifiers NUM_RIS, EXT_SERIAL_NO, and SEGMENT_NUMBER illustrate a violation of this rule. So do the identifiers SEG_PTR and SEGMENT.
- An abbreviation should not be used unless it saves at least three letters. The abbreviations MST and LST for MOST and LEAST, and the abbreviation LGTH for LENGTH, violate this rule.

- It should not be possible to mistake the abbreviation of one word for the abbreviation or full form of another word. We have already seen several abbreviations that violate this rule.

There are various schools of thought on how to extract the most recognizable abbreviation from a word. One is that an abbreviation should only be formed by truncation (so that SEGMENT becomes SEG); another is that an abbreviation should be formed by dropping vowels and certain phonetically less significant consonants (so that SEGMENT becomes SGMT). In any event, the algorithm used for forming abbreviations should be standardized over a given program.

In fact, it is necessary to maintain a project-wide lexicon of standard abbreviations if there is to be a unique rendering of each English word. It is even suggested in [Car82] that a list be drawn up before coding begins, containing all the elements that will be joined by underscores to form identifiers. An important goal in formulating this list is to attain a consistent level of succinctness. MSG is an easily recognized abbreviation for MESSAGE, but if other words of similar length are spelled out in full, MESSAGE should be too. The list of standard abbreviations and their meanings will become an important part of a project's documentation, making the code much easier to read.

Mixing Upper and Lower Case

Akin to the habit of writing short, cryptic identifiers is the habit of typing identifiers entirely in upper case. For many years, the available hardware limited all computer data entry to upper case. Meanwhile, the Algol 60 report established a custom of setting keywords in lower case boldface in programming language definitions. Both of these influences are evident in the Ada Language Reference Manual [DoD83]. All examples in the Reference Manual use lower case boldface letters for reserved words (in accordance with the formal syntactic definition) and upper case letters for identifiers. This style of capitalization has been imitated by others in the Ada community -- with and without the use of boldface -- and may be evolving into a de facto standard.

However, lower and upper case letters are indistinguishable to an Ada compiler outside of character and string literals. Readability may be better served by the following convention: All reserved words are written entirely in the same case. (Upper case seems more appropriate to some, lower case to others). Identifiers are written in mixed upper and lower case, with upper case reserved for abbreviations and the beginnings of words. Single-letter identifiers (which are appropriate as loop parameters in short FOR loops) are written in the case not used for reserved words, to distinguish them from reserved words.

Little is lost by requiring that a single case be used for the fixed set of sixty-three familiar reserved words. In contrast, the ability to mix upper and lower case can add great clarity and expressiveness to programmers' identifiers. In particular, mixed case highlights the presence of an abbreviation in an identifier and makes pluralized abbreviations easier to recognize. Certain identifiers that are cryptic when written only in upper case become more descriptive when written in mixed case. Thus mixing cases enlarges the pool of good identifiers.

Identifiers like EOM_SEQ, BAD_MCB, LOG_SOM_IN, CHECK_EOM, SOM_SEQ_TYPE, CONVERT_TO_JANAP, and CONVERT_TO_ACP are more readily recognized when written as EOM_Seq, Bad_MCB, Log_SOM_In, check_EOM, SOM_Seq_Type, Convert_To_JANAP, and Convert_To_ACP. Sequences of initials stand out, and are not mistaken for words or truncations of words. Identifiers like BAD_RIS and CHECK_RIS appear to be talking about something called a RIS, but they really mean bad R.I.'s. This is more obvious when the identifiers are written as Bad_RIs and Check_RIs. Similarly, the pluralized abbreviations in PHYS_IDS and REMOVE_DELS are more easily recognized when the identifiers are written as Phys_IDs and Remove_DELs.

3. EPILOGUE

One of the major hurdles to be overcome in writing a readable and maintainable Ada program is the selection of appropriate identifiers. There are many different kinds of entities to be named -- some with closely related meanings -- and some of the more useful identifiers are excluded as reserved words. The ability of a context clause to introduce a multitude of identifiers without local declarations makes it critical that identifiers be self-documenting.

Among the considerations that a programmer should keep in mind when choosing identifiers are the following:

- Ada places no limit on identifier length, and long identifiers should be used where appropriate.
- An identifier should serve as documentation for an ignorant reader rather than as a mnemonic shorthand for the writer.
- Each identifier should have a unique pronunciation and a unique meaning.
- The spelling of an identifier with a given meaning should be predictable.
- Identifiers should conform to a project-wide convention.

A project-wide convention ought to address at least the following points:

- Common-sense rules of thumb to avoid confusing identifiers. Examples of such rules are:
 - Words in an identifier should be separated by underscore.
 - Identifiers should not hide outer identifiers with the same name, especially if the scope of the inner identifier will include a reference to the outer identifier.
- Rules requiring different kinds of English grammatical structures and different kinds of suffixes for identifiers naming different kinds of entities. Examples are:
 - Procedure and entry names should be imperative clauses.
 - Non-Boolean variables, constants, numbers, and functions should be named by noun phrases.

- Boolean variable and function names should be based on declarative clauses.
 - Task names should be formed by appending the -er suffix to a verb.
 - Type names should end with _TYPE.
 - Record component names should end with _PART, _FIELD, or _COMPONENT.
 - Package names should end with _PACKAGE.
 - Enumeration literals should have suffixes indicating their type.
 - An access type for designating values of type XXX_TYPE should be named XXX_POINTER_TYPE, unless a more abstract name is appropriate.
- Guidelines for forming abbreviations. Examples are:
 - There should be a unique abbreviation for each English word.
 - An abbreviation should not be used unless it is significantly shorter than the word it abbreviates.
 - The meaning of an abbreviation should be clear.
 - Abbreviations of single words should be formed by truncation.
 - In addition to guidelines for forming abbreviations, a project-wide convention might stipulate a list of all approved abbreviations.
 - Rules governing the use of upper and lower case. Examples are:
 - Capitalize reserved words.
 - Use upper case in identifiers only for abbreviations and the beginnings of words.
 - Write single-letter variables in lower case.

The conventions we have presented are not put forth as the conventions that should be followed in forming Ada identifiers. Rather, they are intended as an illustration of the kind of conventions that can be formulated and some of the issues that a convention must address.

The design of an alternative convention for selecting identifiers is a worthwhile exercise. One such alternative can be found in [Gar83], for instance. Another useful exercise is to rewrite part of a program to conform either with the conventions presented here or some other conventions.

The tradition of short identifiers is well-entrenched, and it will not be easily overcome. The tradition may even continue to be reinforced by the file naming restrictions of the programmer's operating system. Sermons about ease of reading being more important than ease of writing are less likely to make an impression than this observation: Text editors allow a programmer to enter a program using short codes for identifiers, and then to quickly change all occurrences of a given code to a long, descriptive identifier. The objection that long identifiers interfere with formatting is a valid one. However, the use of 132-character lines in place of 80-character lines usually more than makes up for the effect of the long identifiers. When identifiers are modeled after English grammatical forms, English stylistic improvements can produce more succinct but equally descriptive identifiers. It may seem cumbersome to expand MAX_NO_RIS_PER_DELIVERY to

Maximum_Number_Of_RIs_Per_Delivery

but RIS_Per_Delivery_Limit is a more succinct way of saying the same thing.

THIS PAGE INTENTIONALLY LEFT BLANK

2.2 Types

This section contains the following case studies:

Discrete Types

Implementation of Set Types

Constant Array Declarations

Record Types

Recursive Type Definitions

THIS PAGE INTENTIONALLY LEFT BLANK

2.2.1 DISCRETE TYPES

1. BACKGROUND

Case Study Objective

To provide examples of discrete types in Ada. To illustrate the use of discrete types in some Ada control structures (if and case statements).

Problem

Ada offers the user many features not available in other programming languages. Users will be unfamiliar with the proper way of using these features, especially in writing embedded software systems. There is a lack of realistic examples of programming techniques in Ada and of the optimal or intended use of Ada's features. For instance, what kinds of data are candidates for representation as enumeration values and in what ways can these or other discrete types be used effectively?

Discussion

This problem is essentially one of unfamiliarity with a new language. It can be remedied by good examples in Ada courses. Underlying this problem is the need for many examples, not only of Ada's advanced features but also of the language's "straightforward" features. This case study will present a set of coding paradigms on discrete types. These examples are neither earthshaking nor esoteric. In fact, some may even appear trivial at first glance; however, the reader should consider them simply as an illustration of a typical segment from an Ada module in a real-time embedded system.

2. DETAILED EXAMPLE

Example Problem Statement

Consider a message switching system. The basic unit of data in this system is a message which is received, decoded and routed. On a less abstract level, a message is simply a sequence of characters. These characters derive meaning from their specific position in this sequence. Let a particular character in every message denote its classification and hence the security of the message. This character is referred to as the prosign of the message. Furthermore, assume that there exists a system-level requirement to examine the security of every entering message. What is the clearest way of decoding the classification of a message? Three solutions to this problem are discussed.

Assume that there are eight known types of security classification: unclassified, encrypted for transmission only, restricted, confidential, secret, top secret, special category, and DSSCS. Assume for the purposes of this exercise that the following enumeration types have been declared earlier:

```
type Security_Classification is
    (Unclassified,
     Encrypted_for_Transmission_Only,
     Restricted,      Confidential,
     Secret,          Top_Secret,
     Special_Category, DSSCS) ;
```

```
type Validity is (Valid, Bad_RI, Bad_LMF, Security_Mismatch) ;
```

Also, assume that the packages containing the procedures Find_Classification and Read_Security_Classification have been imported at a higher level, making these subprograms visible. Further assume that the procedure Find_Classification was intended as a general purpose procedure to find security classifications one or more characters long.

Solution Outline (1) -- if statement

One solution is to use an if statement. (See Detailed Solution (1).) The prosign character is successively tested against the individual legal prosign characters. A match results in the assignment of the corresponding security classification level to the variable Msg_Classification. No match results in the function evaluating to Security_Mismatch and an immediate return to the caller.

Detailed solution (1)

```
function Validate_Security (Msg: MsgID) return Validity is
-- -----
-- Parameters:
--   Msg - access type of a message
-- Global variables:
--   Phys_Line - physical transmission line number (declared
--               in calling program)
-- Local variables:
--   Classification_Error    - true if error obtained while
--                             searching for security prosign
--                             in message header
--   Msg_Classification      - security classification of message
--   Msg_Security_Prosign    - single character prosign of
--                             message
--   Phys_Line_Classification - security classification of
--                             physical transmission line where
--                             current message is to go
-- Subprograms called:
--   Find_Classification      - procedure
--                             Message      : in MsgID
--                             Classification : out
--                             Msg_Classification
--                             Error_Status  : out Boolean
--                             Find and return classification
--                             of message header (declared in
--                             package imported at higher level)
--   Read_Security_Classification - function
--                             Phys_Line : in Physical_Line
--                             Obtain classification of mes-
--                             sage of header (declared in pack-
--                             age imported at higher level)
-- Description: Classification contained in message header is com-
--               pared with that of physical line on which message is
--               being sent. If transmission line's security is less
--               than that of message, then message is invalid.
-- -----

Classification_Error    : Boolean;
Msg_Classification      : Security_Classification;

-- The following object is declared as a string of length 1 (i.e.
-- a one character string) because the procedure Find_Classifi-
-- cation expects a string as an out parameter, not a character.

Msg_Security_Prosign    : String (1 .. 1);
Phys_Line_Classification : Security_Classification;
```

```

begin -- Validate_Security

    -- Fetch classification of message. (Classification returned in
    -- string of length 1.)

    Find_Classification(Msg, Msg_Security_Prosign, Classification_Error);

    if Classification_Error then
        return Security_Mismatch;
    elsif
        -- Continue checking security
        Msg_Security_Prosign(1) = 'U' then
            Msg_Classification := Unclassified;
        elsif
            Msg_Security_Prosign(1) = 'E' then
                Msg_Classification := Encrypted_for_Transmission_Only;
            elsif
                Msg_Security_Prosign(1) = 'R' then
                    Msg_Classification := Restricted;
            elsif
                Msg_Security_Prosign(1) = 'C' then
                    Msg_Classification := Confidential;
            elsif
                Msg_Security_Prosign(1) = 'S' then
                    Msg_Classification := Secret;
            elsif
                Msg_Security_Prosign(1) = 'T' then
                    Msg_Classification := Top_Secret;
            elsif
                Msg_Security_Prosign(1) = 'A' then
                    Msg_Classification := Special_Category;
            elsif
                Msg_Security_Prosign(1) = 'M' then
                    Msg_Classification := DSSCS;
            else
                return Security_Mismatch;
            end if;

    -- Fetch classification of physical line

    Phys_Line_Classification := Read_Security_Classification
                                (Phys_Line);

```

```

-- Compare security classifications of message and transmission
-- line.
-- Transmission line's must be greater than or equal to message's.

if Phys_Line_Classification < Msg_Classification then
    return Security_Mismatch;
else
    return Valid;
end if;
end Validate_Security;

```

This solution does not read well and is somewhat cumbersome. Writing

```

if Msg_Security_Prosign (1) = 'C'

```

(where 'C' is some character) 8 times clutters the page and thereby reduces readability. There is a potential problem in this solution because the variable `Msg_Classification` is not assigned a value when the `prosign` is found to be an illegal character. Under this circumstance, should `Msg_Classification` be used in a subsequent part of the program, then the code would start producing erroneous results. Furthermore, there is no clear demarcation between the processes of fetching the classification of the message and of decoding the classification of the message.

Solution Outline (2) -- case statement

The problem consists of checking a single character against "allowed" characters and returning a value based on this check. The case statement, which tests a variable of a discrete type against a set of distinct values, lends itself well for the solution. This method is presented in the next section, Detailed Solution (2).

Detailed solution (2)

```
function Validate_Security (Msg: MsgID) return Validity is
-- -----
-- Parameters:
--   Msg - access type of a message
-- Global variables:
--   Phys_Line - physical transmission line number (declared in
--               calling program)
-- Local variables:
--   Classification_Error - true if error obtained while
--                           searching for security prosign in
--                           message header
--   Msg_Classification - security classification of message
--   Msg_Security_Prosign - single character prosign of
--                           message
--   Phys_Line_Classification - security classification of
--                           physical transmission line where
--                           current message is to go
-- Subprograms called:
--   Find_Classification - procedure
--                       Message      : in MsgID
--                       Classification : out
--                       Msg_Classification
--                       Error_Status  : out Boolean
--                       Find and return classification
--                       of message header (declared in
--                       package imported at higher level)
--   Read_Security_Classification - function
--                               Phys_Line : in Physical_Line
--                               Obtain classification of mes-
--                               sageheader (declared in package
--                               imported at higher level)
-- Description: Classification contained in message header is com-
--               pared with that of physical line on which message is
--               being sent. If transmission line's security is less
--               than that of message, then message is invalid.
-- -----

Classification_Error : Boolean;
Msg_Classification   : Security_Classification;

-- The following object is declared as a string of length 1 (i.e.
-- a one character string) because the procedure Find_Classifi-
-- cation expects a string as an out parameter, not a character.

Msg_Security_Prosign : String (1 .. 1);
Phys_Line_Classification : Security_Classification;
```

```

begin -- Validate_Security

-- Fetch classification of message. (Classification returned in
-- string of length 1.)

Find_Classification(Msg,Msg_Security_Prosign,Classification_Error);

if Classification_Error then
    return Security_Mismatch;
else -- Continue checking security
    -- Convert message's single character prosign to the enumeration
    -- type of Security_Classification
    case Msg_Security_Prosign (1) is
        when 'U' =>
            Msg_Classification := Unclassified;
        when 'E' =>
            Msg_Classification := Encrypted_for_Transmission_Only;
        when 'R' =>
            Msg_Classification := Restricted;
        when 'C' =>
            Msg_Classification := Confidential;
        when 'S' =>
            Msg_Classification := Secret;
        when 'T' =>
            Msg_Classification := Top_Secret;
        when 'A' =>
            Msg_Classification := Special_Category;
        when 'M' =>
            Msg_Classification := DSSCS;
        when others =>
            return Security_Mismatch;
    end case;

-- Fetch classification of physical line

Phys_Line_Classification := Read_Security_Classification
    (Phys_Line);

-- Compare security classifications of message and transmission
-- line.
-- Transmission line's must be greater than or equal to message's.

if Phys_Line_Classification < Msg_Classification then
    return Security_Mismatch;
else
    return Valid;
end if;
end if;
end Validate_Security;

```

This solution is much more elegant than the code presented in Detailed Solution (1). Like this earlier solution, though, there is the potential problem of the function executing without assigning a value to the variable `Msg_Classification`, as in the others case when the prosign has an illegal value.

Solution Outline (3) -- using an array

An equally interesting alternative solution serves to illustrate arrays indexed by an index of a discrete, nonnumeric type. Specifically, declare a constant array indexed by ASCII characters. The elements of this array are the security classifications of messages. Thus, to find out the classification corresponding to a particular letter code, this array is simply indexed by that letter (a character).

```
Sec_Class: constant array (Character) of Security_Classification :=
    ('U'   => Unclassified,
     'E'   => Encrypted_for_Transmission_Only,
     'R'   => Restricted,      'C' => Confidential,
     'S'   => Secret,          'T' => Top_Secret,
     'A'   => Special_Category, 'M' => DSSCS,
     others => Unknown) ;
```

Notice that this method requires a small modification to the enumeration type definition for `Security_Classification` because a value of that type must be assignable for all the error conditions (illegal character codes which essentially represent no security classification). This modified type declaration reads:

```
type Security_Classification is
    (Unclassified,
     Encrypted_for_Transmission_Only,
     Restricted,      Confidential,
     Secret,          Top_Secret,
     Special_Category, DSSCS,
     Unknown) ;
```

Under these circumstances, the function `Validate_Security` would be coded as shown in Detailed Solution (3).

Detailed solution (3)

```
function Validate_Security (Msg: MsgID) return Validity is
-- -----
-- Parameters:
--   Msg - access type of a message
-- Global variables:
--   Phys_Line - physical transmission line number (declared in
--             calling program)
-- Local variables:
--   Classification_Error - true if error obtained while
--                           searching for security prosign in
--                           message header
--   Msg_Classification - security classification of message
--   Msg_Security_Prosign - single character prosign of
--                           message
--   Phys_Line_Classification - security classification of
--                           physical transmission line where
--                           current message is to go
--   Sec_Class - table of legal and unknown security
--               classifications
-- Subprograms called:
--   Find_Classification - procedure
--                       Message      : in MsgID
--                       Classification : out
--                       Msg_Classification
--                       Error_Status  : out Boolean
--                       Find and return classification
--                       of message header (declared in
--                       package imported at higher level)
--   Read_Security_Classification - function
--                               Phys_Line : in Physical_Line
--                               Obtain classification of message
--                               header (declared in package
--                               imported at higher level)
-- Description: Classification contained in message header is com-
--               pared with that of physical line on which message is
--               being sent. If transmission line's security is less
--               than that of message, then message is invalid.
-- -----
```

```

Sec_Class: constant array (Character) of Security_Classification :=
    ('U' => Unclassified,
     'E' => Encrypted_for_Transmission_Only,
     'R' => Restricted,      'C' => Confidential,
     'S' => Secret,         'T' => Top_Secret,
     'A' => Special_Category, 'M' => DSSCS,
     others => Unknown);
Classification_Error : Boolean;
Msg_Classification   : Security_Classification;

-- The following object is declared as a string of length 1 (i.e.
-- a one character string) because the procedure Find_Classifi-
-- cation expects a string as an out parameter, not a character.

Msg_Security_Prosign : String (1 .. 1);
Phys_Line_Classification : Security_Classification;

begin -- Validate_Security

    -- Fetch classification of message. (Classification returned in
    -- string of length 1.)

    Find_Classification (Msg,Msg_Security_Prosign,Classification_Error);

    if Classification_Error then
        return Security_Mismatch;
    else -- Continue checking security
        Msg_Classification := Sec_Class (Msg_Security_Prosign (1));
    end if;

    if Msg_Classification = Unknown then
        return Security_Mismatch;
    end if;

    -- Fetch classification of physical line

    Phys_Line_Classification := Read_Security_Classification (Phys_Line);

    -- Compare security classifications of message and transmission
    -- line.
    -- Transmission line's must be greater than or equal to message's.

    if Phys_Line_Classification < Msg_Classification then
        return Security_Mismatch;
    else
        return Valid;
    end if;

end Validate_Security;

```


3. EPILOGUE

Discrete types may be classified as numeric and nonnumeric types. Nonnumeric types consist of enumeration types. (The predefined types Character and Boolean are special instances of enumeration types.) All share the following properties:

- a first and last value
- a finite number of values
- an exact representation for each value
- an ordered set of values, such that the repeated application of the attribute 'Succ gives the progression from first to last value, and such that the repeated application of the attribute 'Pred gives the reverse progression, from the last to the first value
- definition of assignment, membership and relational operators

Numeric types have an additional property: they are defined under arithmetic operations.

The three examples discussed in this case study show instances of using a nonnumeric discrete type in place of a numeric discrete type (i.e. an integer type). In each case, it was shown that what could be done with the integer type could be done just as well if not better by the enumeration or character type. Below is a list of the most common uses for nonnumeric discrete types:

- assignment
-- e.g. Msg_Classification := Top_Secret ;
- relational operation
-- e.g. Phys_Line_Classification < Msg_Classification
- array index
-- e.g. Sec_Class ('A')
- array element
-- e.g. Sec_Class: array (Character) of
Security_Classification ;
- for loop range
-- e.g. for Char in 'A' .. 'Z' loop
- membership test
-- e.g. Classification in Security_Classification
- case statement
-- e.g. case Msg_Security_Prosign (1) is
- with attributes First, Last, Succ, Pred
-- e.g. Validity'Succ (Bad_RI)
- subprogram parameter

- value returned by a function
-- e.g. return Security_Mismatch ;
- discriminant
- entry family index

Missing from the above list is arithmetic. Ada forbids the user to add two characters, for instance, or to multiply the enumeration literal Secret by any value whatsoever. (The user may counter these restrictions by overloading the arithmetic operators; however, for the purposes of this case study, assume that the operators are not overloaded.) While these examples are indeed a little far-fetched, there is an issue worth closer examination. Suppose a character must be converted from lower case to upper case. In FORTRAN, the programmer would simply subtract a fixed offset, 32, from the variable corresponding to the lower case character to compute the ASCII value for the equivalent upper case character. In Ada, the programmer cannot legally write the expression:

Char - 32

How does an Ada programmer accomplish this conversion? An equivalent expression can be derived in Ada using the attributes VAL and POS:

Character'Val (Character'Pos (Char) - 32)

The attribute Pos returns an integer which represents the position of Char in the sequence of ASCII characters which are defined by the declaration of the type Character. The attribute Val returns the character whose position in the ASCII sequence is given by Val's argument. The syntax may look cumbersome to the uninitiated reader; however, it forces the programmer to "intend" the subtraction of 32 from Char.

In other languages without enumeration types, integer is the only discrete type available to the programmer. Consequently, all discrete data are represented as integers, necessitating encoding the data as or converting them to integers. This programming technique is unnecessary and inappropriate in Ada. It is important to realize that the variety of discrete types in Ada offers the potential for writing elegant, readable and maintainable programs, and that these types should be used to their best advantage. This is the fundamental point that this case study has endeavored to illustrate.

It is left as an exercise for the reader to develop Detailed Solution (3) with an array of Security_Classification indexed from 'A' to 'Z' instead of indexed over the entire range of ASCII characters. Handle the possibility that the character returned by Find_Class in Msg_Security_Prosign might be outside this subrange. (Hint: in-line code is much preferable to an exception handler here.)

2.2.2 IMPLEMENTATION OF SET TYPES

1. BACKGROUND

Case Study Objective

To describe how Pascal set types may be simulated in Ada. To illustrate the use of private types and generic packages.

Problem

Pascal provides set types, whose values are all possible subsets of the values in some simple type. For example, the Pascal type definitions

```
PrimaryColor = (Red, Green, Blue);  
Color = set of PrimaryColor;
```

establish a type Color with one element corresponding to each of the sets {}, {Red}, {Green}, {Blue}, {Red, Green}, {Red, Blue}, {Green, Blue}, and {Red, Green, Blue}. The operations +, specifying set union, *, specifying set intersection, and -, specifying set difference, can be applied to operands in set types. The Pascal set membership operator in (not to be confused with the Ada subtype membership operator of the same name) can be used to determine whether a given value of type PrimaryColor is an element of a given set value of type Color. Finally, a set can be constructed by listing its elements between square brackets, separated by commas, as in [1, 3, 5].

Discussion

Sets of values belonging to some discrete type can be a very useful data abstraction tool. However, Ada does not provide set types as part of the language. Fortunately, there is a straightforward way to implement a comparable facility based on the primitives that Ada does provide.

2. DETAILED EXAMPLE

Example Problem Statement

A message switch handles messages exchanged among three classes of organizations, known as the "R", "U", and "Y" communities. A given message may be destined for recipients in more than one community. The processing of a message depends on the set of communities for which it is destined.

The message switch program includes a type for representing sets of communities, declared as follows:

```
type Community_Set_Type is
  ( Community_Set_R, Community_Set_U, Community_Set_Y,
    Community_Set_RU, Community_Set_RY, Community_Set_UY,
    Community_Set_RUY );
```

This declaration is extremely inconvenient, and is at odds with the simplicity of the abstraction it is representing -- subsets of the set {R, U, Y}. It does not allow a simple implementation of typical set operations, such as taking the union of two sets or determining whether a set includes a particular community. For example, if `Destination_Communities` is a `Community_Set_Type` variable and we want to perform some action `S` if and only if `Destination_Communities` includes the "R" community, the simplest formulation is as follows:

```
case Destination_Communities is
  when Community_Set_R | Community_Set_RU | Community_Set_RY |
    Community_Set_RUY =>
    S;
  when others =>
    null;
end case;
```

As inconvenient as this implementation of sets is when dealing with a universe of three elements, it becomes completely impractical when the number of elements becomes even slightly higher.

Solution Outline

Though Ada does not provide set types directly, it does provide all the primitives necessary to implement them simply and efficiently. Assuming that the universe from which sets are formed consists of the elements of some discrete type, a set type can be implemented as an array of Boolean values indexed by the values of the discrete type. Union, intersection, and set difference can be implemented directly in terms of the predefined logical operators for Boolean arrays; set

membership can be determined simply by examining an array element; and set values can be denoted by array aggregates.

Implementation details can be hidden from the user of the implemented type by defining the type in a package with a private part. Furthermore, since set types are a common abstraction, this package can be made generic and instantiated for any discrete type to obtain a type consisting of the subsets of that discrete type's values. Given this generic package, a programmer can indeed regard set types as a primitive feature available to him as readily as the input and output facilities of the language.

Detailed Solution

The declaration for `Community_Set_Type` as an enumeration type describes sets of communities without ever referring to communities themselves. A more rational starting point would be the following type declaration:

```
type Community_Type is (R_Community, U_Community, Y_Community);
```

Given this building block, the type `Community_Set_Type` can be implemented with the following data structure:

```
type Community_Set_Type is array (Community_Type) of Boolean;
```

Then a `Community_Set_Type` value has one Boolean component corresponding to each possible member of a set of `Community_Type` values. A set of `Community_Type` values is represented by a `Community_Set_Type` array in which each component corresponding to a member of the set has the value `True` and every other component has the value `False`.

The union, intersection, and set difference operations are then provided directly by the predefined logical operations for Boolean arrays. The logical operators `and`, `or`, and `xor` can be applied to two equal-length one-dimensional arrays of Boolean values to produce an array of the same length. The components of this result are obtained by applying the logical operation to the two operands component by component. Suppose `A` and `B` are objects of type `Community_Set_Type`. Since a `Community_Type` value is in the union of `A` and `B` when it is in `A`, in `B`, or both, the expression

`A or B`

computes the union of `A` and `B`. Since a `Community_Type` value is in the intersection of `A` and `B` if and only if it is in `A` and also in `B`, the expression

`A and B`

computes the intersection of A and B. Since a `Community_Type` value is in the set difference of A and B if and only if it is in A but not in B, the expression

A and not B

computes the set difference of A and B.

Membership in a set can be tested simply by examining the component of a `Community_Set_Type` array indexed by a particular `Community_Type` value. Suppose `Destination_Communities` is a `Community_Set_Type` variable. The following statement calls procedure S if and only if `R_Community` is an element of `Destination_Communities`:

```
if Destination_Communities(R_Community) then
  S;
end if;
```

Since a `Community_Set_Type` value is an array, it can be described by an aggregate. For example, the aggregate

(`R_Community => True`, others => `False`)

describes the singleton set {`R_Community`}. In general, if `Element_Type` is some discrete type, `Set_Type` is an array type indexed by `Element_Type` with Boolean components, and `L1`, `L2`, ..., `Ln` are `Element_Type` enumeration literals, the aggregate

(`L1 | L2 | ... | Ln => True`, others => `False`)

describes the set {`L1`, `L2`, ..., `Ln`}.

Unfortunately, this is not quite as flexible as the Pascal set constructor []. Pascal allows a set expression of the form [`E1`, `E2`, ..., `En`], where `E1`, `E2`, ..., `En` are arbitrary expressions. In an Ada aggregate of the form

(`E1 | E2 | ... | En => True`, others => `False`),

`E1`, `E2`, ..., `En` must be static expressions. To remedy this problem we can introduce a new type `Community_List_Type` and a new function `Set_Of`. `Community_List_Type` is an unconstrained array of `Community_Type` components:

```
type Community_List_Type is
  array (Positive range <>) of Community_Type;
```

The function `Set_Of` takes a `Community_List_Type` parameter and returns the `Community_Set_Type` value whose elements are the components of the parameter:

```

function Set_Of (Elements : Community_List_Type)
    return Community_Set_Type is

    Result : Community_Set_Type := (Community_Type => False);

begin -- Set_Of

    for E in Elements'Range loop
        Result( Elements(E) ) := True;
    end loop;

    return Result;

end Set_Of;

```

Set_Of can be called with an array aggregate as a parameter, as in Set_Of((1 => R_Community)) and Set_Of((R_Community, U_Community)). (The outer parentheses delimit the parameter list and the inner parentheses are part of a positional aggregate of Community_List_Type. This aggregate is the only parameter in the parameter list.) Since the expressions in a positional aggregate need not be static, a call like Set_Of((X, Y)), where X and Y are Community_Type variables, is also possible. No problems arise if X and Y have the same value. Set_Of((X, Y)) is simply equal to Set_Of((1 => X)) in such a circumstance.

Although set operations can be implemented almost directly in terms of the predefined Ada operations, the notation is not natural for sets. (This is especially true of the notation for set membership tests.) Furthermore, the programmer using sets is acutely aware of their internal implementation. These problems can be overcome by placing the data type declarations and procedures used to implement sets in a package with a private part. By making Community_Set_Type private, we enforce separation of concerns. The programmer using sets is unconcerned with their implementation. The notation he used is based on the abstraction with which he is dealing, not its underlying representation.

The following formulation is based on the view that, as in Pascal, the operators +, *, and - are appropriate names for union, intersection, and set difference operators. A reader who disagrees may substitute function names Union, Intersection, and Set_Difference for "+", "*", and "-". Community_Type is presumed to be declared outside the package, as described at the beginning of the Detailed Solution.

```

package Community_Set_Package is

    type Community_Set_Type is private;

    type Community_List_Type is
        array (Positive range <>) of Community_Type;

    function "+" (Left, Right: Community_Set_Type)
        return Community_Set_Type;

    function "*" (Left, Right: Community_Set_Type)
        return Community_Set_Type;

    function "-" (Left, Right: Community_Set_Type)
        return Community_Set_Type;

    function Element_Of
        (Object: Community_Type;
         Set   : Community_Set_Type)
        return Boolean;

    function Set_Of (Elements: Community_List_Type)
        return Community_Set_Type;

private

    type Community_Set_Type is array (Community_Type) of Boolean;

end Community_Set_Package;

```

```

package body Community_Set_Package is

    function "+" (Left, Right: Community_Set_Type)
        return Community_Set_Type is
    begin
        -- "+"
        return Left or Right;
    end "+";

    function "*" (Left, Right: Community_Set_Type)
        return Community_Set_Type is
    begin
        -- "*"
        return Left and Right;
    end "*";

    function "-" (Left, Right: Community_Set_Type)
        return Community_Set_Type is
    begin
        -- "-"
        return Left and not Right;
    end "-";

    function Element_Of
        (Object: Community_Type;
         Set   : Community_Set_Type)
        return Boolean is
    begin
        -- Element_Of
        return Set(Object);
    end Element_Of;

    function Set_Of (Elements: Community_List_Type)
        return Community_Set_Type is

        Result : Community_Set_Type := (Community_Type => False);

    begin
        -- Set_Of

        for E in Elements'Range loop
            Result( Elements(E) ) := True;
        end loop;

        return Result;

    end Set_Of;

end Community_Set_Package;

```

Once Community_Set_Type is made private, it is no longer possible to write a Community_Set_Type aggregate outside of Community_Set_Package. Sets can only be built out of elements by calling Set_Of. Unlike Community_Set_Type, Community_List_Type is not private. Our intention is that users of Community_Set_Package will exploit the representation of Community_List_Type to write positional aggregates as parameters to

Set_Of.

Community_Set_Package is a good solution to the problem of implementing sets of Community_Type values. Nonetheless, the prospect of writing a similar package every time we wish to define a new set type is not pleasant. For this reason, it would make more sense to write a generic package that could be instantiated with any discrete type to produce a type consisting of sets of values in that discrete type. The generic package specification would look like this:

```
generic

    type Element_Type is ( <> );

package Set_Package is

    type Set_Type is private;

    type Element_List_Type is
        array (Positive range <>) of Element_Type;

    function "+" (Left, Right: Set_Type) return Set_Type;

    function "*" (Left, Right: Set_Type) return Set_Type;

    function "-" (Left, Right: Set_Type) return Set_Type;

    function Element_Of
        (Object: Element_Type;
         Set: Set_Type)
        return Boolean;

    function Set_Of (Elements: Element_List_Type)
        return Set_Type;

private

    type Set_Type is array (Element_Type) of Boolean;

end Set_Package;
```

The generic body of Set_Package is identical to the Community_Set_Package body except for the names of the types and the name of the package itself. The generic parameter is required to be a discrete type because it is used as an index subtype in the full declaration of Set_Type.

Given this generic package, the package Community_Set_Package could be replaced by the following generic instantiation:

```
package Community_Set_Package is  
  new Set_Package (Element_Type => Community_Type);
```

Similarly, given the integer type declaration

```
type Digit_Type is range 0 .. 9;
```

we could write

```
package Digit_Set_Package is  
  new Set_Package (Element_Type => Digit_Type);
```

Thus set types become almost as easy to create and use in Ada as in Pascal.

3. EPILOGUE

Pascal, as defined in [J&W74], allows set types to have elements in any "simple type," including the Pascal type Real. The Ada Set_Package only accepts discrete types as actual parameters, so Element_Type may not be a floating point type or fixed point type. Allowing real numbers as set elements requires a completely different implementation, and many Pascal compilers do not implement it.

Pascal also allows a set expression to contain ranges, as in ['A' .. 'Z', 'a' .. 'z']. If Ada set types are implemented as originally described, without a private type declaration, and if the bounds of the ranges are all static, Ada named aggregates like

```
('A' .. 'Z' | 'a' .. 'z' => True, others => False)
```

can be used in this way. Otherwise, if we want to provide this capability, we must provide a new function like the following:

```
function Element_Range (Low, High: Element_Type) return Set_Type is
    Result : Set_Type := (Element_Type => False);
begin
    -- Element_Range
    for E in Low .. High loop
        Result(E) := True;
    end loop;

    return Result;
end Element_Range;
```

This only accommodates a single range, so, assuming the instantiation

```
package Char_Set_Package is
    new Set_Package (Element_Type => Character);
```

we have to write

```
Char_Set_Package.Element_Range('A' .. 'Z') +
    Char_Set_Package.Element_Range('a' .. 'z')
```

to achieve the effect of the aggregate above.

It might also be useful for Set_Package to provide a facility for iterating over the elements of a set. The generic package declaration would contain the following generic procedure declaration:

```

generic
    with procedure Process_Element (Element: in Element_Type);
    procedure Process_Each_Element (Set: in Set_Type);

```

The package body would contain the following procedure body:

```

procedure Process_Each_Element (Set: in Set_Type) is

begin -- Process_Each_Element

    for E in Element_Type loop
        if Set(E) then
            Process_Element (E);
        end if;
    end loop;

end Process_Each_Element;

```

Assume that we have instantiated Set_Package as follows:

```

type Digit_Type is range 0 .. 9;

package Digit_Set_Package is
    new Set_Package (Element_Type => Digit_Type);

```

Process_Each_Element could be used as follows to compute the sum of the digits in a set of digits:

```

subtype Digit_Set_Type is Digit_Set_Package.Set_Type;

function Sum_Of_Digits
    (Digit_Set : in Digit_Set_Type)
    return Integer is

    Sum : Integer := 0 ;

    procedure Add_To_Sum (Digit : in Digit_Type) is
    begin -- Add_To_Sum
        Sum := Sum + Integer (Digit);
    end Add_To_Sum;

    procedure Accumulate_Sum is
        new Process_Each_Element (Process_Element => Add_To_Sum);

begin --Sum_Of_Digits

    Accumulate_Sum;
    return Sum;

end Sum_Of_Digits;

```

Add_To_Sum references Sum as a non-local variable because the generic formal parameter of Process_Each_Element must be a procedure taking one mode in parameter belonging to the set element type.

Process_Each_Element is not strictly necessary because it can be implemented outside of the package using the primitives provided by the package. In fact, the implementation outside of the package would be essentially identical to that described above. Nonetheless, it is easy to envision representations of sets (such as by linked lists of elements) for which an implementation exploiting the internal representation could be substantially more efficient than iterating over each value in the element type and testing for set membership. The purpose of providing Process_Each_Element as one of the facilities of Set_Package is to decouple the use of the package from any assumptions about its implementation.

Set_Package could also provide the following generic function, similar in function to Process_Each_Element:

generic

with function Element_Mapping_Function
 (Element: Element_Type)
 return Element_Type;

function Map_Set (Set : in Set_Type) return Set_Type;

The result of the function call Map_Set (S) would be a set containing an element Element_Mapping_Function (X) for each element X of S. The body of Map_Set is left as an exercise for the reader.

2.2.3 CONSTANT ARRAY DECLARATIONS

1. BACKGROUND

Case Study Objective

To illustrate issues arising in the declaration of arrays to be used as tables.

Problem

An Ada programmer has many options when declaring an array object. He can declare an array object belonging to an anonymous array type, or he can declare an array type and then declare the object to belong to that type. The array type declaration may be constrained or unconstrained. The array may be a variable or a constant. If it is a variable, the declaration may, but need not, specify an initial value. The initial value can be written either as a positional aggregate, a named aggregate, or, in some cases, a string literal — or as some combination of these joined by catenation. If the array variable is declared in the visible part of a package specification, it may be initialized by the statements of the package body. The programmer should understand when each of these options is appropriate.

Discussion

This case study deals with two different problems. The first is the specification of strings to be used as initial values. The second is the declaration of an array to be used as a read-only table. In each case there are many choices to be made. These choices can affect the likelihood of errors in the program, the readability of the program, the ease with which the program can be modified, and the amount of recompilation that will be necessary when the program is modified.

2. DETAILED EXAMPLES

Example Problem Statement (1)

The various messages transmitted by a message switch include certain special sequences of characters. The program controlling the message switch contains constants holding these character sequences. Usually, sequences of characters are specified by string literals. However, the special sequences occurring in message transmissions are repetitious patterns of printing and non-printing characters. The non-printing characters rule out the use of pure string literals. In any event, string literals may not be the clearest way to convey the patterns of characters.

Every transmission contains a 28-character transmission identifier whose form depends on the character set being used for the transmission. For an ASCII transmission, the transmission identifier consists of:

- eight NUL characters
- six DEL characters
- the characters "VZCZC"
- three characters giving the channel designator
- three characters giving the channel sequence number
- two carriage returns
- a line feed

For an ITA transmission, the transmission identifier consists of:

- six NUL characters
- six SI characters
- the characters "VZCZC"
- three characters giving the channel designator
- one SO character
- three characters giving the channel sequence number
- an SI character

- two carriage returns
- a line feed

In addition to the transmission identifier, a transmission contains a CANTRAN sequence, an EOM (end of message) sequence, and a trailer. The CANTRAN sequence consists of:

- eight repetitions of the pair of characters capital E, space
- capital A
- capital R

The EOM sequence consists of:

- two carriage returns
- eight line feeds
- four capital N's

A trailer consists of twelve DEL characters for ASCII transmissions and twelve SI characters for ITA transmissions.

Solution Outline (1)

Since String values are just arrays of Character values, the alternatives to string literals are positional aggregates and named aggregates. Different parts of strings can be described in different ways and joined by the catenation operator &. This operator may be applied either to strings or to individual characters.

Detailed Solution (1)

The message switch program contains a two-element array Frame_Chars with one element containing the "frame" for the ASCII transmission identifier and one element containing the "frame" for the ITA transmission identifier. In the "frame" of a transmission identifier, the channel designator is replaced by the characters "xxx" and the channel sequence number by the characters "yyy". There are also strings named CANTRAN_Sequence and EOM_Sequence, and a two-element array of strings named Trailer, containing the other character sequences described in the Example Problem Statement.

The following declarations establish these arrays:

```

use ASCII;
type Char_Set_Type is (ASCII_Set, ITA_Set, Any_Set);
TI_Length      : constant := 28;
EOM_Length     : constant := 14;
Trailer_Length : constant := 12;
subtype Frame_String is String (1 .. TI_Length);
subtype Trailer_String is String (1 .. Trailer_Length);

Frame_Chars : constant array (ASCII_Set..ITA_Set) of Frame_String :=
    (ASCII_Set =>
        (NUL, NUL, NUL, NUL, NUL, NUL, NUL, NUL, DEL, DEL,
         DEL, DEL, DEL, DEL, 'V', 'Z', 'C', 'Z', 'C', 'x',
         'x', 'x', 'y', 'y', 'y', CR, CR, LF),
     ITA_Set =>
        (NUL, NUL, NUL, NUL, NUL, NUL, SI, SI, SI, SI, SI,
         SI, 'V', 'Z', 'C', 'Z', 'C', 'x', 'x', 'x', SO,
         'y', 'y', 'y', SI, CR, CR, LF) );

CANTRAN_Seq : constant String := "E E E E E E E E AR";

EOM_Seq      : constant String (1 .. EOM_Length) :=
    CR&CR & LF&LF&LF&LF&LF&LF&LF&LF & "NNNN";

Trailer      : constant array (ASCII_Set .. ITA_Set) of
    Trailer_String :=
    (ASCII_Set => (1 .. Trailer_Length => DEL),
     ITA_Set =>  (1 .. Trailer_Length => SI) );

```

These declarations exhibit a wide variety of ways to describe string values. The initial value of each element of `Frame_Chars` is specified by a positional aggregate listing each of the characters in the string. Printing characters are specified using character literals and non-printing identifiers are specified using the names of the constants declared in package `Standard.ASCII`. The initial value of `CANTRAN_Seq` is given by a simple string literal. The initial value of `EOM_Seq` is written as a catenation of character values and a string literal. The character values are specified using the `Standard.ASCII` constants since they correspond to non-printing characters. The initial values of the two elements of `Trailer` are given by named aggregates.

String literals can only be used when a string consists entirely of printing characters. A string literal is equivalent to a positional aggregate listing individual character values. When the presence of control characters rules out string literals, positional aggregates can be used instead, as with both elements of the initial value of `Frame_Chars`. Similarly, the initial value of `CANTRAN_Seq` could have been written as

```

('E', ' ', 'E', ' ', 'E', ' ', 'E', ' ',
 'E', ' ', 'E', ' ', 'E', ' ', 'A', 'R')

```

Strings containing messages to be read by people are most easily and readably written as string constants; strings that are only interpreted by machine as a sequence of individual characters may be more naturally represented as a list of individual character values. Since it is often difficult to count the number of consecutive spaces occurring in a string constant, a list of individual characters is less error prone when the exact composition of a string is more important than its general appearance. In the declaration of EOM_Seq, spacing around the & operator is used to divide the string visually into a sequence of carriage returns, a sequence of line feeds, and a sequence of capital N's. This partition could have been expressed more strongly with positional aggregates, as follows:

(CR, CR) & (LF, LF, LF, LF, LF, LF, LF, LF) & "NNNN"

In general, a positional aggregate is more appropriate than catenation for expressing a sequence of character values, since a positional aggregate names the sequence directly, while catenation describes the construction of the sequence from smaller components. (In this sense, the aggregate (CR, CR) is to the compound expression CR & CR as the integer literal 2E3 is to the compound expression 2 * 10 ** 3.)

The use of positional aggregates would clearly be inappropriate for the elements of the initial value of Trailer. Each element would be represented by an aggregate listing the same character value twelve times. As the declaration of Trailer shows, named aggregates are an effective way of expressing repetition. Named aggregates also make the pattern of a character string more explicit. The other declarations above could be rewritten with named aggregates as follows:

```

Frame_Chars : constant array (ASCII_Set..ITA_Set) of Frame_String :=
  (ASCII_Set =>
    (1 .. 8   => NUL,
     9 .. 14  => DEL,
     15       => 'V',
     16 | 18  => 'Z',
     17 | 19  => 'V',
     20 .. 22 => 'x',
     23 .. 25 => 'y',
     26 .. 27 => CR,
     28       => LF),
   ITA_Set =>
    (1 .. 6   => NUL,
     7 .. 12  => SI,
     13       => 'V',
     14 | 16  => 'Z',
     15 | 17  => 'C',
     18 .. 20 => 'x',
     21       => SO,
     22 .. 24 => 'y',
     25       => SI,
     26 .. 27 => CR,
     28       => LF ) );

CANTRAN_Seq : constant String :=
  (1 | 3 | 5 | 7 => 'E',
   2 | 4 | 6 | 8 => 'I',
   9             => 'A',
   10            => 'R');

EOM_Seq      : constant String (1 .. EOM_Length) :=
  (1 .. 2   => CR,
   3 .. 10  => LF,
   11 .. 14 => 'N');

```

This notation provides a precise description of the position of each character in the string. Sometimes this is very useful. For instance, the code to create transmission identifiers makes a copy of `Frame_Chars (ASCII_Set)` or `Frame_Chars (ITA_Set)` and replaces the lower case x's and y's of that copy by a channel designator and a channel sequence number, respectively. This code refers to the starting and ending positions of the sequence of x's and sequence of y's. The declaration of `Frame_Chars` using named notation clearly documents those positions.

More commonly, the programmer thinks in terms like "two carriage returns, followed by eight line feeds, followed by four capital N's". Not only are the starting and stopping positions for a given character value irrelevant in such cases; trying to specify them introduces an opportunity for errors. Fortunately, catenation allows the programmer

to disregard the exact position of each character in the string. The index bounds of catenation operands are, in essence, converted automatically to the index bounds required in the result, so a run of N consecutive occurrences of the character C can be written as follows:

(1 .. N => C)

Thus the declaration of EOM_Seq can be rewritten so that its initial value reflects the way we think about it, as follows:

```
EOM_Seq : constant String (1 .. EOM_Length) :=  
          (1 .. 2 => CR) & (1 .. 8 => LF) & (1 .. 4 => 'N')
```

(Because we are dealing with so few repetitions, the segment (1 .. 2 => CR) can be written with fewer characters as (CR, CR), and (1 .. 4 => 'N') as "NNNN". For a larger number of repetitions, the named aggregate quickly becomes more succinct.)

Example Problem Statement (2)

A message switch contains a table of possible transmission line baud rates in ascending order. The declaration occurs in the visible part of a package named Transmission_Line_Package. Transmission_Line_Package is referenced by many other packages. It is reasonable to expect that the message switch system will be modified in the future to accommodate new baud rates.

Solution Outline (2)

All things being equal, a table is most simply and appropriately declared as a constant array belonging to an anonymous array type. However, if a table is likely to be modified, there are other issues to consider. One is the number of places the program text must be changed. Another is the number of compilation units which must be recompiled once the change is made.

First let us consider the problem of textual modification. To add elements to an array belonging to an anonymous array type, it is necessary to change both the index constraint and the initial value. If the array is declared as a member of an unconstrained array type instead, the index constraint can be eliminated because the array is constant. Sometimes a table is most easily specified in terms of an algorithm to compute its contents. Besides making the initial specification of the table easier and less error-prone, this approach makes it very easy to change the table when the only change is to enlarge the table using the same algorithm. If the array is to be declared constant, this algorithm must be implemented as a function called in the initial value specification of the array declaration.

At least with relatively small arrays, however, the cost of recompilation is a greater source of concern than the cost and reliability of textual modifications. The Example Problem Statement specifies that the table of baud rates is declared in the visible part of a package referenced by many other packages. Modification to this visible part will require recompilation of the corresponding package body as well as of all these other packages. However, recompilation costs can be reduced by moving information about the size and contents of the table out of the package specification in which they occur and into a package body.

Detailed Solution (2)

There can only be one object belonging to a given anonymous array type. Thus an object in an anonymous array type cannot be assigned or passed as a parameter. This makes anonymous array types appropriate for one-of-a-kind objects that will be shared by different parts of a program. Tables usually fall into this category.

If the values in the table are fixed rather than updated by the program, the array object should be declared as a constant. Besides documenting the fact that the array's value will not change, a constant declaration prevents accidental updating of the array. A constant declaration must contain an initial value.

Assuming that Baud_Rate_Type is a fixed point type, the declaration of the baud rate list could look like this:

```
Baud_Rate_List: constant array ( 1 .. 12 ) of Baud_Rate_Type :=  
    ( 45.0, 50.0, 56.9, 74.2, 75.0, 150.0, 300.0,  
      600.0, 1200.0, 2400.0, 4800.0, 9600.0 );
```

Unfortunately, this solution fails to take into account the likelihood that the table will be modified. If the most likely modification were the replacement of one baud rate with another, the change could be accomplished simply by changing one of the values in the initial value aggregate. However, the more likely modification is the addition of a new baud rate to the list. This requires changing both the aggregate and the upper bound of the index constraint.

The table can be made more easily modifiable by using an unconstrained array subtype. Although the declaration of any variable in such a subtype must contain an index constraint, the array bounds of a constant array can be inferred from the initial value. Thus the index constraint may be omitted. Unconstrained array subtypes may not be anonymous, so the declaration of Baud_Rate_List must be replaced with the following two declarations:

```

type Baud_Rate_List_Type is
  array ( Positive range <> ) of Baud_Rate_Type;

```

```

Baud_Rate_List: constant Baud_Rate_List_Type :=
  ( 45.0, 50.0, 56.9, 74.2, 75.0, 150.0, 300.0,
    600.0, 1200.0, 2400.0, 4800.0, 9600.0 );

```

When the table is large and its contents can be computed systematically, it is often preferable to write statements to initialize the table. This is both easier and less error-prone than listing all its contents in an aggregate. Baud_Rate_List is too small to benefit from this approach. However, its contents can be computed systematically, since, after the first five baud rates, each baud rate in the list is twice the preceding one. Thus Baud_Rate_List can be used to illustrate initialization by statements even though this approach is most useful for much larger arrays.

If the following statements are placed in the Transmission_Line_Package body, they will be executed when the package specification is elaborated:

```

Baud_Rate_List (1 .. 5) := (45.0, 50.0, 56.9, 74.2, 75.0);

for I in 6 .. Baud_Rate_List'Last loop
  Baud_Rate_List (I) := 2 * Baud_Rate_List (I - 1);
end loop;

```

Of course Baud_Rate_List cannot be declared constant if this approach is taken. An alternative is to place the computation of the table in a function, say Baud_Rates_Computation. Then Baud_Rate_List can be declared as follows:

```

Baud_Rate_List : constant Baud_Rate_List_Type :=
  Baud_Rates_Computation;

```

Unfortunately, this apparently simple solution has a nasty complication. Where is the function Baud_Rates_Computation to be declared? It is an error for Baud_Rates_Computation to be called before its function body is elaborated. Thus the function body must textually precede the declaration of Baud_Rate_List. Since Baud_Rate_List is declared in a package specification, the only way to achieve this is to compile the function before Transmission_Line_Package. Since the specification of Baud_Rates_Computation refers to the declaration of Baud_Rate_List_Type, this entails moving both the type declaration for Baud_Rate_List_Type and the function declaration for Baud_Rates_Computation into a separately compiled package:

```

package Baud_Rate_Definitions is

    Baud_Rate_List_Size : constant := 12;

    type Baud_Rate_Type is delta 0.1 range 0.0 .. 100_000.0;
    type Baud_Rate_List_Type is
        array (1 .. Baud_Rate_List_Size) of Baud_Rate_Type;
    function Baud_Rates_Computation return Baud_Rate_List_Type;

end Baud_Rate_Definitions;

package body Baud_Rate_Definitions is

    function Baud_Rates_Computation return Baud_Rate_List_Type is

        Result : Baud_Rate_List_Type;

    begin -- Baud_Rates_Computation

        Result (1 .. 5) := (45.0, 50.0, 56.9, 74.2, 75.0);

        for I in 6 .. Baud_Rate_List_Size loop
            Result (I) := 2 * Result (I - 1);
        end loop;

        return Result;

    end Baud_Rates_Computation;

end Baud_Rate_Definitions;

-- -- -- -- --

with Baud_Rate_Definitions;

package Line_Transmission_Package is

    subtype Baud_Rate_List_Type is
        Baud_Rate_Definitions.Baud_Rate_List_Type;

    Baud_Rate_List : constant Baud_Rate_List_Type :=
        Baud_Rate_Definitions.Baud_Rates_Computation;

    ...

end Line_Transmission_Package;

```


Computing rather than listing the table entries makes the table very easy to modify — provided that all modifications made later conform to the algorithm. To add baud rates 19200.0 and 38400.0 to Baud_Rate_List, it is only necessary to change the constant Baud_Rate_List_Size from 12 to 14. (The length of the array is specified by a constant declaration at the top of the package body in order to facilitate this kind of change.) In contrast, substantial modifications to Baud_Rates_Computation would be necessary to add the baud rate 10000.0 to the list. Algorithmic specification of table values makes it easier to add new table values that follow the algorithm, but much harder to add new table values that do not. (Again, Baud_Rate_List is a very small array providing a scaled-down illustration of initialization by algorithm. The true impact of algorithmic initialization is felt when expanding a 100-element array to hold 200 elements, for instance.)

Unfortunately, it is not nearly as easy to recompile this program as it is to modify its text. The Baud_Rate_Definitions package specification must be recompiled after Baud_Rate_List_Size is changed. This, in turn, requires that the body of Baud_Rate_Definitions and the specification of Transmission_Line_Package be recompiled. But then all units referencing the Transmission_Line_Package specification (including the Transmission_Line_Package body) must be recompiled, then the units referencing those units, and so forth.

In general, recompilation requirements can be minimized by moving information likely to be changed out of package specifications and into package bodies. In this case, the declaration of Baud_Rate_List_Size can be moved into the Baud_Rate_Definitions body by making Baud_Rate_List_Type unconstrained:

```
package Baud_Rate_Definitions is

  type Baud_Rate_Type is delta 0.1 range 0.0 .. 100_000.0;
  type Baud_Rate_List_Type is
    array ( Positive_range <> ) of Baud_Rate_Type;
  function Baud_Rates_Computation return Baud_Rate_List_Type;

end Baud_Rate_Definitions;
```

```
package body Baud_Rate_Definitions is
```

```
    Baud_Rate_List_Size : constant := 12;
```

```
    function Baud_Rates_Computation return Baud_Rate_List_Type is
```

```
        Result : Baud_Rate_List_Type (1 .. Baud_Rate_List_Size);
```

```
    begin -- Baud_Rates_Computation
```

```
        Result (1 .. 5) := (45.0, 50.0, 56.9, 74.2, 75.0);
```

```
        for I in 6 .. Baud_Rate_List_Size loop
```

```
            Result (I) := 2 * Result (I - 1);
```

```
        end loop;
```

```
        return Result;
```

```
    end Baud_Rates_Computation;
```

```
end Baud_Rate_Definitions;
```

```
-----
```

```
with Baud_Rate_Definitions;
```

```
package Line_Transmission_Package is
```

```
    subtype Baud_Rate_List_Type is
```

```
        Baud_Rate_Definitions.Baud_Rate_List_Type;
```

```
    Baud_Rate_List : constant Baud_Rate_List_Type :=
```

```
        Baud_Rate_Definitions.Baud_Rates_Computation;
```

```
    ...
```

```
end Line_Transmission_Package;
```

Because Baud_Rate_List is a constant, it can be declared without an index constraint. Thus the only reference to Baud_Rate_List_Size is within the Baud_Rate_Definitions package body itself. This package body is the only unit that has to be recompiled when Baud_Rate_List_Size is modified. In fact, any modification to the contents of Baud_Rate_List, whether or not it follows the current algorithm, can be achieved entirely by changing and recompiling the Baud_Rate_Definitions package body.

3. EPILOGUE

In Example Problem Statement (1), the positions of the characters in each component of Frame_Chars were relevant to the processing of the data. It is an instructive exercise to imagine that this were not so and to rewrite the two components of the initial value using catenations of named aggregates. This allows the Ada description of these strings to correspond precisely to the English description in the Example Problem Statement.

In Example Problem Statement (2), the need for a separate package resulted from the desire to declare an array as a constant yet to initialize it algorithmically. Complications arose because of an Ada technicality known as the access-before-elaboration problem [Bel82]. Access before elaboration is an important concern for Ada implementers because an Ada program must raise an exception when it occurs; but because access before elaboration is widely perceived as an anomaly that does not occur in "real" Ada programs, it has not been regarded as an important issue for Ada programmers. Thus it is interesting to find a programming situation that is both typical and conducive to access-before-elaboration errors.

It may appear that the resulting complications outweigh the benefits of declaring the table as a constant rather than a variable. However, the constant declaration also made it possible to limit recompilation requirements, by keeping references to the size and content of the table out of the Line_Transmission_Package specification. Alternatives are available which avoid the need for a separate package, but complicate the type of Baud_Rate_List. One alternative is to use an access type and allocate Baud_Rate_List dynamically, as follows:

```
package Transmission_Line_Package is

  type Baud_Rate_Array_Type is
    array ( Positive range <> ) of Baud_Rate;

  type Baud_Rate_List_Type is access Baud_Rate_Array_Type;

  Baud_Rate_List : Baud_Rate_List_Type;

  ...

end Transmission_Line_Package;
```

```
package body Transmission_Line_Package is
```

```
    Baud_Rate_List_Size : constant := 12;
```

```
begin -- Transmission_Line_Package
```

```
    Baud_Rate_List :=
```

```
        new Baud_Rate_Array_Type (1 .. Baud_Rate_List_Size);
```

```
    Baud_Rate_List (1 .. 5) := (45.0, 50.0, 56.9, 74.2, 75.0);
```

```
    for I in 6 .. Baud_Rate_List_Size loop
```

```
        Baud_Rate_List (I) := 2 * Baud_Rate_List (I - 1);
```

```
    end loop;
```

```
    ...
```

```
end Transmission_Line_Package;
```

Another alternative is to use a record type with a discriminant to enclose an array whose upper bound is given by the discriminant value. An object in the record type can be declared without a discriminant constraint -- keeping the specification of Transmission_Line_Package devoid of any reference to the size of the table -- as long as the discriminant has some dummy default initial value. The resulting package looks like this:

```
package Transmission_Line_Package is
```

```
    type Baud_Rate_Array_Type is
```

```
        array ( Positive range <> ) of Baud_Rate;
```

```
    type Baud_Rate_List_Type (Size : Natural := 0) is
```

```
        record
```

```
            Rates : Baud_Rate_Array_Type (1 .. Size);
```

```
        end record;
```

```
    Baud_Rate_List : Baud_Rate_List_Type;
```

```
    ...
```

```
end Transmission_Line_Package;
```

```
package body Transmission_Line_Package is
```

```
    Baud_Rate_List_Size : constant := 12;
```

```
begin -- Transmission_Line_Package
```

```
    Baud_Rate_List :=
```

```
        (Size => Baud_Rate_List_Size,
```

```
         Rates => (45.0, 50.0, 56.9, 74.2, 75.0, others => 0.0) );
```

```
    for I in 6 .. Baud_Rate_List_Size loop
```

```
        Baud_Rate_List.Rates(I) := 2 * Baud_Rate_List.Rates(I - 1);
    end loop;
```

```
    ...
```

```
end Transmission_Line_Package;
```

Of course all references to Baud_Rate_List in the rest of the program must then be changed to Baud_Rate_List.Rates. Depending on the implementation of record types with discriminants used as array bounds, this solution may be prohibitively expensive in terms of space. Whether an access type or a record type with a discriminant is used, the list of baud rates can be expanded according to the doubling algorithm simply by changing the declaration of Baud_Rate_List_Size and recompiling the Transmission_Line_Package body. Compared with these alternatives, the creation of another package to house the function Baud_Rates_Computation no longer seems very complicated.

THIS PAGE INTENTIONALLY LEFT BLANK

2.2.4 RECORD TYPES

1. BACKGROUND

Case Study Objective

To provide examples of uses of record types in Ada; to illustrate the use of discriminants, variant records, and default values.

Problem

In designing data structures in Ada which use records, the programmer has numerous options to consider. Should the record be "plain" or should it have discriminants? If he chooses to build a record with discriminants, should he assign default values to the discriminants? If the record type has discriminants, how can one declare "independent" variant parts for two or more discriminants?

Discussion

There are no easy answers to these questions. It ultimately depends on the situation: what are the objectives of the data structures, how restrictive should the implementation of the requirements specification be. In the case of record types with discriminants, there are different programming implications that follow from the use or omission of default values. For example, in the absence of default values for the discriminants of record types, objects declared of that type must be constrained for life and are consequently rather inflexible. This inflexibility, however, may better model the requirements. This case study will address these questions and suggest different solution approaches for a problem in a message switch data structure. Each solution has advantages and shortcomings -- none is perfect.

2. DETAILED EXAMPLE

Example Problem Statement

Consider a message switching system. The basic unit of data in this system is a message which is received, decoded, routed and logged. The logging process involves keeping a history of each message that passes through the switch, such as receipt or transmission. The historical information captured for each message is some combination of the following:

General message data:
 number of blocks in the message
 channel
 number of header segments
 channel mode
 sequence number
 reason, if any, that message was not sent
Routing indicators, if applicable
Message control block segment, if applicable
Header segments, if applicable

There are several ways of encoding this information in type declarations using arrays and records. Three such possibilities are discussed in Solution Outlines (1), (2) and (3). These three approaches illustrate different ways of achieving the same result, and they reflect tradeoffs among complexity, design and readability.

In order to define the types of interest, several supporting types must also be declared. One such type, which is modified in one of the three approaches discussed in this case study, is declared below.

```
type Segment_Type is
  record
    Previous_Segment : Segment_Pointer      := null;
    Classification   : Security_Classification;
    Forward_Segment  : Segment_Pointer      := null;
    Number_Chars     : Character_Count_Type := 0;
    Part_Name        : Part_Name_Type;
    Segment_Number   : Segment_Number_Type;
    Text             : String (1 .. 80);
  end record;
```

Solution Outline (1) -- array solution

In the first approach to be considered, the main data structure is declared as an array with four components, corresponding to general message data, routing indicators, message control block segment and

header segments respectively. Since the four components are heterogeneous, the corresponding data structures are declared as variants of an artificially defined record type.

Detailed Solution (1) — array solution

The additional type declarations needed to define a data structure for logging messages are shown below.

```

type Entry_Type is (Log, RI_Set, Segment);

type History_Entry_Type (Entry_Kind : Entry_Type      := Segment;
                        RI_Count   : Number_RIs_Type := 0;
                        Log_Entry  : Log_Entry_Type  := SOM_In) is
  record
    case Entry_Kind is
      when Log =>
        Block_Count   : Number_Blocks_Type;
        Channel       : Channel_Des;
        Header_Segments : Number_Blocks_Type;
        MCB           : Boolean;
        Channel_Mode   : Channel_Mode_Type;
        RIs_Present    : Boolean;
        Sequence_Number : CSN;
      case Log_Entry is
        when Reject =>
          Reject_Reason : Reject_Reason_Type;
        when Cancel_Out | CANTRAN_Out =>
          Cancel_Reason : Cancel_Reason_Type;
        when Scrub =>
          Scrub_Reason  : Scrub_Reason_Type;
        when others =>
          null;
        end case;
      when RI_Set =>
        RI_Array : RI_Array_Type (1 .. RI_Count);
      when Segment =>
        Segment_Buffer : Segment_Type;
      end case;
    end record;
  end record;

```

Because both the MCB and the header are segments, one variant is sufficient to account for either class of information.

The declaration below achieves the second goal stated in Solution Outline (1), specifically to create an array defining the information that must appear in an entity to be logged:

```

type Entry_Set_Type is
  array (Positive range <>) of History_Entry_Type;
-- An Entry_Set_Type object consists of a History_Entry_Type component
-- of type Log followed by:
--   an optional RI_Set
--   an optional MCB Segment
--   0 or more header Segments
-- This ordering MUST NOT be violated.

```

The problem here is that the type declaration `Entry_Set_Type` does not enforce an ordering on the array elements, a requirement imposed in the comments. Although this is a legal Ada implementation of a data structure to log messages, it does not embody all of the requirements of this data structure. The greater degrees of freedom in the type declarations above affect program reliability and maintainability adversely because all program units that manipulate `Entry_Set_Type` objects assume that the array has been correctly filled. Adherence to the order restriction depends on the programmer's awareness of the restriction rather than on any compile-time check. In the event that, say, the second element is found to be a header segment and the third element to be a `RI_Set`, then program execution is likely to raise an exception unexpectedly.

This solution points out a technicality worth noticing. In declaring an array of records, a discriminant constraint must be provided for the record type in question, unless all the discriminants have default values. If, therefore, an array must have heterogeneous components, the type of the component must be defined with default values for all discriminants. A similar precaution must be taken for objects of a record type which can contain different variants at different times.

Solution Outline (2) -- record type solution

In Ada, the proper mechanism for grouping heterogeneous objects is the record. `Entry_Set_Type` objects consist of four parts: a history entry, a set of routing indicators, an `MCB_Segment`, and a set of header segments. The last three objects are optional; however, if more than one of them is present, they must appear in the order listed. In Ada, the programmer can use discriminants to indicate that these components are optional. One discriminant will indicate the number of routing indicators in the `RI_Set`, a second discriminant will specify the existence of the `MCB_Segment`, and a third discriminant will control the presence of header segments.

From a conceptual point of view, a series of variant parts embedded inside the definition of the record is needed to define the `Entry_Set_Type` record. If there are routing indicators present, then an `RI_Set` component is needed. Similarly, if the `MCB_Segment` exists, then a component should be created for it, and depending on the number of

header segments, a component to hold these header segments may or may not be needed. In pseudocode, the declaration would be as follows (note that what follows is not legal Ada but serves to outline what the solution intends to implement):

```
--
-- Pseudo-Ada declarations
--
-- type Entry_Set_Type (Number_of_RIs          : Number_RIs_Type;
--                      MCB_Segment_Option     : Boolean;
--                      Number_Header_Segments : Natural;
--                      Log_Entry              : Log_Entry_Type) is
--   record
--     Log : [some appropriate type];
--     case Number_of_RIs is
--       when 0 =>
--         null;
--       when others =>
--         RI_Set : [some appropriate type];
--     end case;
--     case MCB_Segment_Option is
--       when False =>
--         null;
--       when True =>
--         MCB_Segment : Segment_Type;
--     end case;
--     case Number_Header_Segments is
--       when 0 =>
--         null;
--       when others =>
--         Header_Segment : Segment_Type;
--     end case;
--   end record;
```

Ada does not allow multiple variant parts as written above in the declaration of a record. Either case statements must be nested (as in the History_Entry_Type declaration in Detailed Solution (1)) or each case must be captured by a type or subtype declaration. In this instance, the second approach reflects the desired structure.

Detailed solution (2) — record type solution

Some of the types from Detailed Solution (1) have been modified, and only these revised declarations appear below.

```

type Segment_Type (Part_Name : Part_Name_Type) is
  record
    Previous_Segment : Segment_Pointer := null;
    Classification    : Security_Classification;
    Forward_Segment   : Segment_Pointer := null;
    Number_Chars       : Character_Count_Type := 0;
    Segment_Number     : Segment_Number_Type;
    Text               : String (1 .. 80);
  end record;

```

```

type History_Entry_Log_Variant (Log_Entry : Log_Entry_Type) is
  record
    Block_Count : Number_Blocks_Type;
    Channel      : Channel_Des;
    Channel_Mode : Channel_Mode_Type;
    Sequence_Number : CSN;
    case Log_Entry is
      when Reject =>
        Reject_Reason : Reject_Reason_Type;
      when Cancel_Out | CANTRAN_Out =>
        Cancel_Reason : Cancel_Reason_Type;
      when Scrub =>
        Scrub_Reason : Scrub_Reason_Type;
      when others =>
        null;
    end case;
  end record;

```

```

type MCB_Segment_Type (MCB_Segment_Option : Boolean) is
  record
    case MCB_Segment_Option is
      when True =>
        History : Segment_Type (Part_Name => MCB);
      when False =>
        null;
    end case;
  end record;

```

```

type Header_Segment_Type (Number_Header_Segments : Natural) is
  record
    case Number_Header_Segments is
      when 0 =>
        null;
      when others =>
        History : Segment_Type (Part_Name => Segment);
    end case;
  end record;

```

```

type RI_Array_Type is array (Number_RIs_Type range <>)
  of RI_String_Type;

```

```

type RI_Set_Type (Number_RIs : Number_RIs_Type) is
  record
    RI_Set : RI_Array_Type (1 .. Number_RIs);
  end record;

type Entry_Set_Type (Number_RIs      : Number_RIs_Type;
                     MCB_Segment_Option : Boolean;
                     Number_Header_Segments : Natural;
                     Log_Entry         : Log_Entry_Type) is
  record
    Log : History_Entry_Log_Variant
          (Entry_Kind => Log_Entry);
    RI_Set : RI_Set_Type (Number_RIs);
    MCB_Segment : MCB_Segment_Type (MCB_Segment_Option);
    Header_Segment : Header_Segment_Type (Number_Header_Segments);
  end record;

```

The definition of Segment_Type above has a discriminant which is not used in the actual record declaration. This construction documents the fact that there are different kinds of segments. Furthermore, it allows the Part Name of the segment to be set as a "parameter" when a subsequent record definition includes a component of type Segment_Type. Thus, if the logged message has an MCB_Segment, then the Part_Name of the segment will always be correct because its value is fixed by the discriminant constraints specified in the component declaration using Segment_Type.

While this approach implements the requirements more restrictively, it has resulted in a proliferation of type definitions, in particular of record types with discriminants. The data structure at the most abstract level, in fact, has grown in complexity from being an one-dimensional array type to a record type with four discriminants, none of which have default values.

Solution Outline (3) -- modified record type solution

Depending on the actual application, the data structures for logging message data could be further tailored according to the way in which they are used. Suppose, for example, that the message switch only used three forms of Entry_Set data:

- log entry information only
- log entry information + RI_Set
- log entry information + RI_Set + set of header segments

A record type using nested variants would be appropriate in this case; only if there is a set of routing indicators could there be a set of header segments. In Ada, these specifications would be expressed as shown in Detailed Solution (3).

Detailed Solution (3) -- modified record type solution

The type definitions assumed for this solution are those from Detailed Solution (1). Imitating one step of the second solution, one of the top-level variants, that corresponding to the discriminant Entry_Kind, is isolated and the associated information is captured in the record type Log_Request_Type. In the History_Entry_Type declaration in Detailed Solution (1), the variant part associated with the Entry discriminant shared no components with the variant part corresponding to the discriminant RI_Count. The variant part associated with the discriminant Log_Entry was used exclusively inside the variant part corresponding to the Entry discriminant, and it is carried through in the Log_Request_Type record type. Thus a level of nesting is eliminated, and a record type definition replaced with a simpler one, as shown below:

```
type Log_Request_Type (Log_Entry : Log_Entry_Type := SOM_In) is
  record
    Blocks           : Number_Blocks_Type;
    Channel           : Channel_Des;
    Channel_Mode      : Channel_Mode_Type;
    Sequence_Number   : CSN;
    case Log_Entry is
      when Reject =>
        Reject_Reason : Reject_Reason_Type;
      when Cancel_Out | CANTRAN_Out =>
        Cancel_Reason : Cancel_Reason_Type;
      when Scrub =>
        Scrub_Reason   : Scrub_Reason_Type;
      when others =>
        null;
    end case;
  end record;
```

```

type Entry_Set_Type (Log_Entry           : Log_Entry_Type := SOM_In;
                     RI_Count             : Number_RIs_Type := 0;
                     Header_Segment_Count : Natural := 0) is
record
  Log_Entry : Log_Request_Type (Log_Entry);
  case RI_Count is
    when 0 =>
      null;
    when others =>
      RI_Set_Part : RI_Array_Type (1 .. RI_Count);
      case Header_Segment_Count is
        when 0 =>
          null;
        when others =>
          Header_Segment_Part : Segment_Type;
      end case;
    end case;
  end case;
end record;

```

The type declaration of History_Entry_Type was simplified so that it only has one discriminant, as reflected in the type Log_Request_Type. The type Entry_Set_Type is a record in order to enforce a certain ordering of components, whose presence or absence is in turn determined by the two discriminants RI_Count and Header_Segment_Count.

3. EPILOGUE

This case study has addressed one aspect of record types, namely the use of discriminants. The issue of whether to assign a default value in a discriminant specification is an important design decision because it affects other type declarations (which use this type as the type of a component) and object declarations.

Furthermore, it illustrates a useful coding paradigm. When a record is to have several independent consecutive variant parts, then appropriate types or subtypes must be declared for each variant part. These types are themselves record types with discriminants and contain the corresponding variant part intended for the original record. If an ordinary component is only relevant for a given value of discriminant A, it is placed in a variant part and only exists in those records for which discriminant A has the given value. If discriminant B is only relevant for a given value of discriminant A, it still exists in all records of the type.

2.2.5 RECURSIVE TYPE DEFINITIONS

1. BACKGROUND

Case Study Objective

To illustrate methods for declaring a type T whose declaration refers, directly or indirectly, to T itself.

Problem

Situations often arise in which there is a "circle" of types, say T1, T2, ..., Tn, such that T2 is defined in terms of T1, T3 is defined in terms of T2, and so forth, and T1 is defined in terms of Tn. Then the types T1, T2, ..., Tn are called recursive. Normally, the rules of Ada require that a type be declared before it is referred to in the declaration of another type. In the case of recursive types defined in terms of each other, the circularity makes it impossible to declare each type before it is referred to in the declaration of another type. Thus special measures must be taken when declaring recursive types.

In Ada, an object in a given type cannot contain a subcomponent of the same type (or of a type derived from that type). It follows that circular definitions only make sense when at least one link in the circle involves a type whose values point to values in the next type in the chain, rather than containing them.

Usually, when we speak of one Ada value "pointing to" another, we have in mind an access value designating some allocated object. However, an element in a direct access file can "point to" another such element by containing the direct access key for that file element. Because the data type of the file element includes a component "pointing" to other values of the same data type, there is a sense in which this type can also be considered recursive.

Discussion

Ada has a mechanism specifically designed for the first kind of recursive type definition, in which the pointers are access values. Example Problem Statement (1) describes such a recursive type, and the straightforward way in which it can be declared using an incomplete type declaration. The second kind of recursion, in which pointers are keys of direct access file elements, is illustrated in Example Problem Statement (2). Since the design of Ada does not anticipate this form of recursion, its implementation is more complicated.

2. DETAILED EXAMPLES

Example Problem Statement (1)

In a message switch, physical transmission lines are specified by values in the type Physical_Line_Type, declared as follows:

```
type Physical_Line_Type is range 0 .. 50;
```

In allocating lines for the transmission of various messages, it is useful to maintain lists of lines that are and are not in use. It must be easy to add Physical_Line_Type values to these lists and to remove them. Thus a linked-list data structure is appropriate.

A linked-list data structure is built out of a sequence of list cells. Each list cell is a dynamically allocated record with two components. One component is a Physical_Line_Type value and the other is an access value designating the next list cell. The list as a whole is represented in terms of an access value designating the first cell in the list.

The data type for representing a linked list is recursive because the access type must be defined in terms of the list cell record type it designates and the record type must be defined in terms of the access type that is used for one of its components. Normally, the name of a type must be declared before it can be used in the declaration of another type. However, this would require the list cell record type and the access type each to be declared before the other.

Solution Outline (1)

Ada provides a special mechanism, the incomplete type declaration, for just this situation. An incomplete type declaration is a forward reference to a type declaration that will occur later in the declarative region. It specifies that a certain name will be used as the name of some type to be described in detail later. A type name declared in this way can then be used in an access type declaration to name the type to be designated by values in the access type. The incomplete type declaration must eventually be followed by a complete type declaration describing the designated type. The complete type declaration may follow the access type declaration and refer to it. The forward reference breaks the circle of dependencies and allows the types making up a recursive definition to be declared in a sensible order.

Detailed Solution (1)

Between an incomplete type declaration and the corresponding complete type declaration, the name declared in the incomplete type declaration may only be used for the purpose described in Solution Outline (1) -- to name the type to be designated by the type being declared in an access type declaration. Thus the linked list described in Example Problem Statement (1) must be declared in the following steps:

- (1) Name the list cell record type in an incomplete type declaration.
- (2) Declare the access type designating the list cell record type.
- (3) Declare the list cell record type with a complete type declaration, referring to the access type declared in step (2).

In Ada, the declarations are as follows:

```
type Physical_Line_List_Cell_Type;    -- incomplete declaration

type Physical_Line_List_Type is access Physical_Line_List_Cell_Type;

type Physical_Line_List_Cell_Type is -- complete declaration
  record
    Physical_Line_Part : Physical_Line_Type;
    Link_Part          : Physical_Line_List_Type;
  end record;
```

The steps required may be difficult for Pascal programmers to learn, because they are not carried out in the same order as the corresponding steps in a recursive Pascal type definition. (In Pascal, one could write

```
PhysicalLineListType = ^PhysicalLineListCellType;
PhysicalLineListCellType =
  record
    PhysicalLinePart : PhysicalLineType;
    LinkPart         : PhysicalLineListType
  end;
```

even though PhysicalLineListCellType is used in a pointer type definition before it is itself defined. The Pascal pointer type is defined first, then the record type it points to.)

Example Problem Statement (2)

A data base is to be implemented using Ada. Data is stored in direct access files. Besides data, each element of a direct access file contains pointers to other elements. These pointers are not access values, as in Example Problem Statement (1), but keys identifying file elements. For simplicity, this case study assumes that the only external data in the data base belongs to some type `Data_Type`, and that the only links between file elements are those required to form a doubly-linked list of file elements containing `Data_Type` values.

The data type of the file elements is recursive in the following sense: Each element of the direct access file includes components identifying other elements of the file. Elements of direct access files are identified by values in the integer type `Count` that is provided by an instantiation of the package `Direct_IO`. But this instantiation requires the data type of the file elements to be provided as a generic actual parameter.

If the rules of Ada allowed it, the definition of the file element data type could look something like this:

```
type File_Element_Type;

package Data_Base_IO_Package is
  new Direct_IO ( Element_Type => File_Element_Type ); -- ILLEGAL!

type File_Element_Type is
  record
    Data_Part           : Data_Type;
    Forward_Link_Part,
    Backward_Link_Part : Data_Base_IO_Package.Count;
  end record;
```

This recursive definition is written in the way we would write the definition of a recursive data type involving access values. It starts with an incomplete type declaration for the record to be pointed to. There is then a declaration defining the pointer type in terms of the incompletely declared type -- in this case a generic instantiation rather than an access type declaration -- followed by a complete declaration in which the pointer type is used as a component. This is not legal Ada because an incompletely declared type can only be used in an access type declaration. Thus the use of `File_Element_Type` as a generic actual parameter is forbidden.

Solution Outline (2)

When the recursive type declarations involve access types, an incomplete type declaration is used to provide a forward reference. When the recursive type declarations do not involve access types,

incomplete type declarations are useless. Forward references can be provided by private type declarations instead. If this is done in an appropriate way, the private type declaration will also serve the cause of data abstraction, which is its primary purpose. A difficulty arises when the forward reference is not to a type declaration in the same declarative region, but to a type declared in the specification of a generic package instantiated in that declarative region.

Detailed Solution (2)

As noted in Solution Outline (1), an incomplete type declaration makes recursive definitions possible by providing a forward reference to a yet-to-be declared type. Private type declarations, though devised for a different purpose, also create a forward reference to a yet-to-be-declared type. There are restrictions on the use of a private type between the point of the private type declaration and the corresponding full declaration in a package specification. However, these restrictions are milder than those on incompletely declared types. As with incompletely declared types, it is illegal to use the private type as a generic actual parameter in this part of the package specification. However, the private type can be used in any other type declaration, not just an access type declaration.

Besides containing an illegal use of a not-yet-fully-declared private type as a generic actual parameter, the package specification below hides the type `Data_Base_IO_Package.Count`. This makes it impossible to manipulate pointers to file elements outside of `Data_Base_Definition_Package`. This is not just a matter of failing to provide the necessary subprograms. There is no way to name the data type these subprograms should manipulate, except in the private part and body of `Data_Base_Definition_Package`.

```
package Data_Base_Definition_Package is

  type File_Element_Type is private;

private -- Data_Base_Definition_Package

  package Data_Base_IO_Package is
    new Direct_IO (Element_Type => File_Element_Type); -- ILLEGAL!

  type File_Element_Type is
    record
      Data_Part           : Data_Type;
      Forward_Link_Part,
      Backward_Link_Part : Data_Base_IO_Package.Count;
    end record;

end Data_Base_Definition_Package;
```

Because private types can be used in more ways than incompletely declared types, another approach is available. We need not break the circle of recursive definitions at the point at which we are required to do so when dealing with access types. In particular, we can start with a forward reference to the pointer type, followed by a declaration of the file element type in terms of the pointer type, followed by a full declaration for the pointer type in terms of the file element type:

```
package Data_Base_Definition_Package is

    type Position_Type is private;

    ... -- declarations of subprograms to manipulate Position_Type
        -- values, and of exceptions raised by those subprograms

private -- Data_Base_Definition_Package

    type File_Element_Type is
        record
            Data_Part           : Data_Type;
            Forward_Link_Part,
            Backward_Link_Part : Position_Type;
        end record;

    package Data_Base_IO_Package is
        new Direct_IO ( Element_Type => File_Element_Type );

    subtype Position_Type is Data_Base_IO_Package.Count; -- ILLEGAL !

end Data_Base_Definition_Package;
```

As noted in the program comment above, the use of a subtype declaration as the full declaration of a private type is illegal. Thus the definition of the pointer type in terms of the file element type is not as simple as we might hope. Before we address solutions to this problem, however, it would be appropriate to examine why the general approach sketched above is desirable.

As before, the private part of the package hides certain information from the users of the package. This time, however, the information hidden is precisely that information that should be hidden. The data abstraction implemented by the package is that of a list of Data_Type values. Values of type Position_Type represent positions within the list. The operations of this data abstraction may include inserting a Data_Type value at a given position, deleting the Data_Type value at a given position, retrieving the Data_Type value at a given position, or advancing to the next position or previous position. From this point of view, it is irrelevant that the Data_Type values reside on secondary storage, that the file elements stored there include forward and backward links as well as Data_Type values, or that the Position_Type values are really direct access file keys. Thus

AD-A140 818

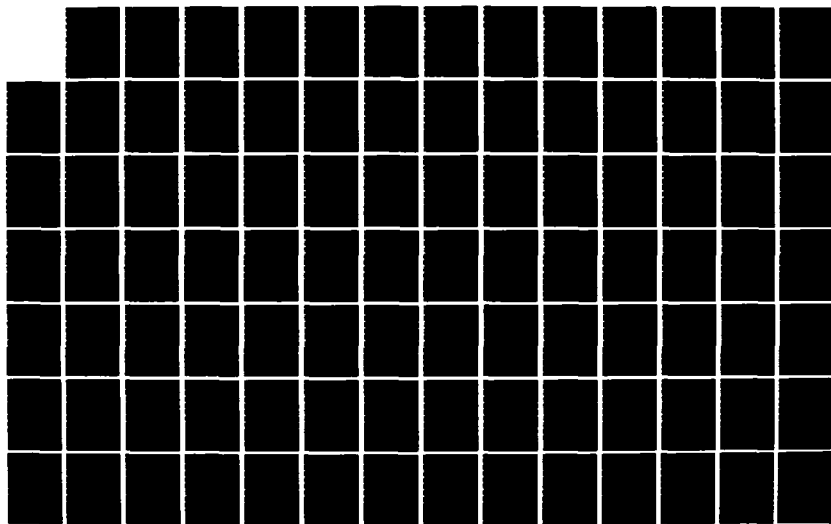
ADA (TRADEMARK) CASE STUDIES II(U) SOFTECH INC WALTHAM
MA JAN 84 DAAB07-83-C-K514

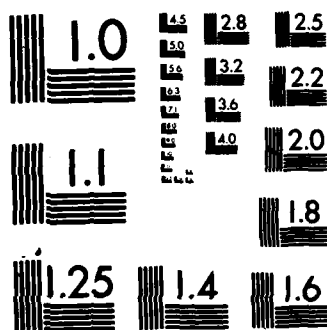
2/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Position_Type ought to be a private type; the type File_Element_Type and the input/output operations provided by Data_Base_IO_Package ought to be inaccessible from outside Data_Base_Definitions_Package.

Now let us return to the problem of declaring Position_Type. It is common for a private part to contain a sequence of declarations necessary for the implementation of a private type, followed by a declaration describing the implementation itself. The intent in Data_Base_Definitions_Package was to implement Position_Type by instantiating the generic package Direct_IO, obtaining an instance of the type Count, and then designating that instance as Position_Type. The subtype declaration was meant to act as a "type renaming declaration." However, the rules of Ada require that a private type declaration be accompanied by a full type declaration introducing a new type.

Data_Base_Definitions_Package could be made legal by replacing the subtype declaration for Position_Type with the following type declaration:

```
type Position_Type is
  record
    Only_Component : Data_Base_IO_Package.Count;
  end record;
```

A record with one component allows the definition of an ostensibly new type, but one whose value essentially consists of a value in an already existing type. Within the body of Data_Base_Definitions_Package, all uses of a Position_Type value X for input or output operations would have to be written as X.Only_Component .

A similar approach is to define Position_Type as a new type derived from the type Data_Base_IO_Package.Count:

```
type Position_Type is new Data_Base_IO_Package.Count;
```

In this case, all uses of a Position_Type value for input or output operations within the Data_Base_Definitions_Package body would have to be written as conversions to type Data_Base_IO_Package.Count. Since the syntax of a type conversion requires a type mark rather than an expanded name, the body of Data_Base_Definitions_Package would have to contain a declaration like

```
subtype Data_Base_IO_Count_Type is Data_Base_IO_Package.Count;
```

so that the use of Position_Type value X for input or output could be written as Data_Base_IO_Count_Type(X).

3. EPILOGUE

Example Problem Statement (1) involved recursion between one access type and one record type. In many applications, more complicated patterns of mutual recursion arise. For instance, there may be several different record types, each containing components designating values in the other record types. There may be more than one access type by which a record may be able to designate other records in its own type. Such applications would be good exercises for testing one's understanding of the rules governing incomplete type declarations.

The second problem could have been made non-recursive by declaring the Forward_Link_Part and Backward_Link_Part components of File_Element_Type to belong to some previously defined integer type, such as the predefined type Integer. Then type conversions like those proposed for the derived type solution above would be used inside the Data_Base_Definitions_Package body to convert values in type Integer to values in type Data_Base_IO_Package.Count when performing input or output. However, this approach has serious drawbacks. First, data declarations would no longer reflect the fact that internally, positions in the list were represented by the keys of direct access file elements. It would be necessary to examine procedural text to determine the significance of the numeric values used as links. Second, the range of the type Count (like that of the type Integer) is implementation-defined. There is no guarantee that an implementation will provide an Integer value for every possible Count value.

In any event, the approach chosen to facilitate recursive definition — the declaration of Position_Type as a private type — is desirable simply for purposes of data abstraction, so there is nothing to be gained in avoiding the recursive approach. Indeed, a good exercise in data abstraction is to supply the declarations missing from the visible part of the Data_Base_Definitions_Package as given in Detailed Solution (2) -- the declarations of subprograms and exceptions to implement the data abstraction described in the text.

In practice, the interconnections between file elements in a data base are considerably more complicated than the doubly-linked list described here. A file element may reside on several different linked lists at once. A file containing one type of element may contain pointers to lists of elements in another file, containing different types of data; each element X of the second file may contain a back-pointer to that element of the first file pointing to the list containing X. An exercise appropriate for those familiar with network data base design and implementation is to apply the techniques of Detailed Solution (2) to a more complicated example.

The designers of Ada seem to have considered only access values in providing for declarations of data structures in which one value "points" to another. In this case study, the observation that values can also point to each other through direct access file element keys led to the discovery of another variety of recursive type definition. An interesting research topic is a search for yet other ways in which one Ada value can point to another.

THIS PAGE INTENTIONALLY LEFT BLANK

2.3 Coding Paradigms

This section contains the following case studies:

Use of Slices

Short Circuit Control Forms

Loop Statements

Use of Block Statements for Local Renaming

THIS PAGE INTENTIONALLY LEFT BLANK

2.3.1 USE OF SLICES

1. BACKGROUND

Case Study Objective

To illustrate the use of slices in Ada.

Problem

To many programmers, slices will appear as a novel feature of Ada, a feature which they are not sure how to use. In what programming situations do the use of slices promote readability and ease of maintenance?

Discussion

Slices are a useful way of identifying a part of a one-dimensional array. They facilitate the assignment and comparison of parts of arrays because they obviate the need to do parts of these assignments componentwise, through for loops. This case study will present two different applications involving slices, one an alternative to a for loop, and the second an instance where a subtype identifies the slice.

2. DETAILED EXAMPLE

Example Problem Statement (1)

The example discussed here is abstracted from a message switching system. The basic unit of data being processed is a message, which ultimately consists of a sequence of characters, in other words an entity which can be represented as a text string. Various operations can be done on this string or a substring of it, including writing to the string, reading from the string, and comparing it to another string. In many languages, these operations are implemented using a for loop, as illustrated below. Two of these examples show operations on a substring of a string because Ada allows the assignment and comparison of entire arrays.

```
for I in Lower_Bound .. Upper_Bound loop
  Target_String (I) := Source_String(I); -- write to Target_String
end loop;
```

```
for I in Source_String'Range loop
  Character_Read := Source_String (I); -- read from Source_String
  .
  .
end loop;
```

```
Same_String := True;
for I in Lower_Bound .. Upper_Bound loop
  if Source_String (I) /= Target_String (I) then
    Same_String := False;
    exit;
  end if;
end loop;
```

(The above code excerpts make certain assumptions which might not necessarily apply in another situation. For example, they assume that the source and target strings have the same absolute indices. In the event that they only had the same relative indices, then a new index, say J, would have to be introduced and specifically incremented inside the loop, or an offset would have to be added to one of the subscripts.)

Solution Outline (1)

In Ada, the above operations can be written much more readably by using array operations. instead of for loops. The use of slices makes it possible to apply array operations to portions of strings. The Detailed Solution (1) contains the code.

Detailed Solution (1)

The write operation would look like:

```
Target_String (T_Lower_Bound .. T_Lower_Bound + L) :=  
Source_String (S_Lower_Bound .. S_Lower_Bound + L);
```

The lower bounds of the source and target strings are differentiated through the prefixes S and T respectively in order to emphasize that it is the relative bounds that are important. The upper bounds are specified in terms of the lower bounds to emphasize that these two intervals must be of the same length.

A practical application of the write operation occurs in operations on the array representation of variable-length sequences. Consider a list which is implemented as the array List, where an element List_Element can be added at or deleted from any location Insertion_Point or Deletion_Point in List. When an element is added or deleted, then the other array elements are shifted appropriately. Assume that the variable List_End marks the last occupied element of the array and that the list full or empty conditions have already been checked. For contrast, the "old-fashioned" for loop implementations of the add and delete operations are shown first:

```
-- add List_Element to List at position Insertion_Point  
for I in reverse Insertion_Point .. List_End loop  
  List (I + 1) := List (I);  
end loop;  
List (Insertion_Point) := List_Element;  
List_End := List_End + 1;  
  
-- delete List_Element from List at position Deletion_Point  
for I in Deletion_Point .. List_End loop  
  List (I) := List (I + 1);  
end loop;  
List_End := List_End - 1;
```

Notice how these same operations are much more elegant when implemented using Ada slices:

```
-- add List_Element to List at position Insertion_Point  
List (Insertion_Point + 1 .. List_End + 1) :=  
  List (Insertion_Point .. List_End);  
List (Insertion_Point) := List_Element;  
List_End := List_End + 1;  
  
-- delete element from List at position Deletion_Point  
List (Deletion_Point .. List_End - 1) :=  
  List (Deletion_Point + 1 .. List_End);  
List_End := List_End - 1;
```

The read operation looks identical to the write operation shown above. If the length of the target array is the same as the length of the slice of the source array, then the range specification can be omitted from the target:

```
Target_String := Source_String (Lower_Bound .. Lower_Bound + L);
```

The comparison operation is very similar:

```
Same_String :=  
  String_1 (Lower_Bound_1 .. Lower_Bound_1 + L) =  
  String_2 (Lower_Bound_2 .. Lower_Bound_2 + L);
```

Notice how the Boolean variable Same_String is assigned the result of a Boolean expression, in contrast to the if - then - else statement inside a loop used earlier.

Example Problem Statement (2)

Text, read in as a single string, must be broken up into its constituent fields. Suppose that the string is 80 characters long and contains a 20 character name, followed by a work and a home phone number, each 10 characters long, followed by a 37 character job title. Each field is separated by a space. Although the range specification of each field is known, the question arises as to whether there is a more readable way of addressing these fields than:

```
Input_String ( 1 .. 20)    -- name  
Input_String (22 .. 31)    -- work phone number  
Input_String (33 .. 42)    -- home phone number  
Input_String (44 .. 80)    -- job description
```

The problem statement above uses positive integers to specify the bounds of the slice for each field. While this is perfectly correct, it is not perfectly readable to the maintenance programmer, who (1) may not understand the meaning of these particular numbers, (2) must find each and every applicable occurrence of them if a modification changes, say, the length of a field, and (3) distinguish the applicable occurrences from all coincidental uses of those numbers for other purposes.

Solution Outline (2)

Instead of specifying the ranges with integer literals, the programmer can define integer subtypes for each of these ranges. The names of these subtypes should correspond to the names of the fields.

Detailed Solution (2)

Below are four subtype declarations, corresponding to the fields of the input string.

```
subtype Name_Field is Positive range 1 .. 20;  
subtype Work_Number_Field is Positive range 22 .. 31;  
subtype Home_Number_Field is Positive range 33 .. 42;  
subtype Job_Description_Field is Positive range 44 .. 80;
```

The slices shown in the Example Problem Statement (2) can now be rewritten:

```
Input_String (Name_Field)  
Input_String (Work_Number_Field)  
Input_String (Home_Number_Field)  
Input_String (Job_Description_Field)
```

Alternatively, renaming declarations can be used to represent the substrings identified by the slices, as shown below:

```
Name_Field      : String renames Input_String (1 .. 20);  
Work_Number_Field : String renames Input_String (22 .. 31);  
Home_Number_Field : String renames Input_String (33 .. 42);  
Job_Description_Field : String renames Input_String (44 .. 80);
```

3. EPILOGUE

Slices can be applied to any one-dimensional array, not just to strings. For instance if a transmitted byte consists of 8 bits, 7 bits for the character and the 8th character for the parity bit, then this byte can be represented as an array of Boolean values, True representing the bit value 1 and False representing the bit value 0. A slice can be used to represent the 7 bits constituting the character, as in

```
Byte (0 .. 6)
Byte (Character_Bits) -- where Character_Bits is an
                      -- integer subtype with range
                      -- 0 .. 6
```

As another example, consider an array of records. The periodic table of chemical elements could be implemented as an array indexed by either chemical symbol or atomic number, where each component of the array contains the English name of the element, its symbolic name (unless that is the index used), its atomic weight, its atomic number (unless that is the index used), and the family to which the element belongs. A row of the periodic table can be visualized as a slice of this representation of the table.

Slices are a powerful Ada construct. They can be used in many expressions besides the ones illustrated in this case study. Slices are extremely useful in conjunction with the catenation operation. For example, the list insertion shown above could be rewritten as:

```
-- insert List_Element in List at Insertion_Point
List := List (List'First .. Insertion_Point - 1) &
        List_Element &
        List (Insertion_Point + 1 .. List_End);
List_End := List_End + 1;
```

As another example, slices can be used as actual parameters. To output the name field of the string discussed in Example Problem Statement (2), this substring can be specified directly in the call on Text_IO.Put:

```
Text_IO.Put (Input_String (Name_Field));
```

If there is a procedure Sort which takes as its parameter the list of elements described in Detailed Solution (1), then to sort the current elements in the list one would write:

```
Sort (List (List'First .. List_End));
```

2.3.2 SHORT CIRCUIT CONTROL FORMS

1. BACKGROUND

Case Study Objective

To explain when it is appropriate to use the short-circuit control forms and then and or else rather than the operators and and or.

Problem

In coding a Boolean expression that expresses a logical and or or relationship, the Ada programmer may use the logical operators and or or or the short circuit control forms and then or or else. A common misconception is that the short circuit control forms are used to optimize the evaluation of a conditional expression. Examples are needed to illustrate the correct application of these Ada constructs.

Discussion

The purpose of the short circuit control forms is not to generate more efficient object code but to ensure that a conditional expression always has a well-defined result, even when the second operand of this expression does not. In those cases in which the value of the second operand cannot be computed, the value of the expression is completely determined by the value of the first operand. For example if the second expression could raise an exception then the first expression would check for the exception raising conditions. The value of this first expression would then prevent evaluation of the second expression should its evaluation result in an exception. This case study will discuss the differences between the logical operators and the corresponding short circuit control forms, showing examples in which the use of the short circuit control forms is justified.

2. DETAILED EXAMPLES

Example Problem Statement (1)

The code for a date package contains numerous compound conditional expressions built using the short circuit control forms and then and or else. In some instances these expressions serve to request a compiler optimization which would likely be performed anyway, even by a naive compiler. Consider the following, where Current_Year and Current_Month are integer valued variables and Julian_Date stores the nth day of the year.

```
if Current_Year mod 4 = 0 and then Current_Month >= 3 then
    Julian_Date := Julian_Date + 1;
end if;
```

This use of the and then construct essentially instructs the compiler to optimize the if expression: if it is not a leap year, then don't bother checking which month it is.

Solution Outline (1)

The purpose of the and then construct is not to optimize code but to allow the value of the Boolean expression to be determined from the value of the first conjunct, if possible. In cases where evaluation of the second conjunct would raise an exception, this feature is particularly useful. In the example above, however, the second expression (Current_Month >= 3) cannot raise an exception or have other side effects, so the short circuit control form is superfluous and a logical and would be in order.

In general it is inappropriate to use the short circuit control forms purely for the purpose of optimization. It could be argued that the above if statement occurs in a time-critical loop, and that it is of paramount importance that the object code be efficient. Before using the and then construct, however, the programmer should first compile this code written with the and construct and run the entire program to determine where the real inefficiencies are. Furthermore, while the programmer cannot assume an optimizing compiler, he should be aware that most compilers perform some optimization and essentially, he should give the compiler its proverbial chance before "hand"-optimizing his code. Specifically in this case, a compiler would probably recognize that the evaluation of the second expression shown in the Example Problem Statement (1) above (Current_Month >= 3) cannot raise an exception or have any side effects, and it might well optimize the object code so that the second conditional is not evaluated unnecessarily.

Detailed Solution (1)

The following simpler statement is sufficient for this particular if statement:

```
if Current_Year mod 4 = 0 and Current_Month >= 3 then
  Julian_Date := Julian_Date + 1;
end if;
```

Note that now the program maintainer cannot wonder how the expression (Current_Month >= 3) would have an undefined value.

Example Problem Statement (2)

The message data structure provides that a message consists of distinct parts, which in turn each consist of a set of segments. In addition to character text, these segments store control information comprised of the part name (to which they belong), the classification of the message, and the serial number of the segment. Reading from a message involves reading the text stored in the segments for a specified part of the message. In the event that the text is stored on more than one segment, then the reading operation transits from the current segment to the next one. For every such transition, this function must verify that the linkage information is correct. The correctness criterion is that the control information is invariant between segments of the same part, and the serial numbers of two adjoining segments always differ by exactly one. The following code was written to implement this check:

```
if Read_Current_Part (Segment => Next) /=
  Read_Current_Part (Segment => Where_From.Segment)
or else
  Read_Classification (Segment => Next) /=
  Read_Classification (Segment => Where_From.Segment)
or else
  Read_Segment_Serial_Number (Segment => Next) /=
  Read_Segment_Serial_Number
    (Segment => Where_From.Segment) + 1 then
  Status := Linkage_Error;
end if;
```

Examination of this code segment shows the use of the short circuit control form or else. The question arises as to whether this usage is appropriate. Once again, it appears that the short circuit control form is used to force a compiler optimization. Unlike the case presented in Example Problem Statement (1) above, it is not immediately obvious whether a compiler would on its own accord perform this optimization. Given that the Boolean expressions include function calls, it is possible that the function calls produce side effects, a nontrivial determination for the compiler to make.

Solution Outline (2)

The conditional expression in the above if statement includes six function calls and, as such, their parameters are all of mode in. Further analysis of these functions reveals that their implementation does not produce side effects. In order to eliminate the overhead of the function call and execution when the linkage error was detected early in the evaluation of this compound Boolean expression, the programmer decided to optimize the code.

The purpose of the or else construct, however, is not to allow the programmer to optimize code but to compute the value of the entire Boolean expression from the value of the first logical operand, if possible. The true intent of the programmer would be better realized by the code shown in Detailed Solution (2) below.

Detailed solution (2)

The code below shows how the if statement presented in Example Problem Statement (2) can be rewritten without using the or else construct.

```
if not Correctly_Linked (Old_Segment => Where_From.Segment,  
                        Next_Segment => Next) then  
    Status := Linkage_Error;  
end if;
```

where the newly introduced function Correctly_Linked is defined below:

```
function Correctly_Linked (Old_Segment, Next_Segment :  
    Segment_Type)  
    return Boolean is  
begin  
    if Read_Current_Part (Segment => Next) /=  
        Read_Current_Part (Segment => Where_From.Segment) then  
        return False;  
    elsif Read_Classification (Segment => Next) /=  
        Read_Classification (Segment => Where_From.Segment) then  
        return False;  
    elsif Read_Segment_Serial_Number (Segment => Next) /=  
        Read_Segment_Serial_Number  
            (Segment => Where_From.Segment) + 1 then  
        return False;  
    else  
        return True;  
    end if;  
end Correctly_Linked;
```


The solution presented above shows in an explicit manner the optimization to the code. While the or else construct accomplishes the same purpose, it also suggests that the first conjunct always has a well-defined value, even when subsequent conjuncts do not. This implication affects program maintenance adversely; therefore such programming practices are not recommended.

This original example statement illustrates a case where an optimizing compiler might not necessarily be able to optimize out the function calls, precisely because these are function calls which might have side effects. Consequently, it could be argued that the programmer using the or else clause is justified in writing this optimization. On the other hand, the code is more readable when the compound logical expression that checks for linkage error is replaced by a function call. There is a subjective style element involved here; given that the or else construct effectively raises a flag in the reader's mind, at the very least there should be a comment that this is not that kind of usage.

Example Problem Statement (3)

The previous two examples have illustrated how the programmer is not meant to use the short circuit control forms. This example will present the appropriate way to apply this Ada construct.

In a conventional logical expression, the Ada language does not impose an order on the evaluation of the operands. Therefore the programmer who relies on a left to right evaluation in an expression in which the order of evaluation matters writes an erroneous program. To illustrate this point in practice, consider the following loop which searches for an element Key in the array A (the construction of loops to search for an element in an array is discussed in detail in another case study):

```
I := A'First;  
while I <= A'Last and A(I) /= Key loop  
    I := I + 1;  
end loop;
```

A problem occurs when none of the elements of the array A has the value Key. The subscript I has assumed the value A'Last; the corresponding element of the array does not equal Key. Thus the conditional expression of the while loop evaluates to True and the body of the loop is executed, in which the subscript I is incremented. The while loop expression is reevaluated. The first operand evaluates to False; the value of the second operand is undefined and in fact, the attempt to evaluate it raises the exception Constraint_Error. The operands of a logical expression are evaluated in some implementation-defined order prior to the application of the corresponding operator. Although the first operand is sufficient to determine the value of the while

conditional expression, the programmer cannot assume that the compiler will not first evaluate the second operand or that the compiler will optimize the code so that the second operand is not evaluated if the first one is False. (It is a reasonable expectation, as suggested in the earlier discussion; however, it is unreliable. Furthermore, the reference manual implies that both operands will be evaluated regardless.) The programmer must rewrite this condition so that his program will execute smoothly.

Solution Outline (3)

The short circuit control forms were designed for logical expressions in which the second operand may not be computable. The right operand is evaluated only when the entire condition depends on it. In the case of the and then form, the second conjunct is evaluated only when the first conjunct is True. For the or else form, the second operand is evaluated only if the first operand is False. By replacing the and with and then in the while conditional expression, the programmer solves the problem.

Detailed Solution (3)

Below is the same loop rewritten with the and then construct:

```
I := A'First;  
while I <= A'Last and then A (I) /= Key loop  
  I := I + 1;  
end loop;
```

Assuming that Key does not match the values of the first (A'Range - 1) elements of the array A, the subscript I is incremented to the value A'Last. The first operand of the while condition evaluates to True. Therefore the second conjunct is also evaluated. Assuming that the last element of A also does not equal Key the entire logical expression evaluates to True, and the body of the loop is executed. The subscript I now has the value A'Last + 1. The first conjunct of the while condition evaluates to False, thus determining the value of the entire expression, obviating the need to evaluate the second operand, whose evaluation would have raised an exception.

3. EPILOGUE

Example Problem Statement (3) illustrates one case in which short circuit control forms are indeed useful. Initially it may seem easier to visualize using the and then rather than the or else construct. This limitation is overcome if the programmer considers the equivalence of the following logical expressions:

$P \ \& \ Q$ is the same as $\neg(\neg P \mid \neg Q)$

In the context of an if - then - else statement, the following forms are equivalent:

if Condition then	if not Condition then
S1;	S2;
else	else
S2;	S1;
end if;	end if;

An application that illustrates both alternatives of the short circuit control forms involves access types. In the simplest case, imagine that some algorithm may be executed, depending on the value of an object being accessed. Let Pointer_To be the access variable to a simple object (i.e. whose value is scalar). Then, merely writing

```
if Pointer_To.all /= Key then
  S1;
else
  S2;
end if;
```

does not account for the instance where Pointer_To accesses the value null. Rather than writing a nested if statement to first check for a null pointer, then for the value of the accessed object, the code may be written straightforwardly either with and then or or else.

```
if Pointer_To /= null and then Pointer_To.all /= Key then
  S1;
else
  S2;
end if;
```

An alternate way of writing this if statement follows:

```
if Pointer_To = null or else Pointer_To.all = Key then
  S2;
else
  S1;
end if;
```

The three examples discussed in this case study demonstrated proper and inappropriate uses of the short circuit control forms. The intent of these constructs is not to generate more efficient object code but to ensure that a conditional expression can have a well-defined value even when one of its operands do not. It is important that the programmer understand this motivation in deciding whether or not the logical operators and and or are sufficient in coding a compound Boolean expression.

2.3.3 LOOP STATEMENTS

1. BACKGROUND

Case Study Objective

To illustrate coding style issues in the use of Ada loop statements.

Problem

There are several variations of looping statements available to the Ada programmer -- loops with and without for or while clauses, loops with and without exit statements, and loops built from goto statements. The Ada programmer must be able to select the form of loop statement most suited to a given problem.

Discussion

The specific problem considered in this case study is abstracted from a common situation found in virtually all embedded system programs -- searching an array for the occurrence of a given value. In an automatic message switch, for example, there is a requirement to locate a physical output line going to a particular destination. One way of implementing this is to use an array containing an entry for each physical output line. The entry holds the destination associated with that line. Then when a message is bound for a given destination, the array may be searched to find a line going to the given destination. If the destination is found in the output line array, the message may be routed over the corresponding line. If the destination is not found, then some other action must be taken, such as routing the message via a designated exit switch.

To make the looping issues show up clearly, the actual problem considered here is reduced to the essential elements of searching an array for a given value. If the value is found, a Found routine is called and passed the location in the array containing the given value. If the value is not found, a Not_Found routine is called.

Since the goal in this case study is to examine loop statements rather than to look at array searching algorithms, only a simple linear search algorithm is considered. In many practical cases, more sophisticated and efficient algorithms, such as a binary search or hash coding would be more appropriate.

2. DETAILED EXAMPLES

Example Problem Statement

The specific problem is to write a procedure with the following specification:

```
procedure Search (A   : Array_Type;  
                 Val : Value_Type);
```

where Array_Type is defined as

```
type Array_Type is array (Index_Type) of Value_Type;
```

with Index_Type and Value_Type defined elsewhere. The procedure should search the array A for the occurrence of the value Val. If it is found, a procedure Found should be called:

```
procedure Found (Index : Index_Type);
```

where index is the first position in the array containing the value Val. If the value is not found, then the procedure Not_Found should be called:

```
procedure Not_Found;
```

[To make the problem completely self-contained, we could imagine that we are writing a generic procedure for Search that is parameterized by the types Index_Type and Value_Type, and the procedures Found and Not_Found.]

Solution Outline

Three approaches to solving this problem are presented, each approach using a different combination of Ada features:

1. while loop without exit
2. while loop with exit
3. for loop with exit

These approaches were chosen to illustrate the range of Ada looping constructs and to cover a variety of Ada programming styles.

For each approach, several potential solutions are presented and analyzed. The problems found in one potential solution motivate the following solutions.

Following the presentation of the approaches, the concluding section compares the advantages and disadvantages of each approach.

Detailed solution (1) -- while loop without exit

Program 1.1 below shows what might be a first (and, as it turns out, incorrect) cut at a search procedure.

```
-- Program 1.1  ** incorrect **
procedure Search (A   : Array_Type;
                  Val : Value_Type) is

    Index : Index_Type;

begin -- Search

    Index := A'First;

    while Index <= A'Last and then A(Index) /= Val loop
        Index := Index_Type'Succ(Index);
    end loop;

    if Index <= Index_Type'Last then
        Found (Index);
    else
        Not_Found;
    end if;

end Search; -- Program 1.1
```

Unfortunately, this solution is incorrect. In the case that the value is not present in the array, the last pass through the loop will attempt to take the successor of the last value in the type `Index_Type`, resulting in an out-of-range value. This can be fixed in a number of ways -- increasing the range of the index, adding logic to avoid the increment after the last value of the range has been tried, or arranging the loop so that it stops before the last value in `Index_Type`.

In Program 1.2 below, the range on the variable index is increased so that the last pass through the loop does not result in an illegal value:

```

-- Program 1.2
procedure Search (A   : Array_Type;
                  Val : Value_Type) is

    Index : Integer
            range Index_Type'First .. Index_Type'Last + 1;

begin -- Search

    Index := A'First;

    while Index <= A'Last and then A(Index) /= Val loop
        Index := Index_Type'Succ(Index);
    end loop;

    if Index <= A'Last then
        Found (Index);
    else
        Not_Found;
    end if;

end Search; -- Program 1.2

```

Program 1.2 is correct and straightforward. It does, however, assume that `Index_Type` is itself a subtype of `Integer`, otherwise, the concept of

`Index_Type'Last + 1`

cannot be expressed. If `Index_Type` is an enumeration type then there is no index value outside of the range of valid subscripts that can be used to carry the information that the value is not in the array. [Actually, if `Index_Type` is a subtype of a larger enumeration type, the solution will work if the addition is replaced by

`Index_Type'Base'Succ(Index_Type'Last)`

This, however, seems too rare and too clumsy to be considered seriously.]

Another stylistic disadvantage of this program is that the variable `Index` has a different subtype than that used to define the array index range.

Adding additional logic results in a program like Program 1.3 below. A test in the loop avoids the increment of index during the last pass through the loop. In this solution a Boolean variable is used to

carry the information about whether or not the search through the array is completed. If the loop goes completely through to the end without exiting, the Boolean will be set to true to indicate that the value was not found.

```
-- Program 1.3
procedure Search (A   : Array_Type;
                  Val : Value_Type) is

    Index           : Index_Type;
    Search_Complete : Boolean := False;

begin -- Search

    Index := A'First;

    while A(Index) /= Val and not Search_Complete loop
        if Index = A'Last then
            Search_Complete := True;
        else
            Index := Index_Type'Succ(Index);
        end if;
    end loop;

    if not Search_Complete then
        Found (Index);
    else
        Not_Found;
    end if;

end Search; -- Program 1.3
```

This program, while not too clear, will handle any type of `Index_Type`, except a null type (a subtype defined with a range whose first value is greater than its last value). Another difficulty is that the test must be made each time through the loop to see if it is the last pass.

In Program 1.4 below, the loop runs up to, but does not include, the last value of `Index_Type`. Effectively, this puts the test to avoid the last increment into the while clause, so it does not need to be shown explicitly by the programmer.

```

-- Program 1.4
procedure Search (A   : Array_Type;
                  Val : Value_Type) is

    Index : Index_Type;

begin -- Search

    Index := A'First;

    while Index < A'Last and A(Index) /= Val loop
        Index := Index_Type'Succ(Index);
    end loop;

    if A(Index) = Val then
        Found (Index);
    else
        Not_Found;
    end if;

end Search; -- Program 1.4

```

The main difficulty with Program 1.4 is that, unless the value is found in the last position in the array, an extra test for

A(Index) = Val

is made. When the test is a simple array element value comparison, this does not present any practical difficulty. However, it is conceivable in other cases, that the test of whether a particular location contains the searched-for value could be expensive. In these cases, it would be better to avoid the redundant test.

If we change the problem slightly, a neater solution is available. Assume that the array A is made an in out parameter to the procedure Search, so that values in the array can be changed. Further assume that instead of being indexed by the desired Index_Type, the array is indexed by an extended Index1_Type whose range is extended by one (as in Program 1.2). The array definition is:

```

subtype Index1_Type is Integer range 1 .. Index_Type'Last + 1;

type Array_Type is array (Index1_Type) of Value_Type;

```

```

-- Program 1.5
procedure Search (A : in out Array_Type;
                  Val : in Value_Type) is

    Index : Index1_Type;

begin -- Search

    A(A'Last) := Val; -- make sure val will be found somewhere

    Index := A'First;

    while A(Index) /= Val loop
        Index := Index + 1;
    end loop;

    if Index < A'Last then
        Found (Index);
    else
        Not_Found;
    end if;

end Search; -- Program 1.5

```

Program 1.5 bypasses the need to check that the search goes beyond the bounds of the array by storing a copy of the value being searched for in the last array entry (the one that was provided for by extending `Index_Type` by one to `Index1_Type`). The loop over the extended array will definitely halt when this last value is found. The not found condition then becomes just that the value of index at the end of the loop is the last entry in the array.

As with Program 1.2, this solution works only for `Index_Type` being an integer subtype. It also depends on the assumption that the extended array type can be defined and that the array can be stored into.

Detailed Solution (2) -- while loop with exit

The next solutions use the Ada exit statement to leave a loop when the value being searched for is found.

```

-- Program 2.1
procedure Search (A   : Array_Type;
                  Val : Value_Type) is

    Index : Index_Type;

begin -- Search

    Index := A'First;

    while Index < A'Last loop
        exit when A(Index) = Val;
        Index := Index_Type'Succ(Index);
    end loop;

    if A(Index) = Val then
        Found (Index);
    else
        Not_Found;
    end if;

end Search; -- Program 2.1

```

In Program 2.1 note that the while loop tests all the array elements up to, but not including, the last one, since when

Index = A'Last

the while clause will be false. This is done so that the increment of the index (with the "Succ" call) will not yield a value out of range on the last pass through the loop.

When the value is found in any position but the last, the exit statement will be satisfied and the loop will be terminated with index indicating the array element containing the desired value. If the value is not found in that range, the while clause terminates the loop with index containing A'Last.

If the loop was terminated due to the exit statement, the duplicate test in the if clause will yield true and the Found procedure will be called. If the loop was terminated due to the while clause, the found or not-found condition depends on whether or not the last array element contains the desired value. If it does the if clause will be true and the Found procedure will be called. If it does not then the if clause will be false and the Not_Found procedure will be called.

Program 2.1 works for any index type other than a null type. For a null type the use of the "first" attribute will result in a value out of range of the index type, causing an exception when the store into index is attempted.

Program 2.2 is really an extension of the idea of a while loop with an exit. It does not actually use a while loop. Instead it uses a plain loop with no while clause together with an exit statement to effectively produce a loop with a single exit in the middle of the loop.

```
-- Program 2.2
procedure Search (A   : Array_Type;
                  Val : Value_Type) is

    Index      : Index_Type;
    Val_Found  : Boolean;

begin -- Search

    Index := A'First;

    loop
        Val_Found := A(Index) = Val;
        exit when Val_Found or Index = A'Last;
        Index := Index_Type'Succ(Index);
    end loop;

    if Val_Found then
        Found (Index);
    else
        Not_Found;
    end if;

end Search; -- Program 2.2
```

Detailed Solution (3)

The final solution approach uses a for loop to step through the array coupled with an exit statement to terminate the loop when the value is found. Program 3.1 shows a first, though incorrect, attempt to write the program with a for loop.

```
-- Program 3.1
procedure Search (A   : Array_Type;
                  Val : Value_Type) is

    Index : Index_Type;

begin -- Search

    for Index in Index_Type loop
        exit when A(Index) = Val;
    end loop;

    if A(Index) = Val then
        Found (Index);
    else
        Not_Found;
    end if;

end Search; -- Program 3.1
```

This simple program is incorrect because it ignores the fact that a for loop is a new variable scope region and that the loop index is a new, implicitly declared variable that exists only in that loop. This means that the variable index used within the loop is not the same as the variable index declared in Search and used outside the loop.

Program 3.2 fixes this problem by introducing a different variable to serve as the index of the for loop.

```
-- Program 3.2
procedure Search (A   : Array_Type;
                  Val : Value_Type) is

    Index : Index_Type;

begin -- Search

    for I in Index_Type loop
        Index := I;
        exit when A(Index) = Val;
    end loop;

    if A(Index) = Val then
        Found (Index);
    else
        Not_Found;
    end if;

end Search; -- Program 3.2
```

Program 3.2 is basically correct, except that in the case of a null `Index_Type`, the value of `Index` will not be set. The only other stylistic disadvantage is the forced introduction of the loop variable `I` coupled with the extra assignment needed to set `index` to the value of `I`.

In Program 3.3, the need for both a loop variable and an outer variable to index the array is eliminated by moving the call on `Found` into the loop, so that the loop variable can be used in the call.

```

-- Program 3.3
procedure Search (A   : Array_Type;
                  Val : Value_Type) is

    Val_Found : Boolean := False;

begin -- Search

    for Index in Index_Type loop
        if A(Index) = Val then
            Val_Found := True;
            Found (index);
            exit;
        end if;
    end loop;

    if not Val_Found then
        Not_Found;
    end if;

end Search; -- Program 3.3

```

Although this program eliminates the extra loop variable, it does require setting a Boolean variable to decide at the end of the loop whether to call the Not_Found procedure. Unlike previous programs with exits, it requires the use of an exit statement without a when clause that is embedded inside a conditional statement. While in this simple case, the program is understandable, in more complicated cases, where the exit is buried deeper in nested logic, the exact conditions under which the exit is taken may not be clear.

3. EPILOGUE

The table below summarizes some of the important characteristics of the programs presented in this case study:

Characteristic	P r o g r a m									
	1.1	1.2	1.3	1.4	1.5	2.1	2.2	3.1	3.2	3.3
Correct?	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Clarity		Med	Low	Med	Med	Med	Med		High	Med
Works for integers		Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes
Works for enumerations?		No	Yes	Yes	No	Yes	Yes		Yes	Yes
Works for null types		Maybe	No	No	Maybe	No	No		No	Yes
Index same type as array index		No	Yes	Yes	Yes	Yes	Yes		Yes	Yes
Single entry, single exit?		Yes	Yes	Yes	Yes	No	Yes		No	No
Exits at top level of loop		N/A	N/A	N/A	N/A	Yes	N/A		Yes	No
No extra variables?		Yes	No	Yes	Yes	Yes	No		No	No
No extra value comparisons?		Yes	Yes	No	No	No	Yes		No	Yes
No other extra computation?		Yes	No	Yes	Yes	Yes	Yes		No	Yes

The most important characteristic of any program is that it be correct. Any program that is illegal or incorrect is clearly not acceptable. Thus Programs 1.1 and 3.1 are removed from further consideration.

After correctness, the most desirable characteristic of a program is clarity. An important contributing factor to clarity is simplicity. The relative clarity of the various programs examined here was subjectively estimated, based on the simplicity and appearance of the programs.

As shown in the table, all legal solutions handle integer index types. All but Programs 1.2 and 1.5 handle enumeration index types. If enumeration types are to be used to index the array, Programs 1.2 and 1.5 could be ruled out.

The importance of the program operating for null `Index_Type`'s is debatable. If the loop were being used in a single application it is probably alright if it does not handle the null type, since it is highly unlikely that null types would be used. On the other hand, if the Search routine was really being incorporated in a generic subprogram whose applications cannot be foreseen, the null type case should be handled properly. [In the table, the "Maybe" in the null type row for Programs 1.2 and 1.5 indicates that the programs work in the case that the lower bound of the null type is one greater than the upper bound of the null type.]

There is a certain amount of clarity and economy of constructs if the type for the variable index is the same as the type used to index the array. In addition to this advantage, a compiler may be able to avoid runtime checks for out-of-range indices if it knows that the types are the same.

Strict structured programming requires that all loops have only one exit point. The reason for this is to ensure that the exact circumstances for leaving a loop will be clear.

The Ada exit statement usually results in a loop that has more than one exit -- one normal exit at the termination due to a while or for clause and one for each exit statement. It is clear that unrestricted use of the exit statement can make it very difficult to determine exactly the conditions that cause loop exit. On the other hand, careful use of exits may be possible without sacrificing the ability to determine loop exit conditions. A reasonable position is that exits should be allowed, but only at the top level of the loop -- i.e. the exit statement is not nested within another loop or an if statement. This implies that the exits used will be of the form exit when, since an unconditional exit at the outer level of the loop will always cause loop termination. Only Program 3.3 violates this modified structured programming restriction.

Some programs seem to contain minor inefficiencies in the form of extra variables, extra value comparisons, or other extra computation. In the vast majority of applications, these inefficiencies are of no consequence. Only when an operation such as value comparison is extraordinarily expensive should much weight be given to these types of

inefficiencies. Also good optimizing compilers should be able to automatically eliminate much of the inefficiency. The moral of this is that the program should be programmed as clearly as possible and then measured to determine if it has acceptable efficiency. If a performance problem is found, then performance improvements can be examined.

Overall, the best of the programs examined here is Program 3.2, since it is the clearest and simplest. The for loop provides the most natural starting point for any operation that must sequence through elements of an array. In applications where the underlying data structure is not an array or when arrays are used but accessed in other than sequential order, it is likely that the while loop will be the most appropriate structure.

THIS PAGE INTENTIONALLY LEFT BLANK

2.3.4 USE OF BLOCK STATEMENTS FOR LOCAL RENAMING

1. BACKGROUND

Case Study Objective

To show how block statements containing renaming declarations can improve clarity and efficiency.

Problem

It is common for Ada programs to contain complex names, perhaps including function calls, selected components, and indexed components. Often these names refer to entities that are used repeatedly throughout some sequence of statements. Repeating the full complex name at each place the entity is used makes a program difficult to read and may lead to redundant computations that evaluate the complex names several times.

Discussion

Renaming declarations can be used to provide simple names with the same meaning as long, complex names. Since the complex names may involve pointer or index values that change during the execution of the sequence of statements, a block statement is needed to place the renaming declaration in the middle of a sequence of statements, so that the simple name takes on the current meaning of the complex name. This use of block statements and renaming declarations is analogous to the use of the with statement in Pascal.

2. DETAILED EXAMPLE

Example Problem Statement

A message switch receives, translates, and retransmits messages. If all transmission lines are in use when a message has been translated, the message switch may preempt the transmission of a lower priority message on one of the lines. Assuming that one or more lower priority messages are being transmitted, the message preempted is the lowest priority message currently being sent over a transmission line. If two or more messages are tied for the lowest priority, the one whose transmission started first is preempted.

A function named `Message_Precedence`, directly visible from the part of the program that selects a transmission line for preemption, takes a message object as an argument and returns that message's precedence. The package `Line_Package` has a function named `Line_Precedence` that takes a transmission line number as an argument and returns the precedence of the message currently being transmitted by that line; and a function named `Line_Start_Time` that takes a line transmission number as an argument and returns the time at which the line began transmitting the message with which it is currently occupied. The message switch control program maintains a linked list of numbers identifying transmission lines currently in use. This list is named `Busy_Line_List`.

The declarations relevant to the selection of a line to be preempted are as follows:

```
type Precedence_Type is
  (Routine, Priority, Immediate, Flash, ECP, Critical);
type Transmission_Line_Type is range 0 .. 50;

type Line_List_Cell_Type;
type Line_List_Cell_Pointer_Type is access Line_List_Cell_Type;
type Line_List_Cell_Type is
  record
    Head : Transmission_Line_Type;
    Tail : Line_List_Cell_Pointer_Type;
  end record;

Preempting_Precedence,
  Candidate_Precedence : Precedence_Type;
Preempting_Message     : Message_Type;
Candidate_Found         : Boolean;
Busy_Line_List,
  List_Pointer          : Line_List_Cell_Pointer_Type;
Candidate_Start_Time    : Time_Type;
Candidate               : Transmission_Line_Type;
```

The definitions of Message_Type and Time_Type are beyond the scope of this case study.

The following statements search for a transmission line to preempt after a message has been translated. The message is indicated by the variable Preempting_Message. Candidate_Found is set to True when there is a line eligible for preemption and False when there is not. When Candidate_Found is set to True, Candidate is set to the line number of the line to be preempted.

```
Preempting_Precedence := Message_Precedence (Preempting_Message);
Candidate_Found := False;
List_Pointer := Busy_Line_List;
```

```
while List_Pointer /= null loop
```

```
  if Line_Package.Line_Precedence(List_Pointer.Head) <
    Preempting_Precedence then
```

```
    if not Candidate_Found
```

```
      or else
```

```
      Line_Package.Line_Precedence(List_Pointer.Head) <
        Candidate_Precedence
```

```
      or else
```

```
      ( Line_Package.Line_Precedence(List_Pointer.Head) =
        Candidate_Precedence
```

```
        and then
```

```
        Line_Package.Line_Start_Time(List_Pointer.Head) <
          Candidate_Start_Time ) then
```

```
        Candidate_Found := True;
```

```
        Candidate := List_Pointer.Head;
```

```
        Candidate_Precedence :=
```

```
          Line_Package.Line_Precedence(List_Pointer.Head);
```

```
        Candidate_Start_Time :=
```

```
          Line_Package.Line_Start_Time(List_Pointer.Head);
```

```
      end if;
```

```
    end if;
```

```
    List_Pointer := List_Pointer.Tail;
```

```
  end loop;
```

Long names like Line_Package.Line_Precedence(List_Pointer.Head) clutter up the program, making it difficult to understand individual statements. In addition, they interfere with the formatting of the program, making its overall structure less obvious. Repeated use of these names is also inefficient. Values required in more than one place

cause functions to be invoked two or more times with the same arguments. Furthermore, each evaluation of the selected component `List_Pointer.Head` involves a redundant dereferencing operation. (Dereferencing is the process of obtaining an object from an access value designating that object.)

Solution Outline

A sequence of statements containing several occurrences of long names can be enclosed in a block statement. The declarative part of the block statement can contain renaming declarations for objects and constant declarations for values denoted by the long names. Within the block's sequence of statements, the longer names can be replaced by the identifiers declared within the block. Besides making the statements more succinct and easier to format, this transformation may eliminate redundant computation. The long names are evaluated once, during elaboration of the renaming declarations and constant declarations.

Detailed Solution

With the addition of a block statement, the selection of a line for preemption looks like this:

```
Preempting_Precedence := Message_Precedence (Preempting_Message);
Candidate_Found := False;
List_Pointer := Busy_Line_List;

while List_Pointer /= null loop

  declare

    Next_Line      : Transmission_Line_Type renames
                    List_Pointer.Head;
    Next_Precedence : constant Precedence_Type :=
                    Line_Package.Line_Precedence (Next_Line);
    Next_Start_Time : constant Time_Type :=
                    Line_Package.Line_Start_Time (Next_Line);

  begin

    if Next_Precedence < Preempting_Precedence then

      if not Candidate_Found
      or else
        Next_Precedence < Candidate_Precedence
      or else
        ( Next_Precedence = Candidate_Precedence
          and then
            Next_Start_Time < Candidate_Start_Time ) then

        Candidate_Found := True;
        Candidate := Next_Line;
        Candidate_Precedence := Next_Precedence;
        Candidate_Start_Time := Next_Start_Time;

      end if;

    end if;

  end; -- block statement

  List_Pointer := List_Pointer.Tail;

end loop;
```

In this version, Line_Package.Line_Start_Time is called exactly once during each passage through the loop, when the third declaration of the block statement is elaborated. The original version had one call at

the end of a short circuit control form in the condition of the inner if statement and one call in the last assignment statement inside that if statement. On any passage through the loop, either one of these calls, or neither of them, or both of them might be reached. Thus the version using the block statement may call `Line_Package.Line_Start_Time` more often than the original during certain passages through the loop and less often during other passages. The Epilogue shows how to expand the block statement to provide explicit control over the evaluation of the function call and avoid unnecessary or redundant calls.

To appreciate the full power of renaming declarations, it is necessary to consider the distinction between a value and an object. An object, that is, a variable or a constant, is a container holding a value. It makes sense to assign to an object, but not to a value. Most names refer to objects, but when a name occurs as part of an expression, it can be viewed as referring to the value contained in the named object. Constant declarations can be used to provide local names for values, but renaming declarations are needed to provide local names for variables.

In the example above, `Next_Line` could just as easily have been declared as a constant with the value `List_Pointer.Head`, since each occurrence of `Next_Line` refers to the value contained in the object, not the object itself. In contrast, consider the following declarations:

```
Message_Count   : array (Precedence_Type) of Natural :=
                    (Precedence_Type => 0);
Current_Message : Message_Type;
```

If the purpose of `Message_Count` is to keep track of the number of messages of each precedence level that have been processed, the program might contain the following statement:

```
Message_Count ( Message_Precedence(Current_Message) ) :=
    Message_Count ( Message_Precedence(Current_Message) ) + 1;
```

The two calls on `Message_Precedence` are, of course, redundant. The redundancy can be eliminated as follows:

```
declare

    Precedence_Level_Message_Count : Natural renames
        Message_Count ( Message_Precedence(Current_Message) );

begin

    Precedence_Level_Message_Count :=
        Precedence_Level_Message_Count + 1;

end;
```

In this case, Precedence_Level_Message_Count becomes another name for the object named by the name

Message_Count (Message_Precedence(Current_Message)).

Thus the local name may appear on the left-hand side of an assignment statement.

3. EPILOGUE

The use of block statements with renaming declarations, as described above, is similar to the use of the with statement in Pascal. However, the Ada construct is more flexible. Given the Pascal type declarations

```
Complex = record Re, Im: real end;  
ComplexPtr = ^Complex;  
ComplexPtrTable = array [1 .. 10] of ComplexPtr;
```

and the variable declaration

```
Table: ComplexPtrTable
```

along with a function *f* taking an integer as an argument and returning an integer, the following with statement could be used to negate *Table[f(3)][^].Re* and *Table[f(3)][^].Im*:

```
with Table[f(3)]^ do  
  begin  
    Re := -Re;  
    Im := -Im  
  end;
```

In Ada, the corresponding declarations are:

```
type Complex is  
  record  
    Re, Im : Float;  
  end record;  
type Complex_Ptr is access Complex;  
type Complex_Ptr_Table is array (1 .. 10) of Complex_Ptr;  
  
Table : Complex_Ptr_Table;
```

The effect of the with statement is achieved by the following block statement:

```
declare  
  
  Complex_Object : Complex renames Table ( f(3) ) . all;  
  
begin  
  
  Complex_Object.Re := -Complex_Object.Re;  
  Complex_Object.Im := -Complex_Object.Im;  
  
end;
```

In fact, block statements with renaming declarations are more flexible than Pascal with statements. For example, they can simultaneously provide shortened names for components of several records in the same record type, as in this example:

```
declare

    Complex_Object_1 : Complex renames Table ( f(1) ) . all;
    Complex_Object_2 : Complex renames Table ( f(2) ) . all;

begin

    Complex_Object_2.Re := Complex_Object_1.Re;
    Complex_Object_2.Im := - Complex_Object_1.Im;

end;
```

Furthermore, Pascal forbids a with statement from altering a variable that is part of the name following the word with. Thus, given the Pascal type declarations

```
IntegerList = ^IntegerListCell;
IntegerListCell =
    record
        Head: integer;
        Tail: IntegerList
    end;
```

and a variable L of type IntegerList, the following statement is illegal:

```
while L <> nil do
    with L^ do
        begin
            Head := Head + 1;
            L := Tail;
        end;
```

However, given the Ada declarations

```
type Integer_List_Cell;
type Integer_List is access Integer_List_Cell;
type Integer_List_Cell is
    record
        Head: Integer;
        Tail: Integer_List;
    end record;
L : Integer_List;
```

there is no problem with the following statement:

```

while L /= null loop
  declare
    Next_Cell : Integer_List_Cell renames L.all;
  begin
    Next_Cell.Head := Next_Cell.Head + 1;
    L := Next_Cell.Tail;
  end;
end loop;

```

The assignment of a new value to L at some time after the elaboration of the renaming declaration does not affect the meaning of the name Next_Cell, so the semantics of the assignment are well-defined.

In some cases, an optimizing compiler may be able to avoid redundant computations resulting from several occurrences of the same long names. However, this optimization is difficult because the compiler must establish that the meaning of the name does not change from one evaluation to another. In particular, it may be necessary for the compiler to ascertain that certain function subprograms always return the same value when called with the same arguments, or that certain subprograms do not have side-effects that could affect the meaning of the name.

The Detailed Solution presented a revised preemption algorithm that eliminates certain redundant computations, but may introduce redundant calls on the function Line_Package.Line_Start_Time. The number of times this function is called can be minimized by a conditional statement which exerts detailed control over the times at which certain expressions get evaluated:

```

Preempting_Precedence := Message_Precedence (Preempting_Message);
Candidate_Found := False;
List_Pointer := Busy_Line_List;

while List_Pointer /= null loop

    declare

        Next_Line      : Transmission_Line_Type renames
                        List_Pointer.Head;
        Next_Precedence : constant Precedence_Type :=
                        Line_Package.Line_Precedence (Next_Line);
        Next_Start_Time : Time_Type;
        Next_Line_Better : Boolean;

    begin

        if Next_Precedence >= Preempting_Precedence then
            Next_Line_Better := False;
        elsif not Candidate_Found then
            Next_Start_Time := Line_Package.Line_Start_Time (Next_Line);
            Next_Line_Better := True;
        elsif Next_Precedence < Candidate_Precedence then
            Next_Start_Time := Line_Package.Line_Start_Time (Next_Line);
            Next_Line_Better := True;
        elsif Next_Precedence = Candidate_Precedence then
            Next_Start_Time := Line_Package.Line_Start_Time (Next_Line);
            Next_Line_Better := Next_Start_Time < Candidate_Start_Time;
        else
            Next_Line_Better := False;
        end if;

        if Next_Line_Better then
            Candidate_Found := True;
            Candidate := Next_Line;
            Candidate_Precedence := Next_Precedence;
            Candidate_Start_Time := Next_Start_Time;
        end if;

    end; -- block statement

    List_Pointer := List_Pointer.Tail;

end loop;

```

(The second and third arms of the if statement can be replaced by a single arm of the following form:

```
elsif not Candidate_Found or else
    Next_Precedence < Candidate_Precedence then
        Next_Start_Time := Line_Package.Line_Start_Time (Next_Line);
        Next_Line_Better := True;
```

While this saves keystrokes, the use of two arms is more uniform and is thus easier to understand.) This version only calls Line_Package.Line_Start_Time when a better candidate for preemption is found (in which case it is needed to update Candidate_Start_Time) or when the result of the function call is necessary to determine whether the next transmission line is a better candidate for preemption.

2.4 Exceptions

This section contains the following case study:

The Use of Exceptions

THIS PAGE INTENTIONALLY LEFT BLANK

2.4.1 THE USE OF EXCEPTIONS

1. BACKGROUND

Case Study Objective

To classify and illustrate various uses of exceptions and to present arguments for and against each use.

Problem

One of the features that sets Ada apart from other languages is the ability to declare, raise, and handle exceptions. Exceptions were included in Ada in order to facilitate the design of fault-tolerant hardware/software systems. However, exceptions are a very versatile and powerful tool, and they may be useful for other purposes as well.

There are many questions about how exceptions can best be used to foster program reliability. When should exceptions be raised? When should exceptions be handled, when should they be propagated, and when should they be handled and then re-raised? What should an exception handler do? Where should exceptions be declared? Does it ever make sense to raise a language-defined exception with a raise statement? What are the relative merits of a subprogram signaling a failure to its caller by setting a status code and by raising an exception? How safe is it to presume that we understand the underlying cause of an exception when we undertake to handle it? Can we distinguish between anticipated and unanticipated exceptions?

Exceptions can be viewed not only as an error-handling mechanism, but as a control structure. Often, exceptions can be raised deliberately and used in a planned way to control the flow of execution. It is important to consider the effects of this practice on program readability, program reliability, and perhaps on program efficiency.

Discussion

This case study explores six different uses for exceptions. In order of consideration, they are:

- (1) The use of exceptions to implement a control structure. The raising of an exception may be part of the normal flow of control.

- (2) The use of exceptions to handle special situations that the programmer fully expects to occur from time to time. It may be that such situations, if allowed to occur, result in the raising of an exception, but that the proper way to resume the computation is clear.
- (3) The use of exceptions in general purpose software components. It may not be assumed that such components will always be invoked as required by their specifications, and exceptions allow the component to respond in a well-defined way when it is invoked improperly.
- (4) The use of exceptions to respond to input data in an invalid format.
- (5) The use of exceptions to respond to a hardware malfunction and allow the hardware/software system to continue working sensibly or to die gracefully.
- (6) The use of exception handlers to respond to an unanticipated software error.

The order in which these cases are considered is deliberate. We start out with scenarios in which the program is very much in control of what is happening, and exceptions are a routine occurrence. In subsequent scenarios, exceptions are progressively less routine, and more indicative of something amiss.

2. DETAILED EXAMPLES

Example Problem Statement (1)

Zahn's construct, first proposed in [Zah74] and refined in [Knu74], is a control structure allowing the elimination of the most common uses of goto statements. In particular, Zahn's construct provides a goto-less way to implement either decision trees with duplicate leaves or loops with multiple exits in which the action to follow the loop is determined by which exit was taken. As described in [Zah74], the construct has the following syntax:

```
until <event name> or ... or <event name> do
  <sequence of statements>
then case
  <event name> : <sequence of statements>
  ...
  <event name> : <sequence of statements>
```

The first line does not perform any sort of test. Rather, it serves to declare certain identifiers as names of "events" that might occur during execution of the statements preceding the word then, causing that sequence of statements to be terminated. Following the word case, there must be exactly one sequence of statements labeled with each of these event names. The sequence of statements preceding the word then may contain statements of the form

<event name> ;

Such a statement terminates execution of the sequence of statements and passes control to that sequence of statements after the word case corresponding to the named event. It is illegal for the first sequence of statements to complete without executing such a statement.

Consider a program that is to keep track of the number of times certain key values occur in the input. Every time a key value is obtained, a table is searched. If the key is found in the table, a corresponding count in the table is incremented. If the key is not found, the variable Others_Count is incremented instead. Given the declarations

```

type Table_Entry_Type is
  record
    Key_Part      : Key_Type;
    Count_Part    : Natural := 0;
  end record;

type Table_Type is array (Positive range <>) of Table_Entry_Type;

Key           : Key_Type;
Frequency_Table : Table_Type (1 .. Table_Size);
Position      : Integer range 1 .. Table_Size;
Others_Count  : Natural := 0;

```

the frequency counts could be maintained using Zahn's construct as follows:

```

until Found or Not_Found do

  for I in Frequency_Table'Range loop
    if Frequency_Table(I).Key_Part = Key then
      Position := I;
      Found;
    end if;
  end loop;
  Not_Found;

then case

  Found      : Frequency_Table(Position).Count_Part :=
                Frequency_Table(Position).Count_Part + 1;

  Not_Found : Others_Count := Others_Count + 1;

```

Solution Outline (1)

It is easy to implement Zahn's construct in Ada by letting exceptions play the role of events. The resulting Ada code corresponds closely to Zahn's construct. The implementation is as follows:

The sequence of statements containing multiple exits is enclosed in a block statement. For each "event" that can cause the sequence of statements to be exited, an exception is declared local to the block. The block contains a handler for each such exception, specifying the follow-up actions to be taken when that event occurs. Statements of the form

<event name>

are replaced by raise statements for the corresponding exceptions.

Detailed Solution (1)

Following this scheme, the code given above using Zahn's construct is written in Ada as follows:

```
declare

    Found, Not_Found : exception;

begin

    for I in Frequency_Table'Range loop
        if Frequency_Table(I).Key_Part = Key then
            Position := I;
            raise Found;
        end if;
    end loop;
    raise Not_Found;

exception

    when Found =>
        Frequency_Table(Position).Count_Part :=
            Frequency_Table(Position).Count_Part + 1;

    when Not_Found =>
        Others_Count := Others_Count + 1;

end;
```

Clearly, this is not the kind of use for which exceptions are primarily designed. However, Zahn's construct is a very powerful control structure, applicable in many of the unusual situations in which no other control structure applies. Exceptions provide nearly a direct translation of this construct into Ada.

An Ada compiler may be astute enough to notice that the exceptions Found and Not_Found are handled locally every place they are raised, and translate the raise statements as unconditional branches. However, if this is not the case, then this translation of Zahn's construct could well be less efficient than alternatives using Boolean variables or redundant tests. Of course, if this use of exceptions becomes a common coding paradigm, the ability to translate it efficiently will become an important benchmark of Ada compilers.

This example can also be implemented using a goto, as follows:

```

for I in Frequency_Table'range loop
  if Frequency_Table(I).Key_Part = Key then
    Frequency_Table(I).Count_Part :=
      Frequency_Table(I).Count_Part + 1;
    goto Found;
  end if;
end loop;

<< Not_Found >>
  Others_Count := Others_Count + 1;

<< Found >>
  null;

```

Zahn's construct is superior to this formulation because it delimits a simple one-entry/one-exit control structure. In the version using a goto statement, it is not as clear that the flow of control involves a loop followed by actions that depend on how the loop was exited. In fact, the follow-up actions in the case that the key is found must be incorporated in the loop body.

Another relevant issue is the enforcement of coding standards. Allowing use of the goto in this restricted circumstance would require those who enforce coding standards to distinguish this use of the goto from other, less justifiable uses. A simple ban on goto statements is much easier to enforce.

Example Problem Statement (2)

Several tasks executing concurrently are required to obtain unique identifying values called asynchronous serial numbers, or ASN's. A package named ASN_Package provides a private type named Serial_Number_Type and a task named ASN_Manager. This task generates Serial_Number_Type values, providing a new value each time it is called; but after one million values have been generated, the task may reuse previously generated values, in the same order. The only operations on Serial_Number_Type values are generating them by calling the entry ASN_Manager.Get_ASN, assigning them, and comparing them for equality and inequality.

Solution Outline (2)

ASN's are represented internally as integers. A counter in the ASN_Manager task is incremented each time an ASN is generated, so that it holds the value that will be provided the next time entry Get_ASN is called; but when the counter reaches 1,000,000, it "wraps around" to 1 instead of being incremented. The wrap-around condition is processed by an exception handler.

Detailed Solution (2)

The specification and body of ASN_Package are as follows:

```
package ASN_Package is

  type Serial_Number_Type is private;

  task ASN_Manager is
    entry Get_ASN (ASN : out Serial_Number_Type);
  end ASN_Manager;

private

  type Serial_Number_Type is range 1 .. 1_000_000;

end ASN_Package;
```

```

package body ASN_Package is

  task body ASN_Manager is

    Counter : Serial_Number_Type := 1;

  begin -- ASN_Manager

    loop

      accept Get_ASN (ASN : out Serial_Number_Type) do
        ASN := Counter;
      end Get_ASN;

      begin
        Counter := Counter + 1;
      exception
        when Constraint_Error =>
          Counter := 1;
      end;

    end loop;

  end ASN_Manager;

end ASN_Package;

```

When the block statement is entered with Counter initially equal to 1,000,000, the attempt to store 1,000,001 in Counter raises Constraint_Error and the handler is executed to set Counter to 1. In all other cases, Counter is incremented normally and the handler is not executed.

The validity of this approach rests subtly on the assumption that Serial_Number_Type'Base'Last does not equal 1,000,000, in other words, that the underlying representation of Serial_Number_Type has sufficient capacity to hold the temporary result 1,000,001. If this is not the case, an exception may be raised when adding 1 to Counter rather than when trying to store the result of this addition back in Counter. Then the exception raised will be Numeric_Error, not Constraint_Error. Since there is no handler for this exception, the task will terminate. Admittedly, we are not likely to find an implementation in which 1,000,000 is the highest value in some predefined integer type (and thus in Serial_Number_Type'Base). Nonetheless, it is easy to envision a maintenance programmer, given the task of expanding the wrap-around interval because duplicate ASN's were found to be in use at the same time, redefining Serial_Number_Type as

```

type Serial_Number_Type is new Positive;

```

in order to guarantee that full use is made of the capacity of the

underlying representation.

There are easy ways to fix this problem. One is to invoke the handler when either `Constraint_Error` or `Numeric_Error` is raised. Another is to redefine `Serial_Number_Type` as

```
type Serial_Number_Type is range 1 .. 1_000_001;
```

and redeclare `Counter` as

```
Counter: Serial_Number_Type range 1..Serial_Number_Type'Last-1 := 1;
```

(In this case, the opportunity for error arose from the possibility of the same situation resulting in more than one exception. Because the partition of anomalous situations into predefined exceptions is so coarse, we are more likely in the long run to see subtle errors arising for just the opposite reason — a given exception being raised because of a situation other than that which the programmer anticipated when he wrote a handler.)

It is not clear why exception handling, with its subtle opportunities for error, should be used to solve the wrap-around problem when other, more obvious, mechanisms are available. The assignment statement

```
Count := (Count mod 1_000_000) + 1;
```

always sets `Count` to its proper cyclic successor, and cannot raise `Numeric_Error`. Even more straightforward are the compound statement

```
if Count = 1_000_000 then
  Count := 1;
else
  Count := Count + 1;
end if;
```

or a for loop nested inside a basic loop:

```
loop
  for I in Serial_Number_Type loop
    accept Get_ASN (ASN : out Serial_Number_Type) do
      ASN := I;
    end Get_ASN;
  end loop;
end loop;
```

Perhaps the programmer felt that the internal check for `Constraint_Error` was somehow more efficient than an explicit test, or that the internal check was going to be performed anyway, so that it

would be pointless to duplicate it with an explicit check. It is rarely useful, and sometimes counterproductive, to second-guess the compiler in this way. Unlike the test `Count = 1_000_000`, the internal check for constraint error may require two comparisons, with the upper and lower bounds of `Count`'s range. If the programmer adopts one of the alternative approaches above, an optimizing compiler may well be able to determine that `Constraint_Error` cannot be raised, and omit the internal check.

Example Problem Statement (3)

Because stacks are used in many different programs, a generic stack package is to be written. The package will provide operations to push elements onto a non-full stack, pop elements off a non-empty stack, create an empty stack, determine whether a stack is empty, and determine the top element in a non-empty stack. The type of the elements and the capacity of the stack are specified as generic parameters.

Solution Outline (3)

Certain stack operations are undefined for certain stack values. An element cannot be pushed onto a full stack or popped off an empty stack. Similarly, the top element of an empty stack cannot be computed.

The stack package is written as a general, reusable software component. Because the writer of the generic package has no idea how the package will be used, he may not assume that these undefined operations will never be attempted. Rather, it is his responsibility to "idiot-proof" his package so that it reacts sensibly whenever one of its subprograms is called. When the subprogram call calls for an undefined operation to be performed, the sensible reaction is to raise an appropriate exception.

Detailed Solution (3)

The specification of the generic stack package is as follows:

generic

type Element_Type is private;
Capacity : in Integer;

package Stack_Package is

type Stack_Type is private;

Empty_Stack_Error, Full_Stack_Error : exception;

function Empty_Stack return Stack_Type;

procedure Push (Element : in Element_Type;
Stack : in out Stack_Type);
-- May raise Full_Stack_Error.

procedure Pop (Element : out Element_Type;
Stack : in out Stack_Type);
-- May raise Empty_Stack_Error.

function Is_Empty (Stack : Stack_Type) return Boolean;

-- May raise Empty_Stack_Error.

private

type Element_Array_Type is array (1 .. Capacity) of Element_Type;

type Stack_Type is
record

Top_Pointer_Part : Integer range 0 .. Capacity := 0;

Element_Array_Part : Element_Array_Type;

end record;

end Stack_Package;

Push raises Full_Stack_Error when called with Stack.Top_Pointer_Part equal to Capacity. Pop and Top_Element raise Empty_Stack_Error when called with Stack.Top_Pointer_Part equal to 0.

Should Push, Pop, and Top_Element fail to check for these anomalies, following the normal course of processing would cause the language-defined exception Constraint_Error to be raised. Because there are so many different ways in which Constraint_Error can be raised, this would make it more difficult for the user of Stack_Package to discern the cause of the exception. Even if he managed to determine that the exception was propagated by one of Stack_Package's subprograms, he would not know whether its cause was an improper call on that subprogram or an internal malfunction.

In addition, the language-defined exception reports the problem to the user at the wrong level of abstraction. Constraint_Error is raised when an array is indexed with an invalid component, among other situations, but the user is not concerned with the fact that the stack is implemented as an array. The external behavior of Stack_Package, including the exceptions it raises, should be independent of its implementation. In general, whenever an abstract data type is defined with operations that are not meaningful for all values, the data type should have its own exceptions, describing the undefined operations in abstract terms.

In contrast, the exceptions Full_Stack_Error and Empty_Stack_Error form part of the interface of Stack_Package, and clearly indicate a call with improper arguments. Two separate exceptions are provided in order to allow the user to distinguish between different kinds of attempted illegal operations. It can be argued that Empty_Stack_Error should be further decomposed into two exceptions, one raised by Pop and one raised by Top_Element. In contrast, it can also be argued that Full_Stack_Error and Empty_Stack_Error should be merged into a single exception, Stack_Error, since the cause of the error is uniquely identified by the subprogram that was called. Like other decisions about levels of abstraction, this is a matter of taste. How distinct, from the user's point of view, are the three situations in which exceptions may be raised?

The declaration of the exceptions in the visible part of the package rather than the package body allows them to be named by the user of Stack_Package. Without this ability, the exceptions would still be propagated to the caller, but could not be handled except by a "when others" handler. There would be no point in considering the declaration of more than one exception, since there would be no way to distinguish among the exceptions outside of Stack_Package.

Example Problem Statement (4)

A human-operated embedded system executes an initialization routine when it is started up. One of the duties of this routine is to obtain the current time of day from the operator. This involves a dialogue at the operator console using the facilities of package Text_IO.

The parameterless function Operator_Entered_Time obtains the time of day from the operator in the form hours:minutes:seconds or hours:minutes and returns a value of type Duration representing that time in seconds since midnight. This function is responsible for prompting the operator and for detecting invalid input. In the case of invalid input, the function conducts a dialogue with the operator until a valid input is obtained.

Solution Outline (4)

It is assumed that the operator console keyboard is the default input file and the operator console display is the default output file. An entire line containing the operator's entry is obtained using the procedure Text_IO.Get_Line, and then analyzed. The colons are located and the fields separated by the colons are then converted to type Integer values using a version of Get that reads an integer literal from a string.

If the fields are not valid signed or unsigned numeric literals, Get raises the exception Data_Error. This exception must be handled as an indication of invalid input data, just like a missing colon or an out-of-range value for hours, minutes, or seconds. Because Get regards all signed Ada integer literals as valid, input like "+23 : 1E1 : 16#F#" is interpreted as valid (in this case, equivalent to "23:10:15"), but this seems harmless.

Detailed Solution (4)

The function Operator_Entered_Time and two supporting subprograms are enclosed in a package whose declaration and body are as follows:

```
package Operator_Time_Package is

    function Operator_Entered_Time return Duration;

end Operator_Time_Package;
```

```

package body Operator_Time_Package is

  procedure Search_For_Colon
    ( Search_String      : in String;
      Starting_Position  : in Positive;
      Found              : out Boolean;
      Colon_Position     : out Positive )
    is separate;

  procedure Interpret_Time_String
    ( Time_String      : in String;
      Valid_Time       : out Boolean;
      Time_In_Seconds  : out Natural )
    is separate;

  function Operator_Entered_Time return Duration
    is separate;

end Operator_Time_Package;

```

The function `Operator_Entered_Time` conducts the dialogue with the operator, issuing prompts and obtaining input, and converts the time of day in seconds from type `Integer` (subtype `Natural`) to type `Duration`. It calls `Interpret_Time_String` to validate and decode the sequence of characters obtained from the operator. The sequence of characters is passed to `Interpret_Time_String` through the parameter `Time_String`. `Interpret_Time_String` sets `Valid_Time` to indicate whether the characters form a valid time of day. If they do, `Time_In_Seconds` is set to that time; otherwise, `Time_In_Seconds` is set to an arbitrary value of subtype `Natural`. `Search_For_Colon` is called by `Interpret_Time_String`. The body of `Operator_Entered_Time` is as follows:

```

with Text_IO;

separate (Operator_Time_Package)

function Operator_Entered_Time return Duration is

    Input_Line      : String (1 .. 80);
    Input_Length    : Integer range 0 .. 80;
    Time_In_Seconds : Natural;
    Valid_Input     : Boolean;

begin -- Operator_Entered_Time

    Text_IO.Put ("Please enter time of day (HH:MM:SS or HH:MM):");

    loop

        Text_IO.Get_Line ( Input_Line, Line_Length );

        Interpret_Time_String
            ( Input_Line (1 .. Line_Length),
              Valid_Input,
              Time_In_Seconds );

        exit when Valid_Input;  --> LOOP EXIT <--

        Text_IO.Put ("Invalid time. Please re-enter:");

    end loop;

    return Duration ( Time_In_Seconds );

end Operator_Entered_Time;

```

It is within the body of `Interpret_Time_String` that input data is analyzed and invalid input is recognized. This is where exceptions come into play. `Interpret_Time_String` calls `Search_For_Colon` to find the colons in the input string. The string to be searched, and the position at which the search is to start, are provided through the parameters `Search_String` and `Starting_Position`, respectively. `Found` is set to indicate whether or not a colon was found. If so, `Colon_Position` is set to the position of the first such colon. If not, `Colon_Position` is set to an arbitrary value of subtype `Positive`. The body of `Interpret_Time_String` is as follows:

```

with Text_IO;

separate (Operator_Time_Package)

procedure Interpret_Time_String ( Time_String      : in String;
                                   Valid_Time       : out Boolean;
                                   Time_In_Seconds  : out Natural) is

    Found                : Boolean;
    Colon_1, Colon_2, Last : Integer range Time_String'Range;
    Hours, Minutes, Seconds : Natural;

    package Type_Integer_IO is new Text_IO.Integer_IO ( Integer ) ;

```

```

begin -- Interpret_Time_String

    Time_In_Seconds := 0; -- arbitrary value to avoid constraint
                          -- error in case no other value is
                          -- assigned later

    Search_For_Colon (Time_String, 1, Found, Colon_1);

    if Found then

        Type_Integer_IO.Get (Time_String(1 .. Colon_1-1), Hours, Last);

        Search_For_Colon (Time_String, Colon_1+1, Found, Colon_2);

        if Found then
            Type_Integer_IO.Get
                (Time_String (Colon_1+1 .. Colon_2-1), Minutes, Last );
            Type_Integer_IO.Get
                (Time_String (Colon_2+1 .. Time_String'Last),
                 Seconds,
                 Last );
        else -- Input was hours:minutes, not hours:minutes:seconds.
            Type_Integer_IO.Get
                (Time_String (Colon_1+1 .. Time_String'Last),
                 Minutes,
                 Last );
            Seconds := 0;
        end if;

        if Hours in 0..23 and Minutes in 0..59 and Seconds in 0..59 then
            Valid_Time := True;
            Time_In_Seconds := Seconds + 60 * (Minutes + 60 * Hours);
        else
            Valid_Time := False;
        end if;

    else -- No colon found.

        Valid_Time := False;

    end if;

exception -- Interpret_Time_String

    when Type_Integer_IO.Data_Error =>
        Valid_Time := False;

end Interpret_Time_String;

```

When one of the fields of Time_String separated by colons does not contain a syntactically valid optionally signed integer literal, a call

on `Type_Integer_IO.Get` raises the exception `Type_Integer_IO.Data_Error`. Furthermore, this is the only way this exception can be raised in `Interpret_Time_String`. Thus the exception can be handled simply by returning from `Interpret_Time_String` with an indication of an invalid time string.

Exceptions could also have been used to detect out-of-range values for syntactically valid hour, minute and second fields. The variables `Hours`, `Minutes`, and `Seconds` could have been declared with range constraints of `0 .. 23`, `0 .. 59`, and `0 .. 59`, respectively. Then the procedure call statements calling `Type_Integer_IO.Get` (as opposed to the body of `Type_Integer_IO.Get` itself) would have raised `Constraint_Error` when trying to place an out-of range value in one of these variables. Alternatively, `Text_IO.Integer_IO` could have been instantiated with new integer types having these ranges. Then the resulting instances of `Get` (the procedures themselves) would raise `Data_Error` when reading out-of-range values.

However, `Interpret_Time_String` is most easily understood when the validity checks, and the way in which they affect the flow of control, are made explicit. This is in part due to the fact that validity checks are one of the principal functions of this subprogram. Rather than hiding the processing of special cases and thus uncluttering the normal processing algorithm, the further use of exceptions would hide the normal processing algorithm, which is a check for invalid data.

The only reason for using exceptions at all is that this is the only way that `Get` communicates the fact that it was given invalid string data. It can be argued that `Get` should not be used at all, but that the digits in the string should be validated and examined explicitly to compute the corresponding numeric value. This approach would make it possible to invalidate signed, exponential, and based numbers in a time string. The counter-argument is that a subprogram to validate and convert numeric strings -- `Get` -- has already been written and widely field-tested. Considerations of software reliability and economy dictate that this effort should not be duplicated.

We present the body for `Search_For_Colon` at this point to illustrate one final consideration:

separate (Operator_Time_Package)

```
procedure Search_For_Colon (Search_String      : in String;  
                           Starting_Position  : in Positive;  
                           Found              : out Boolean;  
                           Colon_Position     : out Positive) is
```

```
    Position : Positive := Starting_Position;
```

```
begin -- Search_For_Colon
```

```
    while Position <= Search_String'Last and then  
        Search_String ( Position ) /= ':' loop  
        Position := Position + 1;  
    end loop;
```

```
    if Position <= Search_String'Last then  
        Colon_Found := True;  
        Colon_Position := Position;  
    else  
        Colon_Found := False;  
        Colon_Position := 0; -- arbitrary value  
    end if;
```

```
end Search_For_Colon;
```

In general, even when a programmer thinks he knows why an exception may occur within some program unit, it is wise to have the handler for that exception apply to only a small region of the program unit. Because `Data_Error` is a specialized exception that can only be raised by certain input/output operations, and because `Interpret_Time_String` is a short program unit containing only three such operations, it was safe to put the handler for `Integer_Type_IO` in the exception part of the procedure body. Had `Constraint_Error` been used as described earlier to detect out-of-range hour, minute, and second values, it would have been wise to enclose the inner "if Found" statement, or perhaps even each individual call on `Type_Integer_IO.Get`, in a block statement with its own handler for `Constraint_Error`.

`Constraint_Error` is a very general exception that can arise in almost any kind of Ada statement due to subtle programming errors. Indeed, had the assignment of the arbitrary value 0 to `Colon_Position` been omitted from `Search_For_Colon`, the second call of `Search_For_Colon` in `Interpret_Time_String` could have raised `Constraint_Error` even for valid time strings of the form hours:minutes. (This is because the initial value of an out formal parameter of a scalar type is undefined, and a constraint check is performed when the formal parameter is copied back into the actual parameter, possibly on this undefined value. The value may or may not pass this check.) The subtlety of this problem and the possibility that it will only manifest itself intermittently make it

a very plausible latent error. The use of small block statements provides confidence that the constraint error being handled resulted from the cause anticipated when the handler was written. One reasonable way to handle Constraint_Error within this block statement is to raise a more specialized exception, say Time_String_Error, that is declared within Search_For_Colon and handled at the level of the procedure body.

Example Problem Statement (5)

A low-level task in a message switch is responsible for sending individual characters to the serial interface for transmission. After sending a character, the task must wait for an interrupt from the serial interface indicating that it is ready to transmit another character. If this interrupt fails to arrive in the expected amount of time, it is presumed to be indicative of a hardware malfunction.

There are many responses, at many different levels of processing, that must be taken in response to such a malfunction. The procedure that traverses the internal representation of the message and sends it character-by-character to the low-level task must determine the point in the message at which the malfunction occurred. It may retry the transmission of the most recent character some number of times, in the hope that the hardware problem is transient. If this fails, the procedure will deduce how much of the message must be sent again later. (This depends on which parts of the message got through before the malfunction occurred.) The table indicating which physical lines are working must be updated to reflect the unusable line. If there is an alternate physical line with the same destination, the program must "install" the alternate line by updating the table indicating which physical lines are being used to carry the messages routed to a given logical line. The message switch operator must be notified of the malfunction. If alternate physical lines are not available and the message switch is part of a network, other nodes in the communications network must be notified that one of the links in the network is unavailable. Messages already sent to the afflicted message switch may have to be re-routed.

Solution Outline (5)

The low-level task explicitly raises an exception, Serial_Interface_Malfunction, when it fails to receive an interrupt from the serial interface in the required amount of time. Exceptions describing the problem are propagated up many levels of processing, until an exception is raised in the program unit having ultimate responsibility for all the actions that are to be taken in response to the malfunction. At each level of processing, a handler for the exception takes whatever action is appropriate at its level and then re-raises whatever exception best describes the problem to the next higher level.

Detailed Solution (5)

The low-level task, Transmitter, is declared in an instantiation of the generic package Transmitter_Package. This package is instantiated for each serial interface in the message switch, with generic parameters indicating the device address and interrupt address for the serial

interface. The generic package specification is as follows:

```
with System;
```

```
generic
```

```
    Device_Address, Interrupt_Address : in System.Address;  
    Interval_Between_Chars           : in Duration;
```

```
package Transmitter_Package is
```

```
    task type Transmitter is
```

```
        entry Transmit_Character(Char : in Character);  
        entry Serial_Interface_Interrupt;  
        for Serial_Interface_Interrupt use at Interrupt_Address;  
    end Transmit_Character;
```

```
    Serial_Interface_Malfunction : exception;
```

```
end Transmitter_Package;
```

The package body provides a code procedure to perform memory-mapped output to the serial interface by storing a byte at the device address. The task body for Transmitter is straightforward. Here is the package body:

```

with Machine_Code;

package body Transmitter_Package is

  procedure Hardware_Output (Byte : in Character) is
    use Machine_Code;
  begin -- Hardware_Output
    Machine_Instruction'(Load_Byte, R1, Byte'Address);
    Machine_Instruction'(Store_Byte, R1, Device_Address);
  end Hardware_Output;

  task body Transmitter is

  begin -- Transmitter

    loop
      begin
        accept Transmit_Char ( Char : in Character ) do
          Hardware_Output ( Char );
          select
            accept Serial_Interface_Interrupt;
          or
            delay Interval_Between_Chars;
            raise Serial_Interface_Malfunction;
          end select;
        end Transmit_Char;
      exception
        when Serial_Interface_Malfunction =>
          null;
      end;
    end loop;

  end Transmitter;

end Transmitter_Package;

```

Normally, an exception raised within an accept statement is propagated both to the task containing the accept statement and to the calling task. The timed wait that receives an interrupt or raises an exception is nested in the accept statement for Transmit_Char so that, from the perspective of the calling task, a call on Transmit_Char raises an exception when transmission of the character it was to send is not acknowledged by an interrupt. Within the Transmitter task, the exception is handled by a null exception handler inside the task loop. In effect, the exception is ignored within the task. The task remains alive so that the calling task can retry the transmission.

The paradigm of an accept statement surrounded by a block statement with a null exception handler is applicable whenever an exception is to be raised in the calling task but ignored in a called task. This will

be the case whenever the called task is serving many other tasks, for example. This paradigm is especially appropriate when the called task detects inappropriate parameter values in an entry call.

When a program unit invokes a subprogram to carry out some actions on its behalf, the subprogram reports an anomaly by raising but not handling an exception. Execution of the subprogram terminates and the exception is propagated to the calling program unit. When a program unit invokes a task to carry out some actions on its behalf, the task reports an anomaly by raising an exception in an accept statement. From the calling task's point of view, it is the entry call that raises the exception. Depending on the purpose of the task, the task can be written to terminate when it raises the exception or, as in the case of task Transmitter, to handle the exception internally with a null handler and continue execution. The invoking task may then choose to terminate the task that raised the exception, by a call to a special "Please terminate" entry or by an abort statement. In summary, whether a program unit uses a subprogram or a task as an agent to carry out some actions, there is a mechanism for the agent to communicate failure to its "boss," so that the failure can be communicated to successively higher levels of abstraction.

The level to which the exception is to be propagated, and the actions to be taken at each intermediate level, must be determined when the system is designed, not left to the programmers of individual modules. This will assure that every module that must react to the exception has a chance to do so, and that the exception will not be raised in a module that is not prepared to handle it. The actions to be taken at intervening levels may include attempts at recovery (retrying the transmit operation at a low level, finding an alternate transmission line at an intermediate level, or reconfiguring the network at a high level, for example), modifications to data structures, and "cleanup" actions to assure that files and data structures are left in a consistent state. The handlers for Serial_Interface_Malfunction at these intervening levels will take the following form:

```
when Serial_Interface_Malfunction =>  
  [take appropriate actions];  
  raise;
```

The raise statement propagates the exception to the next higher level for further handling after it has been partially handled at the current level. At some level it may be appropriate to raise a more general exception that is more in keeping with the abstractions at that level. The handler at that level might look like this:

```
when Serial_Interface_Malfunction =>  
  [take appropriate actions];  
  raise Line_Failure;
```

This handler would be appropriate once the exception has reached a level

responsible for any failure of a line, whatever the cause, provided that all such failures are considered indistinguishable at that level of abstraction.

Often an intervening level has no actions to perform except to propagate the exception. If there is no handler for Serial_Interface_Malfunction, the exception will be propagated automatically, but it is preferable to propagate it explicitly:

```
when Serial_Interface_Malfunction =>  
    raise;
```

This documents the fact that the program unit expects to propagate the exception in certain circumstances. This makes it much easier for a reader to understand the system's exception propagation paths. The explicit handler also prevents the exception from being handled by an "others" handler that handles the exception inappropriately or fails to propagate it. Even if an "others" handler is not present initially, it is wise to guard against the possibility that it will be added later, during maintenance.

Example Problem Statement (6)

A message switch validates messages as they are received. Messages whose contents do not meet certain validity criteria are rejected, and not processed any further. Accepted messages are validated again, by an independently designed algorithm, just before they are transmitted. This results in a high level of confidence that invalid messages will, in fact, be intercepted, even if one of the two validation subprograms is faulty. The danger that a valid message will be rejected is not as crucial a concern as the danger that an invalid message will be passed along.

The second validation algorithm may conclude that the message is invalid, or it may be unable to examine the message because of an inconsistency in the data structure. Either of these circumstances is indicative of an error in the message switch software. In the first case, the error is in one of the two validation subprograms. In the second case, the cause of the error is harder to pinpoint.

Solution Outline (6)

Robust software is software that will continue to operate, albeit possibly in a degraded mode, despite faults in input data, software, or hardware. Exceptions were included in Ada to foster the writing of robust software. Of course software must also be reliable. There is no point in continuing to operate without confidence that the results of continued operation will be valid. An unreliable system purporting to produce accurate results is usually more dangerous than an inoperative system. In determining the appropriate response to a software error, then, we must distinguish between degraded operation and undependable operation.

When the second validation subprogram in the message switch discovers an error in the internal representation of messages, all messages stored in the system are suspect. There can be no confidence that the results of continued operation will be meaningful. The appropriate response is to shut down as gracefully as possible, saving data in files, leaving files in a consistent state, notifying recipients in the process of communicating with the message switch, notifying the message switch operator, and leaving a trace that will be useful in diagnosing the problem. Each level of the program should perform the cleanup actions for which it is responsible, and notify the next higher level of the software failure.

When the second validation subprogram finds fault with the contents of a message that was validated by the first validation subprogram, the situation is quite different. The software error is isolated, and the only possible consequence is the rejection of a valid message. The appropriate response is to continue operation in a degraded mode by rejecting the message, informing the operator that a potentially valid

message has been rejected, and continuing to process any messages that are accepted by both validation routines.

Exceptions are not the only mechanism by which a subprogram can notify its caller of a software error. Status variables -- out parameters dedicated to telling a caller whether or not the called program completed successfully -- can also serve this purpose. This was the approach taken in the message switch. The detailed solution compares this approach with the use of exceptions. The comparison shows that exceptions allow for a much more lucid presentation of the normal processing.

Detailed Solution (6)

The second validation subprogram takes the form of a function `Header_Valid` that takes parameters of type `Message_Type` and `Physical_Line_Type` and returns a result of type `Boolean`. `Header_Valid` is called in the task `Message_Output_Manager`, which schedules the transmission of parts of messages and handles preemption by higher priority messages. `Message_Output_Manager` is concerned only with the value of the function call, but evaluation of the function also has the side effect of notifying the operator of a message rejection, recording the rejection in an audit file, and removing the message from the system. Within `Message_Output_Manager`, a call on `Header_Valid` appears in contexts like the following:

```
loop
  [set Message to the highest-priority message ready to be sent];
  loop
    if Header_Valid(Message, Physical_Line) then
      [check that no higher-priority message arrived during
       validation];
      if [no higher-priority message arrived] then
        [begin transmission of Message]
      else
        Message := [the higher-priority message];
      end if;
    else
      exit;
    end if;
  end loop;
end loop;
```

At this level of abstraction, an invalid message is treated as a normal event, handled as part of the main algorithm.

Internally, `Header_Valid` checks first the routing indicators (RI's) in the message header, then the line media format (LMF) in the message header, and finally the security level in the message header, in order to determine whether the header is valid. These checks are performed by

the functions Valid_RI_Field, Valid_LMF_Field, and Valid_Security_Level, respectively. As soon as a problem is found, the procedure Reject_Message is called to notify the operator, make an entry in the audit trail, and remove the message from the system. Header_Valid looks like this:

```

function Header_Valid (Message : Message_Type;
                      Physical_Line : Physical_Line_Type)
return Boolean is

type Validity_Type is
  (Positive_Velocity, Bad_RI_Velocity, Bad_LMF_Velocity,
   Bad_Security_Level_Velocity);

Result : Validity_Type;

function Valid_RI_Field
  (Message      : Message_Type;
   Physical_Line : Physical_Line_Type)
return Validity_Type is separate;

function Valid_LMF_Field
  (Message : Message_Type)
return Boolean is separate;

function Valid_Security_Level
  (Message      : Message_Type;
   Physical_Line : Physical_Line_Type)
return Boolean is separate;

procedure Reject_Message
  (Message      : in Message_Type;
   Error_Category : in Validity_Type)
is separate;

begin -- Header_Valid

  Result := Valid_RI_Field (Message, Physical_Line);
  if Result = Positive_Velocity then
    Result := Valid_LMF_Field (Message);
    if Result = Positive_Velocity then
      Result := Valid_Security_Level (Message, Physical_Line);
    end if;
  end if;
  if Result /= Positive_Velocity then
    Reject_Message (Message, Result);
    return False;
  else
    return True;
  end if;
end Header_Valid;

```

To restrict the scope of this discussion, we shall consider only the subunit Valid_RI_Field in greater detail. Each routing indicator has a set of logical lines associated with it, stored in an "RI table." A message's RI field is considered valid with respect to the current physical line if the current physical line is valid for the message's logical line and at least one of the routing indicators in the RI field has the message's logical line associated with it. The subunit Valid_RI_Field has context clauses making the following declarations available:

```

subtype RI_String_Subtype is String (1 .. 7);

type RI_Status_Type is (OK_Status, Last_RI_Status, Error_Status);

type Position_Type is private;

type Logical_Line_List_Cell_Type;

type Logical_Line_List_Type is access Logical_Line_List_Cell_Type;

type Logical_Line_List_Cell_Type is
  record
    Head : Logical_Line_Type;
    Tail : Logical_Line_List_Type;
  end record;

function Message_Logical_Line (Message : Message_Type)
  return Logical_Line_Type;

function Valid_Physical_Line (Logical_Line : Logical_Line_Type;
  Physical_Line : Physical_Line_Type)
  return Boolean;

procedure Find_First_RI (Message : in Message_Type;
  Position : out Position_Type;
  Status : out RI_Status_Type);

procedure Obtain_RI (Message : in Message_Type;
  RI : out RI_String_Subtype;
  Position : in out Position_Type;
  Status : out RI_Status_Type);

procedure Read_RI_Table (RI : in RI_String_Subtype;
  Logical_Lines : out Logical_Line_List_Type;
  Success : out Boolean);

procedure Notify_Operator (Operator_Message : in String);

```

The subunit itself is as follows:

separate (Header_Valid)

```
function Valid_RI_Field (Message      : Message_Type;
                        Physical_Line : in Physical_Line_Type)
return Validity_Type is
```

```
Current_Logical_Line : Logical_Line_Type;
Logical_Line_List    : Logical_Line_List_Type;
Read_RI_Table_Success : Boolean;
RI_Position          : Position_Type;
RI_Status            : RI_Status_Type;
RI                   : RI_String_Subtype;
```

begin -- Valid_RI_Field

```
Current_Logical_Line := Message_Logical_Line (Message);
if Valid_Physical_Line (Current_Logical_Line, Physical_Line) then
  Find_First_RI (Message, RI_Position, RI_Status);
  if RI_Status = Error_Status then
    Notify_Operator ("Software malfunction in RI_Field_Valid: " &
                    "Unable to find first RI.");
    return Bad_RI_Validity;          -->> RETURN <<--
  else
    while RI_Status /= Last_RI_Status loop
      Obtain_RI (Message, RI, RI_Position, RI_Status);
      if RI_Status = Error_Status then
        Notify_Operator ("Software malfunction in " &
                        "RI_Field_Valid: " &
                        "Unable to obtain RI.");
        return Bad_RI_Validity;      -->> RETURN <<--
      end if;
      Read_RI_Table ( RI,
                      Logical_Line_List,
                      Read_RI_Table_Success );
      if Read_RI_Table_Success then
        while Logical_Line_List /= null loop
          if Logical_Line_List.Head = Current_Logical_Line then
            return Positive_Validity; -->> RETURN <<--
          end if;
          Logical_Line_List := Logical_Line_List.Tail;
        end loop;
      end if;
    end loop;
    return Bad_RI_Validity;          -->> RETURN <<--
  end if;
else
  return Bad_RI_Validity;          -->> RETURN <<--
end if;
```

end Valid_RI_Field;

It is apparent that Valid_RI_Field makes no attempt to distinguish between errors in trying to read the data structure and successful attempts to read the data structure that reveal the contents of the RI field to be invalid. The logic of Valid_RI_Field depends on the status values returned in out parameters by subprograms like Find_First_RI, Obtain_RI, and Read_RI_Table. To show how these status values are produced, we shall examine the subprogram Obtain_RI. The status values in the other subprograms are produced in a similar manner. Obtain_RI is a primitive operation used by many parts of the message switch. It occurs in a context in which the following declarations, among others, are available:

```
type Message_Part_Status_Type is
  (OK_Status, End_Of_Text_Status, Link_Error_Status,
   Other_Error_Status);
```

```
type Format_Type is (ACP_Format, JANAP_Format);
```

```
procedure Read_Char_From_Message_Part
  (Message : in Message_Type;
   Position : in Position_Type;
   Result : out Character;
   Status : out Message_Part_Status_Type);
```

```
function Position_Type_Pred
  (Position : Position_Type)
  return Position_Type;
```

In addition, Obtain_RI makes use of the fact that a Message_Type value points to a record with a component named Format_Part, of type Format_Type. Here is the body of Obtain_RI:

```
procedure Obtain_RI (Message : in Message_Type;
                    RI : out RI_String_Subtype;
                    Position : in out Position_Type;
                    RI_Status : out RI_Status_Type) is

  Current_Char, Previous_Char : Character;
  Char_Count : Natural := 1;
  Part_Status : Message_Part_Status_Type;
```

```
begin -- Obtain_RI
```

```
  RI := (RI_String_Subtype'Range => ' ');
  Read_Char_From_Message_Part
    (Message, Position, Current_Char, Part_Status);
  if Part_Status /= OK_Status then
    RI_Status := Error_Status;
    return;
  end if;
  -->> RETURN <<--
```

```

for I in RI_String_Subtype'Range loop
  exit when Current_Char not in 'A' .. 'Z'; -->> LOOP EXIT <<--
  RI(I) := Current_Char;
  Read_Char_From_Message_Part
    (Message, Position, Current_Char, Part_Status);
  if Part_Status /= OK_Status then
    RI_Status := Error_Status;
    return; -->> RETURN <<--
  end if;
end loop;

-- Search for start of next RI:
Previous_Char := ' ';
loop
  case Current_Char is
    when 'R' | 'U' | 'Y' =>
      exit; -->> LOOP EXIT <<--

    when '.' =>
      if Message.Format_Part = JANAP_Format then
        RI_Status := Last_RI_Status;
        return; -->> RETURN <<--
      end if;

    when 'D' | 'Z' =>
      if Message.Format_Part = ACP_Format and
        Previous_Char = ASCII.LF then
        RI_Status := Last_RI_Status;
        return; -->> RETURN <<--
      end if;

    when others =>
      null;
  end case;
  Previous_Char := Current_Char;
  Read_Char_From_Message_Part
    (Message, Position, Current_Char, Part_Status);
  if Part_Status /= OK_Status then
    RI_Status := Error_Status;
    return; -->> RETURN <<--
  end if;
end loop;

-- We are now pointing to the second character in the RI.
-- We must back up to point to the first.
Position := Position_Type_Pred (Position);

RI_Status := OK_Status;

```

```
exception -- Obtain_RI
```

```
  when others =>
```

```
    RI_Status := Error_Status; -- Make sure all problems are
                                -- reported.
```

```
end Obtain_RI;
```

Because of the "others" handler, any exception arising during execution of Obtain_RI has the same result as a bad status code being returned by Read_Char_From_Message_Part -- Obtain_RI returns its own bad status code. At first glance, the "others" handler may appear to be a blessing to program reliability. Unfortunately, it causes all errors occurring for unknown reasons to be treated in exactly the same way as errors which can be pinpointed to an inability on the part of Read_Char_From_Message_Part to find a character given its parameters. More seriously, tracing the treatment of the bad status code up through Valid_RI_Field, Header_Valid, and Message_Output_Manager, we see that the ultimate treatment of such errors is to reject the message during whose validation they occurred and continue as if nothing happened. Even though these errors are symptoms of a mysterious and possibly serious problem in the logic of the program, they are treated as if their cause is confined to one of the redundant validation subprograms.

A more careful design would distinguish between errors known to be confined to the logic of the second validation and errors occurring for unknown reasons. This distinction can be clarified by using exceptions to communicate the discovery of unanticipated problems and status codes to communicate "routine" anomalies. In addition, program logic will be simplified by the use of exceptions, which are propagated automatically, in place of explicit statements to check status codes returned by subprograms, set higher level status codes, and return.

This solution will be presented bottom-up. First, we shall assume that Read_Char_From_Message_Part is modified so that it may raise any of three exceptions. The exception Unanticipated_Error is raised when Read_Char_From_Message_Part encounters an exception for unknown reasons. The exception Message_Part_Link_Error is raised in any other situation in which Read_Char_From_Message_Part would have set its Status parameter to Link_Error_Status. The exception Other_Message_Part_Error is raised in any other situation in which Read_Char_From_Message_Part would have set its Status parameter to Other_Error_Status. This leaves the Status parameter with two possible values upon normal completion of a call -- OK_Status and End_Of_Text_Status. Thus we can replace the Status parameter by a single Boolean parameter named At_End_Of_Text. Obtain_RI can then be recoded as follows:

```

procedure Obtain_RI (Message      : in Message_Type;
                    RI           : out RI_String_Subtype;
                    Position     : in out Position_Type;
                    At_Last_RI  : out Boolean) is

    Current_Char, Previous_Char : Character;
    Char_Count                  : Natural := 1;
    At_End_Of_Text              : Boolean;
    End_Of_Text_Error           : exception;

begin -- Obtain_RI

    RI := (RI_String_Subtype'Range => ' ');
    Read_Char_From_Message_Part
        (Message, Position, Current_Char, At_End_Of_Text);
    if At_End_Of_Text then
        raise End_Of_Text_Error;
    end if;
    for I in RI_String_Subtype'Range loop
        exit when Current_Char not in 'A' .. 'Z'; -->> LOOP EXIT <<--
        RI(I) := Current_Char;
        Read_Char_From_Message_Part
            (Message, Position, Current_Char, At_End_Of_Text);
        if At_End_Of_Text then
            raise End_Of_Text_Error;
        end if;
    end loop;

    -- Search for start of next RI:
    Previous_Char := ' ';
    loop
        case Current_Char is
            when 'R' | 'U' | 'Y' =>
                At_Last_RI := False;
                exit; -->> LOOP EXIT <<--
            when '.' =>
                if Message.Format_Part = JANAP_Format then
                    At_Last_RI := True;
                    return; -->> RETURN <<--
                end if;
            when 'D' | 'Z' =>
                if Message.Format_Part = ACP_Format and
                    Previous_Char = ASCII.LF then
                    At_Last_RI := True;
                    return; -->> RETURN <<--
                end if;
            when others =>
                null;
        end case;
    end loop;
end Obtain_RI;

```

```

    Previous_Char := Current_Char;
    Read_Char_From_Message_Part
      (Message, Position, Current_Char, At_End_Of_Text);
    if At_End_Of_Text then
      raise End_Of_Text_Error;
    end if;
  end loop;

  -- We are now pointing to the second character in the RI.
  -- We must back up to point to the first.
  Position := Position_Type_Pred (Position);

exception -- Obtain_RI

  when End_Of_Text_Error =>
    Notify_Operator ("Unanticipated end of text in Obtain_RI.");
    raise Unanticipated_Error;

  when Message_Part_Link_Error | Other_Message_Part_Error =>
    Notify_Operator ("Unsuccessful call on " &
      "Read_Char_From_Message_Part by Obtain_RI.");
    raise Unanticipated_Error;

  when Unanticipated_Error =>
    Notify_Operator ("Unanticipated error propagated " &
      "to Obtain_RI.");
    raise;

  when others =>
    Notify_Operator
      ("Unanticipated error encountered in Obtain_RI.");
    raise Unanticipated_Error;

end Obtain_RI;

```

Though calls on Read_Char_From_Message_Part from other parts of the system may legitimately encounter the end of text, it is an error if the end of text is encountered in Obtain_RI. Therefore, the value of At_End_Of_Text is checked after each call on Read_Char_From_Message_Part. If At_End_Of_Text is true, the locally declared exception End_Of_Text_Error is raised. Any exception raised in Obtain_RI, including End_Of_Text_Error, any of the errors raised by Read_Char_From_Message_Part, and any predefined exception, results in a call on Notify_Operator followed by the raising (or re-raising) of Unanticipated_Error. This reflects the fact that Obtain_RI does not expect to encounter any exception, so any continued processing is unsafe once an error has been found. Because Obtain_RI now raises an exception in every case in which it would otherwise have set Status to Error_Status, Error_Status can now only have one of two values upon normal completion of Obtain_RI -- OK_Status or Last_RI_Status. Thus we

replace the RI_Status_Type parameter Status by the Boolean parameter At_Last_RI.

Similar changes can be applied to the procedure Find_First_RI. As originally written, this procedure never sets its Status parameter to Last_RI_Status -- only to OK_Status or Error_Status. After Find_First_RI is modified to raise Unanticipated_Error instead of setting Status to Error_Status, this parameter can be omitted entirely. The function Valid_RI_Field can now be rewritten as follows:

```
separate ( Header_Valid )
```

```
function Valid_RI_Field (Message      : Message_Type;
                        Physical_Line : in Physical_Line_Type)
return Validity_Type is
```

```
Current_Logical_Line : Logical_Line_Type;
Logical_Line_List    : Logical_Line_List_Type;
Read_RI_Table_Success : Boolean;
RI_Position           : Position_Type;
At_Last_RI           : Boolean;
RI                   : RI_String_Subtype;
```

```
begin -- Valid_RI_Field
```

```
Current_Logical_Line := Message_Logical_Line (Message);
if Valid_Physical_Line (Current_Logical_Line, Physical_Line) then
  Find_First_RI (Message, RI_Position);
  loop
    Obtain_RI (Message, RI, RI_Position, At_Last_RI);
    exit when At_Last_RI; -->> LOOP EXIT <<--
    Read_RI_Table ( RI,
                    Logical_Line_List,
                    Read_RI_Table_Success );
    if Read_RI_Table_Success then
      while Logical_Line_List /= null loop
        if Logical_Line_List.Head = Current_Logical_Line then
          return Positive_Velocity; -->> RETURN <<--
        end if;
        Logical_Line_List := Logical_Line_List.Tail;
      end loop;
    end if;
  end loop;
  return Bad_RI_Velocity; -->> RETURN <<--
else
  return Bad_RI_Velocity; -->> RETURN <<--
end if;
```

```
exception -- Valid_RI_Field
```

```
  when Unanticipated_Error =>
```

```
    Notify_Operator ("Unanticipated error propagated to " &  
                     "Valid_RI_Field.");
```

```
    raise;
```

```
  when others =>
```

```
    Notify_Operator
```

```
      ("Unanticipated error encountered in Valid_RI_Field.");
```

```
    raise Unanticipated_Error;
```

```
end Valid_RI_Field;
```

In this version of Valid_RI_Field, the "expected" and unexpected anomalies are clearly distinguished. The former cause Bad_RI_Validity to be returned using the normal flow of control; the latter cause Unanticipated_Error to be raised or re-raised.

An alternative approach is to rewrite Header_Valid and its three function subunits -- Valid_RI_Field, Valid_LMF, and Valid_Security_Level -- as procedures -- Validate_Header, Validate_RI_Field, Validate_LMF, and Validate_Security_Level. Validate_Header would have the same two in parameters plus a single out parameter, Header_Valid, which it would set to True or False. The advantage of this approach is that the statements

```
loop  
  [set Message to the highest-priority message ready to be sent];  
loop  
  if Header_Valid(Message, Physical_Line) then  
    [check that no higher-priority message arrived during  
     validation];  
    if [no higher-priority message arrived] then  
      [begin transmission of Message]  
    else  
      Message := [the higher-priority message];  
    end if;  
  else  
    exit;  
  end if;  
end loop;  
end loop;
```

in Message_Output_Manager would be rewritten as follows:


```

loop
  [set Message to the highest-priority message ready to be sent];
loop
  Validate_Header (Message, Physical_Line, Header_Valid);
  if Header_Valid then
    [check that no higher-priority message arrived during
     validation];
    if [no higher-priority message arrived] then
      [begin transmission of Message]
    else
      Message := [the higher-priority message];
    end if;
  else
    exit;
  end if;
end loop;
end loop;

```

This is preferable for two reasons. First, the validation process has side effects when a message is rejected, so a procedure is more appropriate than a function. Second, the logic of the code is based on the premise that the validation process takes time, and a higher-priority message may arrive during that time. This is easier to comprehend if the program reader can point to a specific simple statement responsible for using up this time. We tend to regard the selection of one arm of an if statement as virtually instantaneous, despite the fact that evaluation of the condition in the if statement may involve a time-consuming function call.

Validate_Header can be written so that the subunits Validate_RI_Field, Validate_LMF, and Validate_Security_Level simply return without changing anything when validation succeeds, but raise unique exceptions when validation fails. These exceptions are declared and handled within Validate_Header. The resulting code is extremely straightforward:

```

procedure Header_Valid (Message      : in Message_Type;
                        Physical_Line : in Physical_Line_Type;
                        Header_Valid  : out Boolean) is

type Validity_Type is
    (Bad_RI_Validity, Bad_LMF_Validity,
     Bad_Security_Level_Validity);

RI_Field_Error, LMF_Error, Security_Level_Error : exception;

procedure Validate_RI_Field
    (Message      : in Message_Type;
     Physical_Line : in Physical_Line_Type)
    is separate;

procedure Validate_LMF_Field
    (Message : in Message_Type)
    is separate;

procedure Validate_Security_Level
    (Message      : in Message_Type;
     Physical_Line : in Physical_Line_Type)
    is separate;

procedure Reject_Message
    (Message      : in Message_Type;
     Error_Category : in Validity_Type)
    is separate;

begin -- Validate_Header

    Validate_RI_Field (Message, Physical_Line);
    Validate_LMF_Field (Message);
    Validate_Security_Level (Message, Physical_Line);

exception --- Validate_Header

    when RI_Field_Error =>
        Reject_Message (Message, Bad_RI_Validity);
        Header_Valid := False;

    when LMF_Error =>
        Reject_Message (Message, Bad_LMF_Validity);
        Header_Valid := False;

    when Security_Level_Error =>
        Reject_Message (Message, Bad_Security_Level_Validity);
        Header_Valid := False;

```

```

when Unanticipated_Exception =>
    Notify_Operator
        ("Unanticipated error propagated to Validate_Header.");
    raise;

when others =>
    Notify_Operator
        ("Unanticipated error encountered in Validate_Header.");
    raise Unanticipated_Exception;

end Validate_Header;

```

(The procedure `Validate_RI_Field` can be obtained from the function `Valid_RI_Field` simply by changing the statements "return `Positive_Validity`;" and "return `Bad_RI_Validity`;" to "return;" and "raise `RI_Field_Error`;", respectively.)

`Validate_Header` illustrates the distinction between two different kinds of exceptions, system-wide exceptions and local exceptions. `Unanticipated_Error` is a system-wide exception. It may be raised directly or indirectly by `Find_First_RI` and `Obtain_RI`, subprograms that are called from `Validate_Header` but also from other parts of the system. `Unanticipated_Error` is propagated to any subprogram calling `Validate_Header`. The raising, handling, and propagation of `Unanticipated_Error` are specified as a part of system design. `RI_Field_Error`, `LMF_Error`, and `Security_Level_Error` are local exceptions. They are declared in `Validate_Header`, raised in subunits of `Validate_Header`, and always handled within `Validate_Header`. These exceptions are of no concern to any other part of the system. The use of these exceptions is specified as part of the implementation of `Validate_Header`.

It is clear that the use of exceptions can greatly simplify the specification of the normal flow of control. This is particularly evident in `Validate_RI_Field` and `Validate_Header`. In `Validate_Header`, the use of exceptions allowed a complex structure of nested if statements to be replaced by a sequence of three simple statements, without changing the underlying flow of control. Of course the increased clarity in the presentation of the normal processing algorithm comes at the expense of a more subtle presentation of the processing of abnormal conditions. If these conditions are truly abnormal, however, this may be appropriate.

3. EPILOGUE

C. A. R. Hoare, who feels that exception handling facilities do not belong in a programming language, argues as follows [Hoa81]:

... The danger of exception handling is that an "exception" is too often a symptom of some entirely unrelated problem. For example, a floating point overflow may be the result of an incorrect pointer use some forty-three seconds before; and that was due perhaps to programmer oversight, transient hardware fault, or even a subtle compiler bug.... The right solution is to treat all exceptions in the same way as symptoms of disaster; and switch the entire operating regime to one designed to survive arbitrary failure of the entire computer running the program which generated the exception.

... A more subtle danger is that the programmer is encouraged to postpone the vitally important considerations of safety, in the hope that he will be able to patch up the problem by exception handlers written afterward. Experience with PL/I and MESA shows that this hope is too often disappointed; and many users of these languages have abandoned use of exceptions.

Hoare's concerns are quite valid, though his conclusion may be extreme. The danger of an exception being attributed to the wrong cause is magnified by the generality of Ada's predefined exceptions. For example, the exception `Constraint_Error` may be raised by an assignment statement in which the value assigned does not lie within the subtype of the target variable; by a subprogram call in which there is a mismatch between the value of an in or in out actual parameter and the subtype of the corresponding formal parameter, or between the value of an in out or out formal parameter and the subtype of the corresponding actual parameter, or between the value in a function return statement and the result subtype of the function; by a subscript out of bounds; by a reference to a record component inconsistent with the current value of the record's discriminants; or by an attempt to use the object designated by a null access value.

In a part of the message switch program implementing a data abstraction for variable length character strings, a substring extraction subprogram checks that the specified substring lies within the current logical bounds (as opposed to the permanent physical bounds) of the source string. If not, the subprogram raises `Constraint_Error`, the same error raised when a slice of an object of the predefined type `String` has illegal bounds. The analogy is accurate, but this is precisely the worst thing to do.

Rather than making the predefined error categories even broader, designers should strive to make exceptions as narrow and precise as possible. Predefined exceptions that are presumed to arise from a known cause should be handled in small block statements, even if the handlers simply raise more specific programmer-defined (or, more appropriately, designer-defined) exceptions. The smaller the block statement and the narrower the category of situations encompassed by the programmer-defined exception, the less the danger that a predefined exception arising from some other cause will be mistaken for one arising from the presumed cause.

Hoare's second concern, that "the programmer is encouraged to postpone the vitally important considerations of safety," reflects confusion between the roles of the system designer and the programmer. The externally visible exceptions raised by various modules of a system, and the ways in which they are propagated, handled, and re-raised, are an essential part of the system design. (This is especially evident, for example, in the design of a message switch's response to the failure of one of its serial interfaces.) The designer has not completed his assignment until he has specified how system-wide exceptions are to be used.

A description of the exceptions raised by a given program unit, and the circumstances in which they are raised, is as much a part of the program unit's interface as a list of parameter names and types. A design must specify all the connections between interfaces of program components. The purpose of a handler of the form

```
when [some exception] =>  
    raise;
```

is to specify this vital part of a program unit's interface when the design calls for that unit to propagate the exception without doing anything else.

A corollary of the thesis that exceptions form part of a module's interface is that a programmer should never take it upon himself to propagate an exception not specified in the design, any more than he would take it upon himself to add a parameter to one of the subprograms declared in a package specification. Neither should a programmer take it upon himself to "absorb" a system-wide exception -- that is, to handle the exception without re-raising it -- when the design does not explicitly call for this to be done. A programmer is free to declare any exceptions he finds useful in the implementation of a module -- provided that such exceptions are handled within the module and invisible to the outside world.

The writer of a general, reusable software component like `Stack_Package` defines the interfaces of that component for himself. It is incumbent on the designer of a system using that component to conform to the component's interface. Assuming that the component is a package,

the exceptions raised by the package's visible subprograms should be declared in the package declaration — indicating explicitly that they are part of the package's interface. The circumstances under which each exception is raised, and the consequences of continuing to use the package once an exception has been raised, must be carefully documented.

It may be that Hoare underestimates the extent to which careful design and programming make it possible to anticipate and determine the true cause of exceptions. Nonetheless, when a design attempts to anticipate all sources of exceptions, any unanticipated exceptions encountered during program execution ought indeed to be treated as symptoms of disaster. In some cases, such as when redundant software components give inconsistent results, it may be possible to ascertain that the potential effects of the malfunction are limited and to continue degraded operation. Another possible course of action when the cause and effects of the malfunction can be isolated to one module is to invoke an independently developed version of that module. (This approach to fault-tolerant software is sometimes known as multiple-version programming.) When no such damage-containment strategy is applicable, the only safe course of action is to terminate execution, performing appropriate cleanup actions at each level and providing a diagnostic error walkback. A global exception like Unanticipated_Error is a natural mechanism for accomplishing this graceful termination.

This case study has only scratched the surface of the problem of planning for exceptions in system design. Detailed Solutions (5) and (6) illustrated some issues that must be considered and approaches that may be taken, but they should not be read as universal solutions. Handling hardware and software failures remains one of the most difficult problems in the design of real-time systems. There is much research still required in the area of fault-tolerant software design.

The use of exceptions to implement a control structure or to handle normal processing events should be extremely localized. Thus it is a coding issue rather than a design issue. Important criteria in deciding to take such an approach are the availability of more straightforward approaches and the likelihood that a program reader will readily recognize the pattern of control flow that the programmer is trying to express. The very word "exception" tends to suggest to the reader that actions specified in handlers are performed only in unusual circumstances.

2.5 Program Structure

This section contains the following case studies:

Specifying Interfaces for General Purpose, Portable Software:

A Study of Ada Input/Output

Information Hiding

Reducing Depth of Nesting

Library Units Versus Subunits

THIS PAGE INTENTIONALLY LEFT BLANK

AD-A140 818

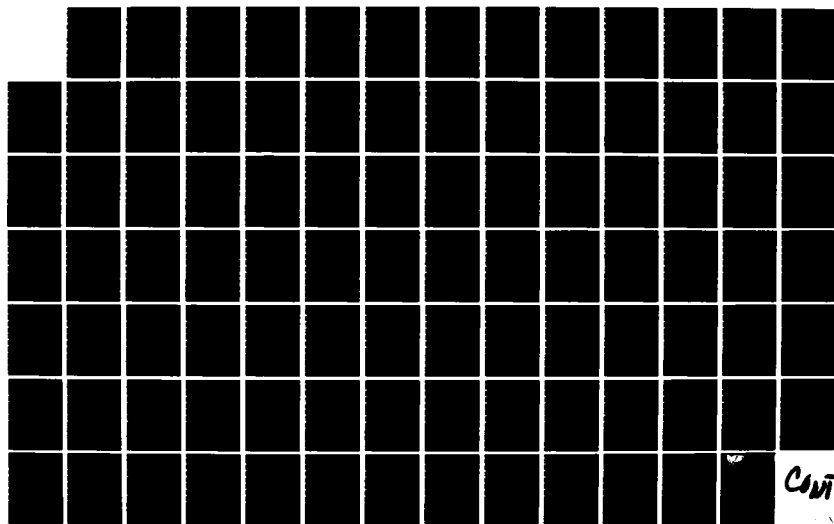
ADA (TRADEMARK) CASE STUDIES II(U) SOFTECH INC WALTHAM
MA JAN 84 DAB007-83-C-K514

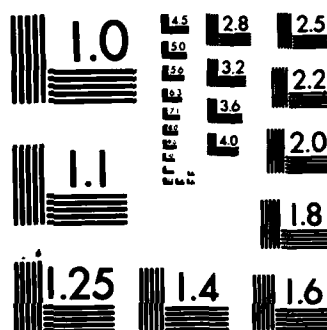
3/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

2.5.1 SPECIFYING INTERFACES FOR GENERAL PURPOSE, PORTABLE SOFTWARE: A STUDY OF ADA INPUT/OUTPUT

1. BACKGROUND

Case Study Objective

To reveal potential problems that should be kept in mind when specifying software interfaces that are intended to be portable and provide a wide range of functionality.

Problem

Relatively little software is actually designed and tested on a wide variety of hardware architectures. Although lip service is often paid to producing and implementing portable designs, in practice, it is hard to avoid hidden (and even conscious) dependencies on the system that is first used to support the software. Portability and generality are often compromised in favor of meeting a schedule or satisfying efficiency requirements.

The Ada input/output packages as specified in the Ada Reference Manual (RM) [DOD83] are the only widely circulated examples of Ada specifications intended to be implemented on a broad range of hardware architectures and operating systems. Despite differences in hardware and operating system support, the packages are intended to provide a standard, implementation independent functionality. Since the Ada I/O packages are also to be used directly by programmers, they also illustrate some of the problems of specifying general purpose capabilities. Thus, an examination of Ada's input/output capabilities will show some of the problems that can arise when attempting to specify general purpose, portable Ada packages.

Discussion

The following difficulties in designing a general purpose portable package are evident in the Ada Text_IO package specification:

- indirect limitations on implementation freedom and efficiency due to interactions between functions (specifically, restrictions on how the Text_IO.File_Type can be implemented);
- when and how to give in parameters default values (specifically, giving the Base argument of Integer_IO.Put a default value of 10);

- making implicit assumptions about underlying hardware/software support that are not valid over the anticipated range of systems (specifically, assuming that files can be dynamically created and deleted);
- providing information useful for recovering from exceptional situations (e.g., encountering the end of volume when writing a tape, or finding that a mounted tape requires initialization);
- avoiding compiler dependencies in a specification (specifically, supporting variable length I/O in the Sequential_IO and Direct_IO packages by requiring support for instantiations with unconstrained array and record types).

2. DETAILED EXAMPLES

Example Problem Statement (1)

Preserving Implementation Freedom

Ada Text_IO files are declared as limited private types. At first, such a declaration may seem to ensure complete freedom in how to implement the type, but this is not the case for Text_IO. Interactions among the functional requirements imply that File_Type must be implemented, in essence, as an access type. In particular, the definitions of Set_Input and Set_Output implicitly impose this requirement. Consider the following example:

```
with Text_IO; use Text_IO;
procedure Example_1 is
  File_1 : File_Type;
begin
  Create (File_1);
  Set_Output (File_1);
  Put_Line ("Line 1");
  Put_Line (File_1, "Line 2");
  if Line (File_1) /= Line (Current_Output) then
    Put_Line (Standard_Output, "Failed");
  end if;
end Example_1;
```

Note that the line position count is a property of the internal file, File_1. If File_Type were implemented as a record, e.g.:

```
type File_Type is
  record
    Line : Count := 1;
    ...
  end record;
```

then Set_Output would only be able to copy the File_Type record, and the line counts would no longer be coordinated, as is shown by the following outline of an implementation:

```
package body Text_IO is
  Current_Output_File : File_Type;

  -- The implementations given below do not satisfy all require-
  -- ments of the RM; they are only intended to illustrate how the
  -- functional requirements for Set_Output potentially affect the
  -- representation of File_Type.
```

```

procedure Set_Output (File : in File_Type) is
begin
    Current_Output_File := File;
end;

function Line (File : in File_Type) return Positive_Count
is
begin
    return File.Line;
end;

function Line return Positive_Count is
begin
    return Current_Output_File.Line;
end;

function Put (File : File_Type; Item: String) is
begin
    ...
    File.Line := File.Line + 1;
    ...
end Put;

function Put (Item : String) is
begin
    ...
    Current_Output_File.Line := Current_Output_File.Line
                                + 1;
    ...
end Put;
end Text_IO;

```

The above is an incorrect implementation of File_Type because separate copies of the line count will be kept after Set_Output is called. The most straightforward way of meeting all the requirements is to implement File_Type as an access type, although ingenious programmers can find various somewhat inefficient ways of simulating the effect of access types if necessary. The point, however, is that implementation freedom is not guaranteed just because a type is declared as a limited private type.

Example Problem Statement (2)

Providing Parameters with Default Values

One small interface design decision is when to provide in parameters with default values, e.g., what reasons should influence a choice between:

```
procedure P (A : Integer := 0);
```

and

```
procedure P (A : Integer);  
procedure P;
```

From a user viewpoint, the alternatives appear identical: the user can call P with or without an argument.

The choice between these design alternatives depends on the nature of the parameter having the default value. For example, consider the Put procedure for integer types:

```
Put (Item : in Num;  
     Width : in Field := Default_Width;  
     Base : in Number_Base := Default_Base);
```

Put (8, 0, 10) outputs "8", and so does Put (8, 0). However, if Base is any value other than 10, the value is output in based literal notation, e.g., Put (8, 0, 16) yields "16#8#". Because the value 10 determines both the base and the form of the output, it is not possible to output an integer as a based literal with base 10, i.e., it is not possible to obtain 10#8#. Now this is not a significant loss of functionality, but if Put had been declared as:

```
Put (Item : in Num;  
     Width : in Field := Default_Width;  
     Base : in Number_Base);           -- output based lit.  
Put (Item : in Num;  
     Width : in Field := Default_Width); -- output decimal lit.
```

then based literal output would always be produced when a base was given explicitly. The form of the call would determine the form of the output -- based literal or decimal literal.

(In defense of the current design choice, note that when outputting several integer values all in hexadecimal notation, one can first set Default_Base to 16, and then simply call Put with a single argument; with the alternative described above, the base would have to be specified explicitly in each call to Put.)

Although a decision about providing default values is not a major design issue, this example does show that whether default values are provided can affect the functionality available to a user.

Example Problem Statement (3)

Assumptions about Underlying Hardware/Software Support

Dealing with differences in underlying hardware/software support is one of the most important problems to be solved when developing a general purpose specification. In [Kaf82], it is suggested that each specification explicitly state what demands it makes on underlying software and hardware. The importance of documenting such requirements explicitly is well illustrated by the Text_IO package, which does not provide such documentation. It is consequently not easy to be sure that the required capabilities can be supported on an acceptably wide range of systems (although this does seem likely).

There are at least two implementation dependent assumptions that may cause difficulty in implementing Text_IO:

- files can be dynamically created and deleted;
- lines have an unbounded length by default.

In addition, because the circumstances under which Use_Error can be raised are broadly defined, an implementation is, in effect, allowed to provide implementation defined functional subsets of Text_IO functionality. This approach for allowing functional subsets causes some difficulty in validating the Text_IO package.

Dynamic File Creation/Deletion

In some operating systems, all files to be used by a program must be declared to the operating system before invoking the program; in such systems, file deletion only occurs after the main program returns, and is under the control of commands given directly to the operating system. The Ada input/output packages assume that files can be created and deleted dynamically:

```
Open (F, Out_File, "Test");  
Delete (F); -- (1)  
Open (F, Out_File, "Test"); . -- (2)
```

If Use_Error is not raised at (1), the Open at (2) must raise Name_Error, since the "Test" file no longer exists (see RM 14.3.1, paragraph 13). If an operating system does not support dynamic file deletion, an implementation will have to raise Use_Error at (1). A casual reader of the Text_IO specification could easily be surprised if Use_Error is raised at (1) when the underlying operating system does allow files to be deleted, non-dynamically, at the end of a job (see RM 14.2.1, paragraph 13).

Bounded Line Lengths

The RM specifies that after opening a text file for output, the line length is implicitly set to the value Unbounded (RM 14.3.1). This specification cannot seemingly be satisfied if the underlying operating system actually requires all text files to have fixed lengths. Although it would be consistent with the specification to raise Use_Error when opening a file that is required to have bounded line lengths, this would not be very friendly. It would also be consistent with the RM to raise Use_Error only when a call on Put attempts to exceed the line length limit imposed by the external file. The problem here stems from the desire to specify a standard default value for Line_Length after opening a text file, and an assumption that an unbounded length poses the minimal demand on operating systems. Fortunately, even when this "minimal" assumption is not satisfied, useful functionality can be obtained by arranging to raise Use_Error at the last possible moment.

Functional Subsets

One problem in designing packages to be implemented on a wide range of architectures is coping with architectural limitations that do not allow the desired full functionality to be implemented. One approach to dealing with this problem is to explicitly specify functional subsets to be supported by an implementation. This approach was not taken for Ada I/O. Instead, the Use_Error exception was defined and can be raised whenever some function required by the RM is not supported by an implementation. This approach leads to implementation defined functional subsets.

The Use_Error approach was taken in part because there was not enough time to define functional subsets within the Standard, and in part, because it was not agreed that such functional subsets should be defined. The effect of the Use_Error approach is to defer the subsetting of I/O capabilities to the validation stage rather than the standardization stage. In writing I/O validation tests, for example, it has been assumed that if an implementation accepts a call to Open or Create (i.e., does not raise Use_Error), then it must also accept calls that read and write data, and a call that closes the file. In essence, the tests are written on the assumption that once an implementation undertakes to support any Text_IO functions, it must support all of them, e.g., the tests will invalidate a compiler that raises Use_Error for all calls to instantiations of Float_IO but does not raise Use_Error when attempting to Open a text file.

The Use_Error approach to dealing with implementation dependent limitations has certainly been helpful in speeding the development of a standard, but poses obvious dangers in terms of having a standard that supports portable I/O.

Example Problem Statement (4)

Error Recovery and Exceptions

Every system of general purpose subprograms will encounter exceptional situations in which the proper way to proceed depends on the user. Ada provides exception conditions to signal the occurrence of such situations, but exception conditions must be used in conjunction with other techniques to provide sufficient flexibility to a user. There are two major limitations of Ada's exception signaling mechanism that affect system design:

- exceptions are not parameterized, so there is no standard way to provide additional information about the nature of the situation giving rise to the exception;
- after raising an exception, execution cannot be restarted at the point of the exception situation.

The Ada I/O packages demonstrate the importance of these limitations. For example, when writing a tape, a system will eventually encounter the end of the reel. The RM allows an implementation to raise `Use_Error` in this case, but `Use_Error` can also be raised for other situations as well, e.g., a non-recoverable error in attempting to write due to a defect in the tape. How will the user know which is the actual reason for raising `Use_Error`? For the end of volume situation, an implementation could also decide simply to output a message to the user (assuming interactive usage) asking whether to mount another reel, and if so, what reel to mount. If the user provides another reel, then `Use_Error` need never actually be raised. However, the action of querying the user might not always be appropriate. With the current I/O design, there is no standard way of intercepting or aborting the standard system action for this situation.

There are a variety of techniques for providing more flexibility in handling exceptional situations. For example, Multics has a standard set of system routines for providing more information about any exceptional situation and a standard set of practices for how exception situations should be handled by general purpose subprograms. Such conventions need to be established on a system wide basis, and can provide important flexibility in dealing with implementation dependent limitations. The approach taken by the Ada I/O packages is clearly too inflexible for the KAPSE and for APSE tools, but possible approaches for providing more flexibility within Ada are beyond the scope of this case study.

Compiler-dependent Specifications

The Ada `Sequential_IO` and `Direct_IO` packages provide

compiler-dependent functionality because the element type for a sequential or direct file is declared as a formal private type. This would seem to allow for a wide range of legal types for an instantiation, but if either of these packages is instantiated with an unconstrained array type or an unconstrained type with discriminants, an implementation is allowed to reject the instantiation. The reason is that such instantiations are illegal if the generic type is used in the package body in a context where a constraint would be required (RM 12.3.2, paragraph 4). Since the use of the formal type within the package body is implementation dependent, the legality of the instantiation is implementation dependent. The implementation dependency can only be removed by adding an additional requirement: "Instantiations of Sequential_IO and Direct_IO shall be legal even if these packages are instantiated with an unconstrained array type or an unconstrained type with discriminants." In short, one must understand some of Ada's subtle restrictions in order to write truly general purpose specifications.

THIS PAGE INTENTIONALLY LEFT BLANK

2.5.2 INFORMATION HIDING

1. BACKGROUND

Case Study Objective

To illustrate how Ada packages allow the programmer to implement data abstraction.

Problem

In building a system, every designer faces the issue of data abstraction. How much information about a particular data structure should be hidden and how much should be publicly available for the user? There are tradeoffs in the generality and flexibility of an implementation, in the level of abstraction actually implemented, in the maintainability of the interface and the readability of the code. In this particular case study, the problem centers on designing an interface for a hardware clock.

Discussion

Because the issue of information hiding and data abstraction is large and complex, this discussion will just summarize the major points. There are several criteria to be considered when deciding what information to hide about a data structure:

- what aspects are likely to change
- the separation of the physical from the abstract structure
- the ability to select different implementation alternatives
- the level of the interfaces to be provided for the user
- the ability to define an easy-to-use, logical interface with other system components so that their design may proceed independently.

As an example of information hiding, consider a hardware component such as a clock. Specifying the logical interfaces between the software and hardware components allows the designers to hide the physical hardware structure and interfaces.

2. DETAILED EXAMPLE

Example Problem Statement

A message switch contains some time-dependent processes for which it is necessary to provide a time delta of at most 10 milliseconds. Ordinarily, the predefined Ada type Duration would be used; however, some implementations of the language may not have a fine enough delta -- the language only requires that the smallest value to be represented be no greater than 20 milliseconds.

These time-dependent processes require a precise hardware clock. Within one character transmission interval from the time the serial interface receives a character this interface must send an interrupt to acknowledge receipt of the character. The message switch must monitor its transmissions and acknowledgements so that it can time-out a transmission that hangs and take corrective action for this particular physical line. Because of the 10 millisecond delta requirement, the designers decided to build a hardware clock component and provide a software interface to this clock. The problem lies in defining this interface.

Solution Outline

The designers chose to simulate the clock through a well-defined software interface. A type is declared to represent the ticks of this clock, and operations are defined on it to

- initialize the clock
- read the clock
- convert the ticks to seconds
- calculate the elapsed time in ticks between two clock readings

This type and its operations are encapsulated in a package. Moreover, the type is declared to be a private type so that information concerning its actual implementation is hidden from users. The only operations which users of this package will be allowed to apply to this type are assignment, equality, inequality, membership tests, and the four operations listed above. This package provides an abstraction of the hardware clock, separating the concerns of the physical implementation from the logical entity.

Detailed Solution

The detailed solution to this problem is the package specification outlined above. For completeness, the package specification describing the interface to the Intel 8254 hardware counter / timer chip is also given. The private part of the hardware clock package and the package body are provided for reference in the ensuing discussion. This package imports another package called `Byte_Definitions`, which is not shown below. The package `Byte_Definitions` contains declarations for bytes, words, and long words in terms of 8, 16, and 32 bits respectively. This package also includes subprograms to convert between these different types.

```

with Interface_to_8254; use Interface_to_8254;
with Byte_Definitions; use Byte_Definitions;
package Hardware_Clock_Package is
--
-- This package implements a hardware clock using a binary
-- counter. It is implemented when one needs a time Delta
-- which is less than Duration'Delta.
--          !!!!!!!!!!!!! NOTE !!!!!!!!!!!!!
-- This clock is useful for elapsed time only; it is not an
-- absolute (i.e. time of day) clock. It counts time in ticks.
-- Depending on the physical clock used, there will be a dif-
-- ferent conversion factor from ticks to seconds.
--
type Hardware_Clock_Time_Type is private;

-- Subprogram to start the clock chip running in the proper
-- configuration
procedure Initialize_Hardware_Clock;

-- Subprogram to convert from ticks to seconds
function Seconds
    (Ticks : Hardware_Clock_Time_Type)
    return Float;

-- Subprogram to compute elapsed time in ticks
function Elapsed_Time
    (Start_Time, Final_Time : Hardware_Clock_Time_Type)
    return Hardware_Clock_Time_Type;

-- Subprogram to read number of ticks marked by clock
function Read_Clock return Hardware_Clock_Time_Type;

private

Clock_Size           : constant := Long_Word_Size;
Counter_Range_Max    : constant := Max_Long_Word_Value;
Clock_Channel_Low_Word : constant := Channel_0;
Clock_Channel_High_Word : constant := Channel_1;
Use_Channel          : Select_Channel_Array_Type :=
    (True, True, False);

type Hardware_Clock_Time_Type is new Long_Word;

end Hardware_Clock_Package;

```

Shown below is the specification of Interface_to_8254.


```

package Interface_to_8254 is
  type Counting_Format_Type is (Binary, BCD);
  type Permissible_Modes_Type is (Mode_0, Mode_1, Mode_2,
                                   Mode_3, Mode_4, Mode_5);
  type Permissible_Channels_or_Read_Back_Type is
    (Channel_0, Channel_1, Channel_2, Read_Back);
  subtype Permissible_Channels_Type is
    Permissible_Channels_or_Read_Back
    range Channel_0 .. Channel_2;
  type Channel_Word_Array_Type is array
    (Permissible_Channels) of Word;
  type Select_Channel_Array_Type is array (Permissible_Channels)
    of Boolean;

  function Read_Multiple_Channels
    (Read_Channel : Select_Channel_Array_Type)
    return Channel_Word_Array_Type;
  function Channel_Clock_Frequency
    (Channel_Number : Permissible_Channels_Type)
    return Float;
  procedure Set_Counting_Format
    (Channel_Number : in Permissible_Channels_Type;
     Counting_Format : in Counting_Format_Type);
  procedure Set_Operating_Mode
    (Channel_Number : in Permissible_Channels_Type;
     Operating_Mode : in Permissible_Modes_Type);
end Interface_to_8254;

```

The package body of Hardware_Clock_Package follows:

```

package body Hardware_Clock_Package is

  procedure Initialize_Hardware_Clock is

    begin -- Initialize_Hardware_Clock

      null;

    end Initialize_Hardware_Clock;

  function Seconds
    (Ticks : Hardware_Clock_Time_Type)
    return Float is

    begin -- Seconds

      return Float (Hardware_Clock_Time) /
        Channel_Clock_Frequency (Clock_Channel_Low_Word);

    end Seconds;

end Hardware_Clock_Package;

```

```

function Elapsed_Time
    (Start_Time, Final_Time : Hardware_Clock_Time_Type)
    return Hardware_Clock_Time_Type is

```

```

begin -- Elapsed_Time

```

```

    if Start_Time <= Final_Time then
        return Final_Time - Start_Time;
    else
        return (Counter_Range_Max - Start_Time) + 1 + Final_Time;
    end if;

```

```

end Elapsed_Time;

```

```

function Read_Clock return Hardware_Clock_Time_Type is

```

```

    Channel_Reading : Channel_Word_Array;

```

```

begin -- Read_Clock

```

```

    Channel_Reading := Read_Multiple_Channels
                        (Read_Channel => Use_Channel);
    return Hardware_Clock_Time_Type
           (Words_to_Long_Word
            (Channel_Reading (Clock_Channel_High_Word),
             Channel_Reading (Clock_Channel_Low_Word)));

```

```

end Read_Clock;

```

```

end Hardware_Clock_Package; -- package body

```

The fact that `Hardware_Clock_Time_Type` is declared to be a private type means that users of `Hardware_Clock_Package` may assign to variables of this type, apply the equality / inequality operations or use them as actual parameters in the subprograms described in the specification. For example, given the object declaration in a compilation unit that imports `Hardware_Clock_Package`

```

Hardware_Clock_Time : Hardware_Clock_Time_Type;

```

then the following statements are legal:

```

Hardware_Clock_Time := Read_Clock;
if Seconds
    (Elapsed_Time
     (Hardware_Clock_Time, Read_Clock)) > 0.0001 then
    Time_Out_Physical_Line;
end if;

```

The following statement, however, is illegal

```
Hardware_Clock_Time := Hardware_Clock_Time + 1;
```

because addition is an operation which is neither predefined for any private type nor specified in the specification of `Hardware_Clock_Package`. Furthermore, such an operation requires knowledge of information which is hidden from users of `Hardware_Clock_Package` through the use of the private type declaration.

In contrast, consider the package body for `Hardware_Clock_Package`. Inside this package body, the implementation knowledge is available, and because the type is implemented as a numeric type, arithmetic operations are allowed, as in the function body of `Elapsed_Time`. The distinction between the operations allowed inside the package body and those allowed outside the body (in other words to the package importer) is an important mechanism through which Ada implements information hiding.

One of the benefits of information hiding is that the user of the abstraction operates on it independent of the implementation. Suppose, for example, that one day in the system's life cycle the clock is replaced by a different unit, capable of measuring longer intervals. The package body may have to be entirely rewritten, but it will not be necessary to hunt for other parts of the software affected by the change, since the rest of the software is not allowed (by the rules of Ada) to make use of any knowledge of what is inside the package. Thus the formal interface imposed by the package specification acts as a barrier against rippling effects of change.

Just as `Hardware_Clock_Package` represents the logical interface between the software and the clock, the package `Interface_to_8254` encapsulates the physical hardware characteristics. This package uses low-level features such as shared variables and representation specifications. Such features are machine dependent. Hiding them in the package body and isolating them from the rest of the software increase the maintainability of the program. Furthermore, this programming practice enforces a separation of concerns of the logical from the physical, fulfilling one of the goals of information hiding.

3. EPILOGUE

A good measure of a data abstraction is to analyze the way in which the abstraction is used. This package, `Hardware_Clock_Package`, is imported by another package called `Physical_Port`, which handles the transmissions along the physical lines. Within `Physical_Port`, several task types are defined to handle the different modes of transmission that in fact occur. Examination of the code for the task body for the task type `Send_Mode_V` reveals several interesting declarations, shown below:

```
type Milliseconds is delta 0.1 range 0.0 .. 2.0E9;

Start_Time : Hardware_Clock_Package.Hardware_Clock_Time_Type;

function Timer return Milliseconds is
    use Hardware_Clock_Package;
begin -- Timer
    return Milliseconds
        (Seconds
         (Elapsed_Time (Start_Time,
                       Read_Clock)) / 1000.0);
end Timer;
```

The variable `Start_Time` is defined here so that each task object of type `Send_Mode_V` can have its own time marker. It may seem misplaced, though, that the type `Milliseconds` and the function `Timer` are defined here. The reader may feel that these functions belong in `Hardware_Clock_Package` because these are detailed manipulations that he would expect to be a part of the interface of the package. In other words, the package interface of `Hardware_Clock_Package` is at too low a level of abstraction in that it is too close to the physical reality; one might prefer a more user-oriented abstract clock, which measures elapsed times (rather than returning a simple reading of the counter) and measures them in milliseconds.

The designers argued that the more user-oriented interface would be tailored to one particular usage. An unanticipated change in requirements might reveal that the package was hiding too much. The argument is valid; in the ultimate analysis it is the designer's intuition that dictates the trade-off.

With the current package interface, if the user feels strongly that it is too primitive, then he can always add a higher level of interface between the hardware clock package and his code:

```

with Hardware_Clock_Package;
package Timer_Service_Package is
  type Microseconds is delta 0.1 range 0.0 .. 2.0E12;

  procedure Start_Timer (Timer : out Microseconds);
  function Elapsed_Time (Since : Microseconds) return Microseconds;

end Timer_Service_Package;

```

On the other hand, hiding too much is not a catastrophic mistake. If, in view of a requirement change, the package specification is found too restrictive, it is very simple to add new operations to the specification. In the example discussed in this case study, the hardware of the clock is hidden, and the package user does not know that it is in fact a binary counter. Should this information be needed, then the package designer could provide an additional operation to determine whether, for example, the clock has begun recycling.

THIS PAGE INTENTIONALLY LEFT BLANK

2.5.3 REDUCING DEPTH OF NESTING

1. BACKGROUND

Case Study Objective

To show how appropriate use of Ada constructs can avoid the need for deeply-nested structures.

Problem

Excessive nesting of statements and subprograms makes programs hard to read. As readers descend more deeply into a nested structure, they have difficulty remembering the context of outer levels. This makes it difficult to understand inner structures that depend on outer structures, or to remember where one was in an outer structure after getting through a long inner structure. When nested structures span several pages, it is hard to match the ends of statements and unit bodies with their beginnings, and indentation ceases to be of any help. It becomes difficult to find all the declarations in effect at a given point.

Discussion

There are several circumstances in which programmers mistakenly perceive that Ada requires them to use a nested structure. Often, Ada provides mechanisms that can eliminate the need to nest. By using these constructs to "flatten" the program structure, we can obtain a program that is not only easier to read, but logically simpler.

This case study examines three separate instances of deep nesting. One involves nesting of block statements, one involves nesting of select statements, and one involves heavily nested program units (packages, subprograms, and tasks).

2. DETAILED EXAMPLES

Example Problem Statement (1)

A message switch maintains a log in which it records important events. Among these events are the transmission or reception of the start of a message, or SOM, the transmission or reception of the end of a message, or EOM, and various forms of message cancellation and rejection. The enumerated type `Log_Entry_Class_Type` has one value for each class of log entry:

```
type Log_Entry_Class_Type is
  (SOM_In, EOM_In, Reject, Cancel_In, CANTRAN_In, SOM_Out,
   EOM_Out, Cancel_Out, CANTRAN_Out, Scrub, Out_To_Overflow,
   In_From_Overflow, Out_To_Interrupt, In_From_Interrupt,
   Service_Gen, Version_Out, Version_In);
```

Log entries are made by activating a task of type `Logging_Task` and calling the only entry of that task, `Log`, with a parameter of type `Log_Request_Type`. `Log_Request_Type` is a record type with a `Log_Entry_Class_Type` discriminant. Its declaration has the following form:

```
type Log_Request_Type
  (Log_Entry_Class : Log_Entry_Class_Type := SOM_In) is
  record
    Message_Part : Message_Type;
    ...
    case Log_Entry_Class is
      ...
    end case;
  end record;
```

Within the `Logging_Task` body, the actual log entry is made by calling the subprogram `Write_History` with an argument of type `Entry_Set_Type`, which is an array of records of type `History_Entry_Type`:

```
type History_Entry_Class_Type is
  (Log_Entry, RI_Set_Entry, Segment_Entry);

type RI_Array_Type is
  array (Positive range <>) of RI_Type; -- "R.I." stands for
                                         -- "Routing Indicator"
```



```

type History_Entry_Type
  (Entry_Class      : History_Entry_Class_Type := Segment_Entry;
   RI_Count         : Natural := 0;
   Log_Entry_Class  : Log_Entry_Class_Type := SOM_In) is
  record
    case Entry_Class is
      when Log_Entry =>
        ...
        case Log_Entry_Class is
          ...
        end case;
      when RI_Set_Entry =>
        RI_Array : RI_Array_Type (1 .. RI_Count);
      when Segment_Entry =>
        Segment_Part : Segment_Type;
    end case;
  end record;

type Entry_Set_Type is
  array (Positive range <>) of History_Entry_Type;

```

The Entry_Set_Type array will be one of the three forms shown in Figure 1. In the case of a SOM_Out log entry, the Entry_Set_Type array contains one record with an Entry_Class discriminant of Log_Entry, followed by one record with an Entry_Class discriminant of RI_Set_Entry, followed by one or more records with an Entry_Class discriminant of Segment_Entry, one for each segment of the message header. In the case of a Reject log entry, the Entry_Set_Type array contains only one record, with an Entry_Class discriminant of Log_Entry. For any other log entry, the Entry_Set_Type array has two components -- a record with an Entry_Class discriminant of Log_Entry followed by a record with an Entry_Class discriminant of RI_Set.

Each Logging_Task object begins by accepting a value of type Log_Request_Type and making a copy of it for use outside the accept statement. A case statement examines the Log_Entry_Class component of the log request to determine the required size of the Entry_Set_Type array. In the case of a SOM_Out log request, this determination entails calling the function Number_Of_Header_Segments to find the number of segments in the message header, since the array will contain a component for each segment. The remainder of the processing takes place inside a block statement that declares an Entry_Set_Type array of the appropriate size. This processing depends on the class of the log entry, so the executable part of the block consists of a second case statement based on the Log_Entry_Class component of the log request record. The general pattern is as follows: The first component of the Entry_Set_Type array is filled in a manner that varies for each class of log entry. The second component is then filled with routing indicators in a uniform manner for all classes of log entry, by a call on the procedure Add_RIs. The actual log entry is made by calling procedure Write_History with the Entry_Set_Type array as a parameter. Finally, the operator is notified

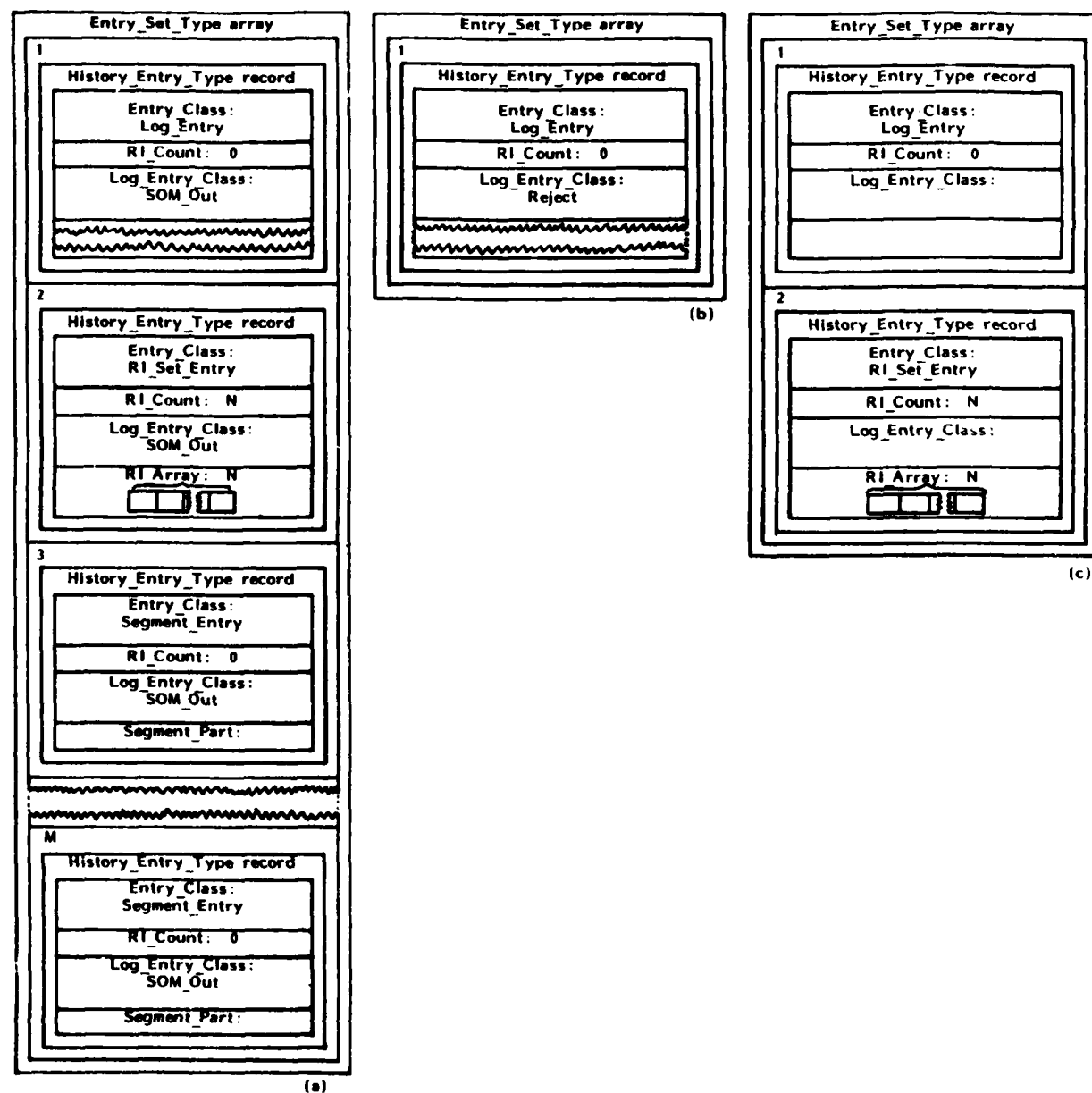


Figure 1. The three possible forms of an **Entry_Set_Type array**. Form (a) is passed to **Write_History** to produce a log entry of class **SOM_Out**, form (b) is passed to **Write_History** to produce a log entry of class **Reject**, and form (c) is passed to **Write_History** to produce any other class of log entry.

of the event that has been logged. The exceptions to this general pattern are as follows: When the log entry class is SOM_In, EOM_In, Cancel_In, CANTRAN_In, Out_To_Overflow, In_From_Overflow, Out_To_Interrupt, In_From_Interrupt, Service_Gen, Version_Out, or Version_In, the only action is to notify the operator. When the log entry class is Reject, there is no second component to be filled with routing indicators. When the log entry class is EOM_Out, the call on Add_RIs is followed by statements to obtain the header segments of the message and place them in subsequent elements of the Entry_Set_Type array. Header segments are obtained by calling the procedure Get_Header_Segments with an in parameter containing the message and an out parameter belonging to type Segment_Array_Type, declared as follows:

```
type Segment_Array_Type is array (Positive range <>) of Segment_Type;
```

Each component of the array is filled with one of the segments. The code to process the header segments is placed in a nested block statement containing a declaration of a Segment_Array_Type variable of the appropriate size. The Logging_Task body reads as follows:

```

task body Logging_Task is

    Request_Copy          : Log_Request_Type;
    Header_Segment_Count,
        Entry_Set_Size    : Natural;

begin -- Logging_Task

    accept Log (Request : Log_Request_Type) do
        Request_Copy := Request;
    end Log;

    case Request_Copy.Log_Entry_Class is
        when SOM_Out => -- log entry + RI's + header segments
            Header_Segment_Count :=
                Number_Of_Header_Segments (Request_Copy.Message_Part);
            Entry_Set_Size := 2 + Header_Segment_Count;
        when Reject => -- log entry only
            Entry_Set_Size := 1;
        when others => -- log entry + RI's
            Entry_Set_Size := 2;
    end case;

    Build_Entry_Set:
    declare

        Entry_Set : Entry_Set_Type (1 .. Entry_Set_Size);

        procedure Add_RIs is

            begin -- Add_RIs

                [statements to place routing indicators
                 and other information in Entry_Set(2) ]

            end Add_RIs;

```

```

begin -- Build_Entry_Set block

  case Request_Copy.Log_Entry_Class is

    when SOM_Out =>
      [statement to fill in Entry_Set(1)];
      Add_RIs;

      Insert_Header_Segments:
        -- This block exists to declare an array of the
        -- required size to place the header segments in.
        declare

          Segment_Array :
            Segment_Array_Type(1..Header_Segment_Count);

        begin -- Insert_Header_Segments

          Get_Header_Segments (Request_Copy.Message,
                               Segment_Array);

          for I in 2 .. Header_Segment_Count loop
            [statement to fill in Entry_Set(2+I) with
             Segment_Array(I) and other information];
          end loop;

        end Insert_Header_Segments;

      Write_History (Entry_Set);
      [statements to notify operator];

    when EOM_Out =>
      [statement to fill in Entry_Set(1)];
      Add_RIs;
      Write_History (Entry_Set);
      [statements to notify operator];

    when Cancel_OUT | CANTRAN_Out =>
      [statement to fill in Entry_Set(1)];
      Add_RIs;
      Write_History (Entry_Set);
      [statements to notify operator];

    when Scrub =>
      [statement to fill in Entry_Set(1)];
      Add_RIs;
      Write_History (Entry_Set);
      [statements to notify operator];
  end case;
end Build_Entry_Set block;

```

```

        when Reject =>
            [statement to fill in Entry_Set(1)];
            Write_History (Entry_Set);
            [statements to notify operator];

        when others =>
            [statements to notify operator];

    end case;

end Build_Entry_Set;

end Logging_Task;

```

The statements designated as "[statement to fill in Entry_Set(1)]" and "[statements to notify operator]" are different on each branch of the case statement.

Solution Outline (1)

The Logging_Task body uses block statements in order to create arrays whose size is not known until a certain point in the program. This can also be accomplished by declaring access types designating those arrays and allocating the arrays dynamically. Once the block statements are eliminated, the structure of the program can be simplified even further, because two case statements performing the same test can be merged into a single case statement.

Detailed Solution (1)

Let us define access types named Entry_Set_Pointer_Type and Segment_Array_Pointer_Type, pointing to objects of type Entry_Set_Type and Segment_Array_Type, respectively. Access variables Entry_Set, of type Entry_Set_Pointer_Type, and Segment_Array, of type Segment_Array_Pointer_Type, will be declared in the declarative part of the Logging_Task body. The statements nested in block statements will be merged into the surrounding sequences of statements. An Entry_Set_Type object will be dynamically allocated and its pointer assigned to Entry_Set at the point where Entry_Set had previously been declared in a block statement. A Segment_Array_Type object will be dynamically allocated and its pointer assigned to Segment_Array at the point where Segment_Array had previously been declared in a block statement. References to the whole variables Entry_Set and Segment_Array will be changed to Entry_Set.all and Segment_Array.all to reflect the fact that these are now access type variables rather than array type variables. References to components of these variables need not be changed, since Entry_Set(i), for example, can refer either to a component of the array Entry_Set or to a component of the array designated by the access value Entry_Set. The resulting program is as

follows:

task body Logging_Task is

```
type Entry_Set_Pointer_Type is access Entry_Set_Type;
type Segment_Array_Pointer_Type is access Segment_Array_Type;
```

```
Entry_Set           : Entry_Set_Pointer_Type;
Segment_Array       : Segment_Array_Pointer_Type;
Request_Copy        : Log_Request_Type;
Header_Segment_Count,
  Entry_Set_Size     : Natural;
```

procedure Add_RIs is

begin -- Add_RIs

```
  [statements to place routing indicators
   and other information in Entry_Set(2) ]
```

end Add_RIs;

begin -- Logging_Task

```
accept Log (Request : Log_Request_Type) do
  Request_Copy := Request;
end Log;
```

```
case Request_Copy.Log_Entry_Class is
  when SOM_Out => -- log entry + RI's + header segments
    Header_Segment_Count :=
      Number_Of_Header_Segments (Request_Copy.Message_Part);
    Entry_Set_Size := 2 + Header_Segment_Count;
  when Reject => -- log entry only
    Entry_Set_Size := 1;
  when others => -- log entry + RI's
    Entry_Set_Size := 2;
end case;
```

-- Former beginning of Build_Entry_Set block.

```
Entry_Set := new Entry_Set_Type (1 .. Entry_Set_Size);
```

```

case Request_Copy.Log_Entry_Class is
  when SOM_Out =>
    [statement to fill in Entry_Set(1)];
    Add_RIs;

    -- Former beginning of Insert_Header_Segments block.

    Segment_Array :=
      new Segment_Array_Type (1 .. Header_Segment_Count);

    Get_Header_Segments(Request_Copy.Message,Segment_Array.all);

    for I in 2 .. Header_Segment_Count loop
      [statement to fill in Entry_Set(2+I) with
        Segment_Array(I) and other information];
    end loop;

    -- Former end of Insert_Header_Segments block.

    Write_History (Entry_Set.all);
    [statements to notify operator];

  when EOM_Out =>
    [statement to fill in Entry_Set(1)];
    Add_RIs;
    Write_History (Entry_Set.all);
    [statements to notify operator];

  when Cancel_OUT ; CANTRAN_Out =>
    [statement to fill in Entry_Set(1)];
    Add_RIs;
    Write_History (Entry_Set.all);
    [statements to notify operator];

  when Scrub =>
    [statement to fill in Entry_Set(1)];
    Add_RIs;
    Write_History (Entry_Set.all);
    [statements to notify operator];

  when Reject =>
    [statement to fill in Entry_Set(1)];
    Write_History (Entry_Set.all);
    [statements to notify operator];

  when others =>
    [statements to notify operator];

end case;

```


-- Former end of Build_Entry_Set block.

end Logging_Task;

These transformations greatly elucidate the algorithm and the syntactic structure of the statements in the Logging_Task body. As a matter of fact, it now becomes evident that the task body can be further simplified, by merging the two case statements. Essentially, the first case statement is deleted and statements allocating various sizes of Entry_Set_Type objects are inserted into the arms of the second case statement. The result is a much more straightforward expression of the algorithm, in which the role of each component of the Entry_Set array is obvious:

task body Logging_Task is

type Entry_Set_Pointer_Type is access Entry_Set_Type;
type Segment_Array_Pointer_Type is access Segment_Array;

Entry_Set : Entry_Set_Pointer_Type;
Segment_Array : Segment_Array_Pointer_Type;
Request_Copy : Log_Request_Type;
Header_Segment_Count, Entry_Set_Size : Natural;

procedure Add_RIs is

begin -- Add_RIs

 [statements to place routing indicators
 and other information in Entry_Set(2)]

end Add_RIs;

```

begin -- Logging_Task

  accept Log (Request : Log_Request_Type) do
    Request_Copy := Request;
  end Log;

  case Request_Copy.Log_Entry_Class is

    when SOM_Out =>
      Header_Segment_Count :=
        Number_Of_Header_Segments (Request_Copy.Message_Part);
      Entry_Set := new Entry_Set (1 .. 2 + Header_Segment_Count);
      [statement to fill in Entry_Set(1)];
      Add_RIs;
      Segment_Array :=
        new Segment_Array_Type (1 .. Header_Segment_Count);
      Get_Header_Segments
        (Request_Copy.Message, Segment_Array.all);

      for I in 2 .. Header_Segment_Count loop
        [statement to fill in Entry_Set(2+I) with
          Segment_Array(I) and other information];
      end loop;
      Write_History (Entry_Set.all);
      [statements to notify operator];

    when EOM_Out =>
      Entry_Set := new Entry_Set_Type (1 .. 2);
      [statement to fill in Entry_Set(1)];
      Add_RIs;
      Write_History (Entry_Set.all);
      [statements to notify operator];

    when Cancel_Out | CANTRAN_Out =>
      Entry_Set := new Entry_Set_Type (1 .. 2);
      [statement to fill in Entry_Set(1)];
      Add_RIs;
      Write_History (Entry_Set.all);
      [statements to notify operator];

    when Scrub =>
      Entry_Set := new Entry_Set_Type (1 .. 2);
      [statement to fill in Entry_Set(1)];
      Add_RIs;
      Write_History (Entry_Set.all);
      [statements to notify operator];
  end case;
end Logging_Task;

```

```
when Reject =>
    Entry_Set := new Entry_Set_Type (1 .. 1);
    [statement to fill in Entry_Set(1)];
    Write_History (Entry_Set.all);
    [statements to notify operator];

when others =>
    [statements to notify operator];

end case;

end Logging_Task;
```

Example Problem Statement (2)

There is a task whose role is to communicate with several other tasks. As long as there is an appropriate entry in one of these other tasks ready to be called, the task is to call it. When there are no longer any appropriate entries ready to be called, the task is to perform final actions and go on to other work. This is accomplished by a heavily-nested structure of the following form:

```
loop
  select
    [entry call 1];           -- call 1 if possible
    [follow-up actions 1];
  else
    select
      [entry call 2];         -- else call 2 if possible
      [follow-up actions 2];
    else
      select
        [entry call 3];       -- else call 3 if possible
        [follow-up actions 3];
      else
        [final actions];      -- else perform final actions and
        exit;                 -- exit loop
      end select;
    end select;
  end select;
end loop;
```

The follow-up actions are quite involved, so that parts of this loop appear on three different pages of the source listing.

A conditional entry call is a select statement consisting of an entry call optionally followed by other statements and an "else part" specifying an action to perform if the entry call cannot be performed immediately. The only way to conditionally execute whichever of several entry calls is immediately executable is to use several such conditional entry calls in combination.

Solution Outline (2)

The most obvious solution is to redesign the tasking structure of the program so that the rendezvous go in the opposite direction. Then the entry calls can be replaced by accept statements. A set of arbitrarily many accept statements can be enclosed in a selective wait without nesting the accept statements. However, this kind of redesign is not always possible.

An alternative is available to programmers willing to use goto statements in a very restricted and disciplined way — to branch to the end of the immediately enclosing sequence of statements. The three nested select statements can be rewritten with goto statements as a sequence of three non-nested select statements. Use of the goto statement in this way may actually improve readability.

Detailed Solution (2)

If the direction of the rendezvous can be reversed, so that the entry calls become accept statements, the nested select statements specifying conditional entry calls can be replaced by a single select statement acting as a selective wait:

```
loop
  select
    [accept statement 1];
    [follow-up actions 1];
  or
    [accept statement 2];
    [follow-up actions 2];
  or
    [accept statement 3];
    [follow-up actions 3];
  else
    [final actions];
    exit;
  end select;
end loop;
```

This solution assumes that the order in which the various rendezvous are considered is not critical. When two or more accept statements are simultaneously eligible for execution, a selective wait chooses one of them arbitrarily. Guards can be used to enforce the order, as described in the case study on preferential acceptance of entry calls.

Other considerations may make it impossible to reverse the direction of the rendezvous. One of the entries called in the original program may be an interrupt entry, which can be called as a result of either a hardware interrupt or the execution of an entry call statement. The loop containing the entry calls may occur in a subprogram body, where the rules of Ada forbid accept statements. The flexibility available with accept statements may be required at the other end of the rendezvous, and it may be too complicated or expensive to introduce an interface task that calls two tasks so that they may communicate with each other through accept statements.

There is another alternative to nesting conditional entry calls, however. The loop containing the entry calls can be rewritten as follows:

loop

```
select
  [entry call 1];
  [follow-up actions 1];
  goto Bottom_Of_Loop;
else
  null;
end select;
```

```
select
  [entry call 2];
  [follow-up actions 2];
  goto Bottom_Of_Loop;
else
  null;
end select;
```

```
select
  [entry call 3];
  [follow-up actions 3];
  goto Bottom_Of_Loop;
else
  null;
end select;
```

-- This point is reached when no rendezvous is possible.

[final actions];

exit;

```
<< Bottom_Of_Loop >>
  null;
```

end loop;

Semantically, this is precisely equivalent to the original program, preserving the order in which each potential rendezvous is considered. It reflects a symmetry among the three alternatives that is not obvious in the nested form. It is to emphasize this symmetry, and to highlight the idiomatic use of the goto statement, that the third select statement, the final actions, and the exit statement were not rewritten as follows:

```

select
  [entry call 3];
  [follow-up actions 3];
else
  [final actions];
  exit;
end select;

```

The solution chosen makes it possible to consider each select statement in turn, dismissing one select statement before going on to consider the next.

Example Problem Statement (3)

A message switch contains a package that is extremely large because it contains many nested units. The nesting is deep: at one point, there are subprograms nested in another subprogram nested in a task body nested in the package body. In all, there are 23 units nested at various levels inside the package. In many places, one programmer's work is nested inside another's. The source listing for the package extends over 63 pages. The following outline describes the pattern of the nesting:

```

package body P (written by programmer 1)
  task body T1 (written by programmer 1)
    variable declarations for T1
    function F1 (written by programmer 2)
      type and variable declarations for F1
      procedure P1 (written by programmers 2 and 3)
        variable declarations for P1
        statements for P1
      function F2 (written by programmer 2)
        subtype and variable declarations for F2
        statements for F2
      function F3 (written by programmer 2)
        variable declarations for F3
        statements for F3
      procedure P2 (written by programmer 3)
        variable declaration for P2
        statements for P2
      statements for F1
    statements for T1
  task body T2 (written by programmer 1)
    subtype declarations for T2
    statements for T2
  function F4 (written by programmer 1)
    subtype and object declarations for F4
    statements for F4

```

```

task body T3 (written by programmers 2, 3, and 4)
  type and variable declarations for T3
task specification T4 (written by programmer 2)
  entry declarations for T4
task body T4
  variable declaration for T4
  statements for T4
procedure P3 (anonymous)
  constant declaration for P3
  statements for P3
statements for T3
task body T5 (written by programmers 3 and 4)
  type and object declarations for T5
  function F5 (written by programmer 2)
    variable declaration for F5
    statements for F5
task specification T6 (written by programmer 2)
  entry declarations for T6
task body T6
  variable declaration for T6
  statements for T6
procedure P4 (written by programmer 2)
  constant declaration for P4
  statements for P4
procedure P5 (anonymous)
  statements for P5
procedure P6 (anonymous)
  statements for P6
procedure P7 (written by programmer 2)
  statements for P7
procedure P8 (anonymous)
  constant declaration for P8
  statements for P8
procedure P9 (written by programmer 2)
  statements for P9
statements for T5
task body T7 (written by programmer 1)
  variable declarations for T7
  statements for T7
task body T8 (written by programmer 1)
  variable declarations for T8
  statements for T8
  block statement B1
    subtype and variable declarations for B1
    procedure P10 (written by programmer 1)
      statements for P10
    statements for B1
  block statement B2
    subtype and variable declarations for B2
    statements for B2
  statements for B1

```


There are many obvious problems with a package that is so large and so heavily nested. One is that the complexity of the package makes it difficult to maintain. Not only is the sheer volume of code overwhelming to a maintainer; the extensive use of global variables and other global entities makes the package hard to understand, by obscuring the ways in which different parts of the package interact. The context specifications at the beginning of the package make any imported entity used anywhere throughout the package available throughout the entire package. This makes it hard to find all references to the entity.

It is very hard to find the declaration for a given entity, because there may be many pages intervening between the declaration of an entity and its use, and these pages may include other declarative parts. For example, suppose a programmer has to find the declaration of an entity used in a statement of procedure P2, and the declaration is located in the declarative part of T1's task body. He must search backwards through the declarative parts of procedure P2, function F3, function F2, procedure P1, function F1, and task body T1 (along with the intervening sequences of statements) to find the declaration. During this search, he must keep track of which units surround P2 and which do not. A declaration of the identifier in question is irrelevant if it occurs in the declarative part of F3, P2, or P1, but it may be the sought-after declaration if it occurs in the declarative part of P2, F1, or T1.

The logistics of revising this package are cumbersome. A managerial problem arises because different programmers are responsible for parts of the same compilation unit, and will be in competition for access to the source file. A minor change in one line of one unit requires that the entire package be recompiled.

Solution Outline (3)

Package P can be made much easier to read and maintain by breaking it up into several separate compilation units. Breaking it into subunits is a mechanical transformation that eliminates the physical nesting but preserves the logical nesting. Breaking it into library units requires some redesign to handle references to global entities.

Detailed Solution (3)

Package P can be broken into separate compilation units by replacing bodies of nested units with body stubs. Then the package body and some of its subunits would read as follows:

```

package body P is
  task body T1 is separate;
  task body T2 is separate;
  function F4 ( ... ) return ... is separate;
  task body T3 is separate;
  task body T5 is separate;
  task body T7 is separate;
  task body T8 is separate;
end P;

```

```

separate (P)

```

```

task body T1 is
  [variable declaration for T1]
  function F1 ( ... ) return ... is separate;
begin -- T1
  [statements for T1]
end T1;

```

```

separate (P.T1)

```

```

function F1 ( ... ) return ... is
  [type and variable declarations for F1]
  procedure P1 ( ... ) is separate;
  function F2 ( ... ) return ... is separate;
  function F3 ( ... ) return ... is separate;
  procedure P2 ( ... ) is separate;
begin -- F1
  [statements for F1]
end F1;

```

separate (P.T1.F1)

```
procedure P1 ( ... ) is
  [variable declarations for P1]
begin -- P1
  [statements for P1]
end P1;
```

These subunits make the structure of the program much more evident than the nested version. The structure is exposed in a top-down manner, with each unit indicating the identities of its constituent units, but not the contents of those units. This actually makes the pattern of logical nesting clearer than the original, physically nested, package.

Program components are easier to follow because of the drastically shorter distance between the declarations and statements of a unit. The declaration of an entity used in a particular statement is easier to find. If the entity is declared locally, the declaration is nearby in the same compilation unit. If the entity is global, the separate clause at the beginning of the subunit names all the logically surrounding units that must be examined. The language rules require the name in the separate clause to be fully expanded to indicate the name of each logically surrounding unit.

A subunit may have its own context specification, giving it access to library units to which other parts of the parent unit do not have access. Thus library units used in only a few of the subunits of P can be deleted from the context specification for P and added to the context specifications for the subunits that use them. This makes it easier to find all uses of a given library unit within P and easier to determine the source of a name used but not declared in P.

Subunits obviously limit recompilation costs. When a subunit is modified, it and all of its descendants must be recompiled, but its ancestors, siblings, and cousins need not be. Partitioning a large unit into subunits makes project management easier, since rights to and responsibility for a particular file can be assigned to a single programmer.

Partitioning P into subunits does not eliminate the complexity that results from the extensive use of global entities. Subunits eliminate physical nesting, but not logical nesting. By physically separating references to global entities from their declarations, subunits may even make such references harder to follow. In contrast to the partition of P into subunits, which is a purely mechanical transformation, the elimination of global references requires careful thought.

One approach is to transform a subunit into a library unit that refers to its former parent in a with clause. Entities declared in the former parent and referred to in the former subunit must be declared in the visible part of the former parent. This has the advantage of

documenting that the entity forms part of the former parent's interface; in other words, it documents that the entity is used by other units. A variation -- indeed, the only variation available when the parent is a subprogram or task -- is to create a new library package named in the with clauses of both the former parent and former subunit and containing all the entities used by both of them.

Global variables occurring in logically nested subprograms can be replaced by new parameters. Again, this makes the program easier to understand by making the interface between its components more explicit. If the new parameters are declared as being of mode in or mode out, the program then conveys important data flow information that is absent in the original version.

3. EPILOGUE

Detailed solution 1 did not discuss when storage for dynamically allocated arrays is reclaimed. Fortunately, this is not a serious problem. A Logging_Task object is short-lived. It comes into existence when it is necessary to make a log entry, and terminates as soon as it completes this job. The access types Entry_Set_Pointer_Type and Segment_Array_Pointer_Type are declared inside the Logging_Task body. The storage for the one or two arrays allocated by each Logging_Task object is deallocated when the task terminates and the scope of the access type declarations is exited.

The use of the goto statement to reduce nesting, as suggested in Detailed Solution (2), is bound to be controversial. Unrestricted use of the goto statement can certainly make programs hard to read. However, the use of the goto statement in very restricted circumstances, such as to escape to the end of a sequence of statements, is usually not hard to understand. In some circumstances, it could well improve readability.

In the past, language designers have taken highly idiomatic uses of goto statements as evidence of the need for more powerful high-level control structures, such as the case statement and the exit statement. The problems described in Example Problem Statement (2) would not have arisen if Ada had a control structure of the form

```

select
    <entry call>
    <sequence of statements>

or else

    <entry call>
    <sequence of statements>

or else

    ...

or else

    <entry call>
    <sequence of statements>

else

    <sequence of statements>

end select;

```

defined as equivalent to the following nested structure:

```

select
    <entry call>
    <sequence of statements>
else
    select
        <entry call>
        <sequence of statements>
    else
        ...
        select
            <entry call>
            <sequence of statements>
        else
            <sequence of statements>
        end select;
    ...
end select;
end select;

```

(Besides being suggestive of the "else"'s occurring in the nested version, "or else" suggests an analogy to Boolean expressions. In both Boolean expressions and select statements, "or" would be used to separate the elements in a list of unordered alternatives and "or else" would be used to separate the elements in a list of alternatives to be

considered in sequence until a successful one was found.)

It is possible to format the nested select statements in a way which reflects that we are thinking of them as a list of alternatives:

```
select
  <entry call>
  <sequence of statements>
else select
  <entry call>
  <sequence of statements>
...
else select
  <entry call>
  <sequence of statements>
else
  <sequence of statements>
end select;
..
end select;
end select;
```

This is analogous to the way in which nested if statements are often formatted in languages without an elsif. Of course the long sequence of "end select;" closing delimiters at the end is disconcerting.

The effect of a goto statement exiting a surrounding sequence of statements can be simulated by Ada's *exception handling mechanism*. However, such use of exceptions is itself controversial. A concrete example and a discussion of the relevant issues can be found in the case study on the use of exceptions.

The effect of a goto statement exiting a surrounding sequence of statements can also be simulated by enclosing the sequence of statements in a loop and placing an exit statement everywhere the sequence of statements is to be abandoned, including at the end of the sequence. The loop is a loop in name only, since it never completes a full iteration. For example,

loop

```
select
  [entry call 1];
  [follow-up actions 1];
  goto Bottom_Of_Loop;
else
  null;
end select;
```

```
select
  [entry call 2];
  [follow-up actions 2];
  goto Bottom_Of_Loop;
else
  null;
end select;
```

```
select
  [entry call 3];
  [follow-up actions 3];
  goto Bottom_Of_Loop;
else
  null;
end select;
```

-- This point is reached when no rendezvous is possible.

[final actions];

exit;

```
<< Bottom_Of_Loop >>
  null;
```

end loop;

can be rewritten as follows:


```
Outer_Loop:
  loop
```

```
    False_Loop:
      loop
```

```
        select
          [entry call 1];
          [follow-up actions 1];
          exit False_Loop;
        else
          null;
        end select;
```

```
        select
          [entry call 2];
          [follow-up actions 2];
          exit False_Loop;
        else
          null;
        end select;
```

```
        select
          [entry call 3];
          [follow-up actions 3];
          exit False_Loop;
        else
          null;
        end select;
```

```
-- This point is reached when no rendezvous is possible.
```

```
    [final actions];
    exit Outer_Loop;
```

```
  end loop False_Loop;
```

```
end loop Outer_Loop;
```

(In this case, the "exit False_Loop;" statement at the bottom of the inner loop has been dropped because it would not be reachable immediately after the statement "exit Outer_Loop;".) This is a poor substitute for a statement which blocks what the exit statement is to loops. Although the word "goto" is dutifully eliminated from this code, it is harder to understand than the version containing the goto statement.

The ability to break up a monolithic expanse of code into separately compilable chunks, as suggested in Detailed Solution (3), may be Ada's greatest strength. Besides increasing readability, body stubs make it possible to reduce recompilation costs, give the programmers

responsible for a particular unit exclusive access to all of the unit's source code, and localize context specifications to clarify where a library unit's facilities are actually used. Many of these benefits can also be derived by breaking a large unit into several independent library units. The relative advantages and disadvantages of subunits and library units are discussed in the next case study.

2.5.4 LIBRARY UNITS VERSUS SUBUNITS

1. BACKGROUND

Case Study Objective

To compare the consequences of writing packages and subprograms as library units and as subunits.

Problem

Ada provides two ways of decomposing a large program into multiple compilation units. A package or subprogram can be compiled as a library unit, to be made available throughout the scope of any other unit naming the library unit in a "with" clause. Alternatively, a subprogram, package, or task can be logically nested inside another unit, but with a body stub replacing the body and the actual body compiled separately as a subunit. A subunit has access to the declarations it would have had access to had it been physically located in the parent unit. In addition, the subunit can have its own context clause, providing access to library units not available throughout the parent unit.

Designers and programmers need guidelines in choosing between these alternative methods of program decomposition.

Discussion

Subunits are often advertised as tools for top-down program development. During top-down program design, one determines the need for a particular facility and writes code using that facility before turning attention to how the facility is to be implemented. In Ada, one can write the specification of a subprogram, package, or task and write a body stub in place of a body. The program using the specified subprogram, package, or task can then be written and compiled immediately, with coding of the subunit body deferred to a later stage of stepwise refinement.

In contrast, library units are often advertised as tools for bottom-up program construction using independently produced components. A general purpose library package, written without any knowledge of how it will be used, can be incorporated as a building block in a program. Once such off-the-shelf software components are generally available, it may be possible to assemble whole systems almost entirely out of such building blocks.

In reality, this dichotomy is an oversimplification. Top-down design does not preclude the use of library units. For instance, it may be determined that a particular subprogram is needed in many different modules of a large program. The subprogram can be written as a library unit to be used by each module needing it. Similarly, packages implementing data abstractions may be written as library units. In a top-down design, the specification of the library unit is written as soon as the need for the unit is recognized; the body is written later.

Subunits may result from a coding or maintenance decision rather than a design decision. Pieces of a large unit may be broken off into subunits after the program has already been written. This might be done to make the program more readable (as explained in the case study on reducing depth of nesting, for instance) or to reduce recompilation costs.

The choice between library units and subunits involves a number of issues besides top-down program design and bottom-up program composition. Among these issues are:

- the manner in which a large program should be structured out of individual modules
- the role of nesting in determining the visibility of declarations
- the use of global data
- the amount of recompilation necessitated by a change to some module

2. DETAILED EXAMPLE

Example Problem Statement

The preliminary source code for a message switch contains 24 compilation units, each written as a library unit. The dependencies of these compilation units, as reflected in "with" clauses, are presented in Figure 1. When examining the source code, considerable effort is required just to determine how each unit fits into the system. It is not apparent from the source code that `Convert_To_ITA`, `History_Ops`, `Service_Message_Ops`, `Manage_Translated_Queues`, and `Hardware_Clock` are used only by the package `Physical_Port`, for instance, or that `Storage_Unit_Definitions` and `Interface_To_8254` are used only by `Hardware_Clock`.

The contractor was hesitant to use subunits, primarily out of fear of the impact on maintenance when a reference to a global entity appeared without the context of the parent unit. This is certainly a valid concern. Nonetheless, subunits could have done much to clarify the overall structure of the system.

Solution Outline

If a library unit A has only one other unit B dependent upon it, A can be nested inside B, with A's body compiled separately as a subunit. Applying this transformation wherever possible can clarify the role of each module in the message switch. Separate subunits can also make it easier to understand names declared in one compilation unit and used in another. The "separate" clause makes it easy to find declarations of global variables (though it remains difficult to find all uses of such variables); the effect of context specifications can be localized by moving context specifications from the parent unit to subunits whenever possible.

Use of separate subunits rather than library units enforces a tree-like structure upon the modules of a program. This results in a simple model of the program. However, the scope rules of Ada do not restrict intermodule communication to communication between parents and children in this tree. Library units can provide more precise control over intermodule interfaces, but the structure of the system will not be as apparent from the program text as when the system is built out of subunits.

To the extent that subunits encourage use of global data, they can make programs harder to follow. However, abuse of global data is also possible when the data is declared in the visible part of a library package. More importantly, logically nested program units do not require the use of global data. To the contrary, it is possible to

write units that serve only as containers for the subunits logically nested inside of them, to prevent some of those subunits from being referenced globally.

For most programs, combined use of library units and subunits is likely to be more beneficial than exclusive reliance on one or the other. In the case of the message switch, analysis of the intermodule dependencies reveals that the program is largely, but not entirely, hierarchical. Groups of modules can be combined into small hierarchies using subunits, with a library unit at the top of each hierarchy. These subunits can then be combined by context clauses. The result is a highly structured system whose basic design can be more simply depicted and more readily understood than the system depicted in Figure 1.

One of the inevitable practical considerations in any high-level design is the amount of recompilation that will be required when a given unit is modified. Building a program out of subunits rather than out of library units can increase the potential need for recompilation. However, a carefully considered design can minimize the amount of recompilation required in practice.

Detailed Solution

The message switch is built entirely out of library units. Every unit using another unit begins with one or more context clauses listing the units it uses. However, the program provides no information in the other direction, indicating which other units use a given unit. In a large system, it is difficult to gather and maintain this reverse information without the use of automated tools.

In contrast, imagine a system built entirely out of subunits. Body stubs fill the role played by context clauses when using library units. That is, they indicate which subunits are used by a given parent unit. In addition, each subunit begins with a "separate" clause naming its parent unit, the parent of its parent, and so forth. A subunit's "separate" clause indicates which unit uses that subunit. This two-way link makes it easier to learn and comprehend the module-level structure of a system.

The two-way link also makes it easier to determine the source of names appearing in a unit. A context clause containing a "use" clause for several library packages may introduce a multitude of names, and, without the output of a good cross-reference tool, it is not apparent which package a particular name comes from. However, if a subunit uses a name declared in a logically enclosing unit, the declaration can be found by examining only those units named in the "separate" clause. (See Example Problem (3) of the case study on reducing depth of nesting

for a more detailed discussion and a concrete example.)

Subunits can also make it easier to keep track of names by reducing the number of declarations visible at a given place. Suppose a library unit A is used in only a few places in some other unit B. If A is named in B's context clause, it will be visible throughout B. However, if the few places in B that actually use A are broken off into subunits, the reference to A can be removed from B's context clause and placed on the context clauses of those subunits.

Writing a package or subprogram as a library unit provides considerable freedom in the use of that unit. A library unit can be used by any other compilation unit. It can even be reused in another program. A subunit can only be used in the parent unit containing its stub. However, this lack of freedom makes it easier for a maintenance programmer to understand how a subunit fits into a large system.

When a library unit is used by only one other unit, it can be rewritten nested inside that other unit. The nested unit is then visible only within the surrounding unit. Because of this, the intended use of the nested unit, and its role within the system as a whole, are well documented. The body of the nested unit can be replaced by a body stub and rewritten as a subunit in order to preserve the division of the program into separate compilation units.

For example, the package `Asynch_Seq_Numbers` is used only inside the package `Global_Types`. `Asynch_Seq_Numbers` has a package body, but `Global_Types` does not. The body of `Asynch_Seq_Numbers` can be rewritten as a subunit, and `Global_Types` can be given a body containing only a stub for the body of `Asynch_Seq_Numbers`. The package specification of `Asynch_Seq_Numbers` can then be nested in the private part of the `Global_Types` specification. (This entails writing private declarations for any entities declared in `Global_Types` in terms of entities provided by `Asynch_Seq_Numbers`.) As a result of this transformation, a reader of the program knows that he need not be concerned with package `Asynch_Seq_Numbers`, except when trying to understand the implementation of package `Global_Types`.

In the same way, the package `Message_IO` can be nested in package `Manage_Overflow`, the package `Queue` can be nested in package `Manage_Translated_Queues`, and packages `Service_Message_Ops`, `History_Ops`, `Convert_To_ITA`, `Hardware_Clock`, and `Manage_Translated_Queues` can be nested in package `Physical_Port`. More interestingly, consider package `Storage_Unit_Definitions`, used both by package `Interface_To_8254` and by package `Hardware_Clock`. `Interface_To_8254` is used only by `Hardware_Clock`, so it can be nested in `Hardware_Clock` as described above. Once this is done, all uses of package `Storage_Unit_Definitions` are from within `Hardware_Clock`, so `Storage_Unit_Definitions` can also be nested within `Hardware_Clock`.

When compilation units are logically nested, we think of their interconnections differently. Rather than representing the entire system structure as an arbitrary acyclic network, as in Figure 1, we can depict nested units as internal components of other units, as in Figure 2. This makes the structure of the system somewhat clearer, since certain units can now be recognized as part of the implementation of other units.

More interesting than the transformation of library units into subunits is the initial design of a system using only subunits. A system designed in this way has a strict tree-like structure, and can be depicted entirely with nested circles. Turner [Tur80] argues that tree-structured systems are desirable because each branch of the tree performs its own function that can be understood in isolation from the other branches and because the topology of the tree reduces the number of interactions among modules. In general, a collection of n modules can contain up to $n(n-1)$ interfaces between modules, but a tree containing n modules contains only $n-1$ interfaces between modules. (Turner condones departure from a strict tree structure for common service routines with very simple specifications and limited, apparent interfaces.)

However, Clarke, Wileden, and Wolf [CWW80] argue that the natural structure of a program is usually not tree-like. Furthermore, they point out that nested program units do not really enforce tree structure. That is because the scope rules allow interactions not only between parent and child, but also between sibling units, between a unit and its higher-level ancestors, and between a unit and siblings of its ancestors. (Sibling subprograms can call each other, for example. Similarly, the siblings of a package can use the entities declared in that package's visible part.) Rather than using logically nested subunits to achieve an imprecise representation of the topology of a program, Clarke, Wileden, and Wolf advocate writing library units and using context clauses to construct that topology precisely. When data must be shared between two units, they advocate creation of a new library package, analogous to a FORTRAN named COMMON block, that contains only the declarations of the shared data and is referenced only by the modules sharing that data.

Nesting is often associated with the use of global variables -- that is, variables declared at the outer level of a large unit but used within several nested inner units. Wulf and Shaw [W&S73] argue that global variables complicate the interfaces of the nested units. This makes it hard to characterize the effect of the nested code in abstract terms. Procedures that have the side effect of altering the value of a global variable are especially troublesome to a program reader, because the variable is not even named in the procedure call statement that causes it to be changed. In a large program, the reader has no easy way to determine which inner units use a given global variable. Each use of the global variable must be understood without the context explaining how the variable is used by other parts of the program, or even how it

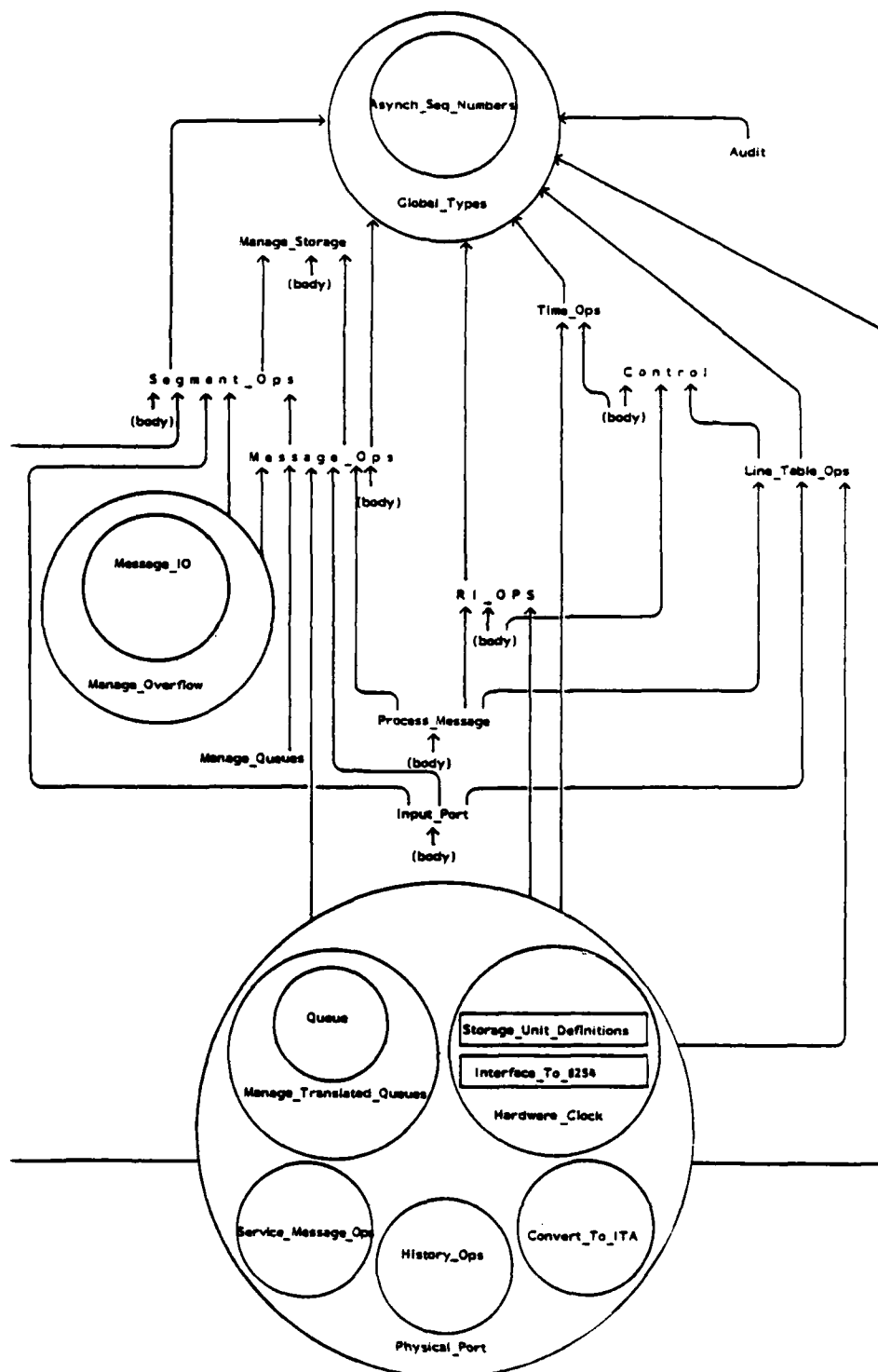


Figure 2. Revised structure of the message switch program, in which each library unit with only one module dependent on it is transformed into a subunit of that module. Parent units and their subunits are depicted as nested circles, reflecting their logical relationships. Physically, the parent units and subunits are still separate compilation units.

is declared. Separate compilation of subunits exacerbates this problem.

However, the use of nested program structure does not require the use of global variables. Besides making an outer entity available to many inner units, and thus allowing indiscriminate access to this entity, nesting can be used to make an inner entity unavailable to the outside world, thus encapsulating implementations of abstractions. An outer unit can be used simply as a container that holds several subunits but only allows some of them to be seen by the outside world.

To be used in this way, the outer unit should generally not contain any variable declarations or any significant amount of top-level code. When the outer unit is a package, only those nested entities to be made available to the outside world should be declared in the package specification, and the package body should contain only declarations of nested packages and stubs for nested package and procedure bodies. Any permanent data structure used by the package should be declared in a nested inner package.

When the outer unit is a procedure, the main algorithm should be performed by one of the nested procedures, so that the only statement in the outer procedure is a single procedure call. The inner procedures should all be separate subunits, represented in the outer procedure only by body stubs. The inner procedures should communicate only with parameters. (It can be argued that an outer procedure used in this way should really be written as a package, with the package specification containing only the specification of the nested procedure that executes the main algorithm. If so, procedures should never be nested.)

In the Mesa system [L&S79], large programs are built out of configurations, whose components may include subconfigurations. Configurations act as barriers, restricting the use of entities on one side of the barrier by entities on the other side. A configuration may explicitly export an entity for use by other configurations, or import an entity defined outside the configuration so that it may be used inside the configuration. A component of a configuration may only import an entity imported by the configuration or exported by another component of the configuration. A configuration may only export an entity exported by one of its components. Configurations, written in a special configuration description language called C/Mesa, define the interfaces between system components written in the Mesa programming language. They clarify the role of each Mesa program unit within a large system, distinguish between exported entities and their implementations, and communicate the programmer's abstractions to the program reader. They allow components of a program to be modified in isolation, and facilitate automatic checks that the modified components still conform to the interface defined by the configuration.

All these capabilities are also available in Ada, whose design was strongly influenced by Mesa. An Ada package acting only as a container serves the same function as a C/Mesa configuration. It specifies which compilation units are meant for the world at large and which compilation units are meant to implement those intended for the outside world. A significant difference is that in Ada, the configurations are written directly in the programming language. (Another difference is that in Ada, a component of a configuration can import an entity not explicitly imported by the configuration. Specifically, a subunit of a package acting as a container can have its own context clause.)

Neither the absolutist position that program structure should always be tree-like nor the absolutist position that programs should never be tree-like is likely to be optimal for most programs. The most effective use of Ada can be made by combining use of subunits -- to hide implementations of abstractions -- with library units connected by context clauses -- to connect those abstractions whose relationships are not hierarchical. For example, careful examination of the message switch system depicted in Figure 1 reveals the existence of two sets of units such that most of the units in the first set use most of the units in the second set, but none of the units in the second set uses any of the units in the first set. The first set consists of the units Physical_Port, Input_Port, Process_Message, Manage_Overflow, and Manage_Queues, along with the units depicted as subunits of these units in Figure 2. The second set consists of units Line_Table_Ops, Segment_Ops, Message_Ops, and RI_Ops. In essence, the second set acts as a set of service routines used in the implementation of the units in the first set. These service routines can be combined into a new package, Utility, that acts only as a container. Line_Table_Ops, Segment_Ops, Message_Ops, and RI_Ops are declared in the specification of Utility, and stubs for their bodies are given in the body of Utility. In the original system design, Message_Ops used the unit Segment_Ops. This can continue to be the case even when Segment_Ops and Message_Ops are both nested in Utility. The two units Manage_Storage and Control are used only by packages "exported" by Utility. (Manage_Storage is used only by Segment_Ops and Message_Ops; Control is used only by RI_Ops and Line_Table_Ops.) Consequently, they can be declared in the body of Utility, and hidden from the rest of the system.

The result of these transformations, shown in Figure 3, is a system consisting of nine library units with relatively simple interconnections. With the possible exception of Utility, the library units are strongly functionally cohesive. The functional cohesion of Utility is somewhat weaker, relying on its characterization as a package of service routines, but it is not absent. The text of the program provides the reader with very useful, easily spotted clues about the structure of the system and the role of each compilation unit within it.

In some ways, the system of Figure 3 provides less information than the system of Figure 1. For instance, the text of the program in Figure 3 does not specify which subunits of Utility use which other subunits.

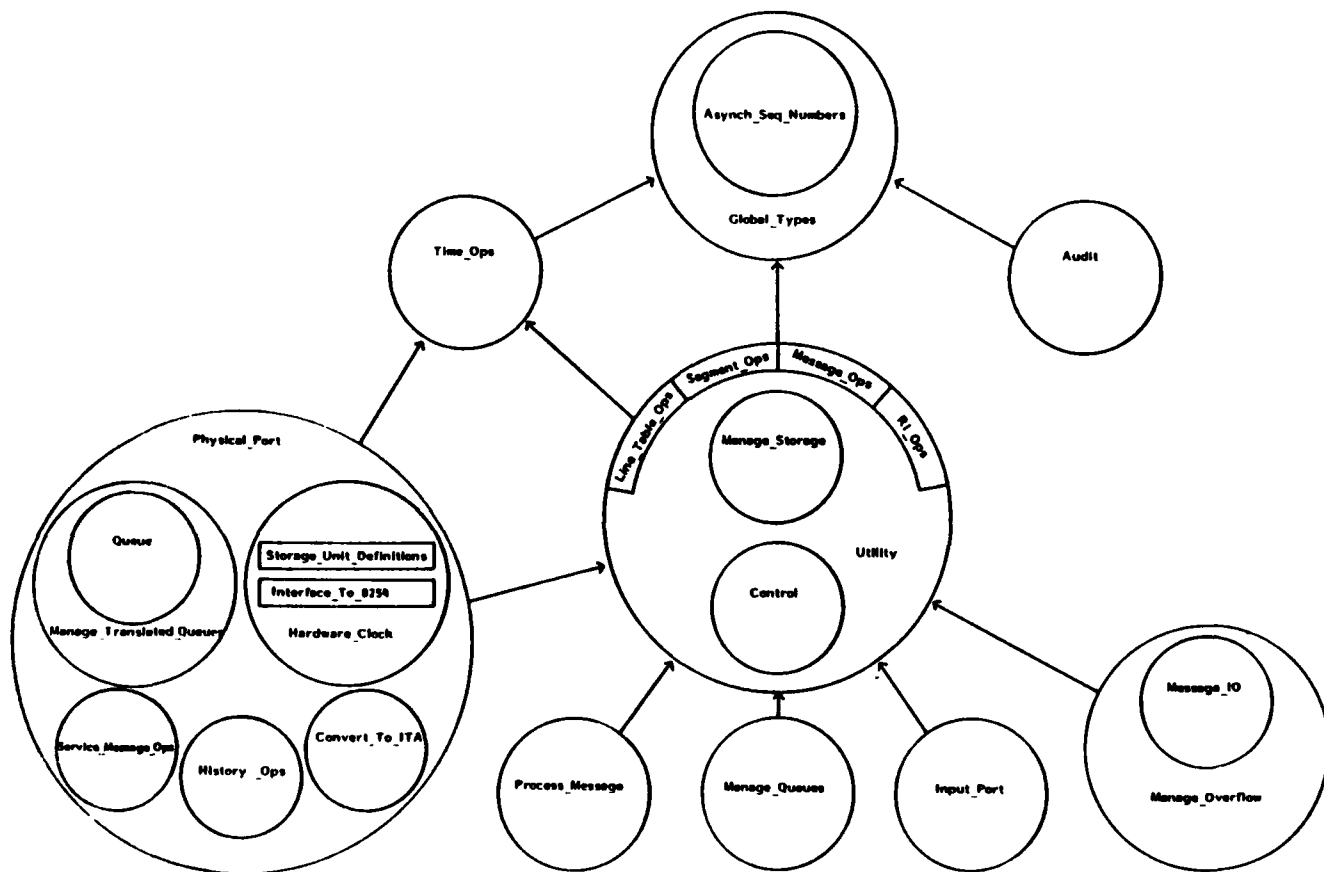


Figure 3. The result of incorporating several library units as subunits of a new "container" package named Utility. The four packages bordering the outer edge of Utility are declared in the visible part of Utility and provided to users of the package. The other two packages nested inside Utility are declared in the body of Utility. They are used to implement Utility and are not visible outside the package.

However, the information provided by the system in Figure 3 is at a higher level of abstraction, and conveys more of "the big picture."

An important consequence of the modular structure of a program is the amount of recompilation entailed by program changes. Experience with Mesa has shown that when system development is not carefully planned, or when interfaces between modules continue to evolve once coding begins, small changes can result in massive recompilations, often of the entire system. During the development of a personal computer operating system in Mesa [L&S79], recompilation marathons of from one and a half to two days were typical. Not only did this expend computing resources; members of the project team were unable to proceed with further development until recompilation was complete.

On the surface, use of subunits appears to increase the potential need for recompilation. Suppose library unit A is dependent on library units B, C, and D, and no other compilation unit is dependent on B, C, or D. Modification of the specification of B, C, or D requires recompilation of A, but modification of A does not require recompilation of B, C, or D. Now suppose B, C, and D are incorporated as subunits of A. Then recompilation of A requires recompilation of each of these subunits. Furthermore, the specifications of the subunits are now part of the text of A. A change to the specification of one of the subunits requires recompilation of A and hence of all three subunits.

(This analysis assumes that the Ada implementation requires recompilation whenever the language allows the implementation to impose this requirement. A clever implementation can be more discriminating, and determine, for instance, that a change to the specification of one subunit does not affect the consistency of a sibling subunit. In Figure 1, library package Line_Table_Ops is not dependent on library package Manage_Storage. In Figure 3, Line_Table_Ops and Manage_Storage are both subunits of Utility. A naive implementation will require recompilation of Line_Table_Ops when Utility is updated to change the specification of Manage_Storage. A clever implementation will conclude that if this is the only change to Utility, recompilation of Line_Table_Ops is unnecessary, but recompilation of subunits Segment_Ops and Message_Ops -- which refer to the specification of Manage_Storage -- is necessary. When library units are used only as containers for subunits, there are few actual interdependencies among subunits.)

To the extent that subunits enforce hierarchical system structure, they can actually help limit recompilation costs. Lauer and Satterthwaite [L&S79] observe that

Because of the hierarchical structure of [the personal computer operating system coded in Mesa], universal recompilations were rare. In most cases, only the components of one of the nested

configurations needed to be recompiled, requiring much less time and effort and affecting fewer people.

In a well-designed hierarchical system, the upper levels of the hierarchy reflect the fundamental decisions made early in the design process; the lower levels of the hierarchy represent more detailed decisions made later on. Early design decisions tend to be concerned with the definition of interfaces between modules, while later design decisions tend to be concerned with the implementation of modules. Rescinding an early, fundamental design decision entails massive recompilation, while rescinding a later design decision entails relatively little recompilation.

This property of hierarchical systems holds whether the hierarchy is specified by nesting or by context clauses. It places a premium on careful initial design and on a willingness to regard initial design decisions as binding commitments. In a hierarchical system, major changes entail much recompilation and minor changes entail little recompilation, provided that "major" and "minor" are taken to refer to height in the hierarchy rather than the size of the changed text.

Subunits only lead to increased recompilation costs when the parent unit is modified. When a parent unit acts as a container, it is modified only when a design change causes the interface of one of its subunits to be rewritten. In a carefully designed system, such changes should be rare, and the impact of subunits on recompilation costs should be minimal.

3. EPILOGUE

The logical dependencies among the subunits of Utility are much more restricted than the modular structure of Utility (as reflected in Figure 3) indicates. Potentially, every subunit could logically depend on every other unit. (Here, logical dependency of one package on another means that either the specification or the body of the first package refers to the specification of the second. Thus two packages can be logically dependent on each other.) In fact, there are only five logical dependencies: Line_Table_Ops and RI_Ops are logically dependent on Control, Message_Ops and Segment_Ops are logically dependent on Manage_Storage, and Message_Ops is logically dependent on Segment_Ops.

A more descriptive modular structure can be obtained by rewriting Utility to contain two subunits, say Utility_1 and Utility_2, each specified in the visible part of Utility. Utility_1 would contain Line_Table_Ops and RI_Ops, specified in the visible part of Utility_1, and Control, specified in the declarative part of Utility_1's body. Utility_2 would contain Segment_Ops and Message_Ops, specified in the visible part of Utility_2, and Manage_Storage, specified in the declarative part of Utility_2's body. (In practice we would use more meaningful names, of course.)

Although this division leads to a more precise description of the logical structure of the message switch, it has drawbacks. Textual depth of nesting is increased, since the specification of Utility must contain the specification of Utility_1 and the specification of Utility_1 must contain the specification of Line_Table_Ops, for instance. This nesting does not even serve to hide inner entities from the outside world, since nesting is within visible parts of package specifications. As indicated in the case study on reducing depth of nesting, deep textual nesting can be detrimental to readability. It is not clear that the benefits of decomposing Utility into subunits Utility_1 and Utility_2 are worth the cost.

A possibly more attractive alternative is to discard the container Utility and make Utility_1 and Utility_2 library units. Neither Utility_1 nor Utility_2 would be given a context specification naming the other, and this would document their logical independence of each other. Besides providing more information to the reader, this would reduce potential recompilation costs. (As sibling subunits of Utility, Utility_1 and Utility_2 have access to each other, and an implementation may require recompilation of the Utility_1 subunit when the Utility_2 specification is changed.) The drawback of this approach is that it makes the interconnections between library units more complex than the interconnections depicted in Figure 3. There would be fourteen connections between library units instead of ten.

Objections have been raised to programming styles based on nesting, but it is important to put these objections in perspective. Physical nesting makes it difficult to read nested structures spread over several pages and keep track of the opening and closing of each structure. Body stubs eliminate physical nesting, but preserve the logical properties of a nested unit. The logical properties of nesting can be misused, making a program hard to understand, but they can also be used in a way that contributes to readability. In particular, subunits that reference global variables are generally hard to understand, but subunits can be written without reference to global variables. In fact, nesting can be used to reduce the visibility of a name.

The same can be said for the use of library units. Arbitrarily complex, non-tree-like system structures can be built by combining library units. Such systems are to programming in the large what unstructured "spaghetti code" is to programming in the small. However, library units can also be connected in a disciplined way to produce well-structured, perhaps even purely tree-structured, programs.

Using only library units does not avoid the problems associated with global data. These problems — indiscriminate access to data, hidden side effects, complex interfaces that are hard to describe abstractly, and subtle interdependencies of remote sections of code — can also result when several library units make use of data declared in a package. Technically, data declared in the visible part of a library package is global, since the Ada Language Reference Manual [DoD83] stipulates that library units are, in effect, declared in the package Standard. The real villain in the abuse of global data is not nested scoping, but the interaction of disjoint program components without an explicit interface. The real solution to these problems is not to ban subunits, but to require that program components communicate explicitly, using subprograms calls and parameters.

A nested program structure appears on the surface to enforce tree-like connections between modules, but in fact it does not. Conversely, context clauses can be used to enforce a strict tree-like structure, but they do not appear to do so! This is because the program text does not contain a concise indication of where a library unit is used. This information is present in the program text, but it can only be obtained by culling through all the context clauses in the system and collating the information thus obtained.

Summary

A program built out of separate subunits contains important cues to help a reader understand the program. Unlike a program built out of library units, a program built out of separate subunits documents where each module is used. The "separate" clause of a subunit makes it easy to find the declarations of names declared in other subunits, and decomposition into subunits can help keep the name space sparse by

allowing context specifications to refer to a more limited scope.

Library units can be combined in arbitrary ways to produce a complex module structure with many interfaces among modules. Subunits necessarily form a hierarchy of modules, with relatively few interfaces. In a hierarchy, it is more evident that certain modules are implementations of others, and it is easier to learn how a program works by top-down examination.

While nesting provides a more explicit picture of program structure than context clauses provide, it provides a less precise picture. Nesting does not convey tree structure precisely, because it allows a module to refer to its siblings, its ancestors, and the siblings of its ancestors.

Nesting encourages the use of global variables, which complicate interfaces between modules and make it hard to draw abstractions. However, nesting does not require the use of global data, and exclusive use of library units does not prevent it. In fact, it is possible to use outer units not as a place to declare variables to be used by several inner units, but simply as a container to prevent the inner units from being seen throughout the entire program. Rather than providing indiscriminate access to declared entities, this use of nesting actually enforces disciplined access to declared entities. Outer units used strictly as containers act much like Mesa configurations, to document the modular structure of a system and carefully control interfaces.

Since system design is rarely purely hierarchical, it is often appropriate to combine the use of library units and separate subunits. Hierarchical portions of the system can be written as logically nested subunits ultimately enclosed in library units, and the intermodule connections that violate strict tree-structure can be achieved by connecting these library units with context clauses. This can lead to a program that conveys a simple and elegant overview of its modular structure, like Figure 3.

Use of subunits in place of library units has the potential of increasing recompilation costs, because it is difficult to determine that sibling subunits are not logically dependent on each other. However, massive recompilation usually results when major design decisions are changed. Carefully designed modules and interfaces can help minimize the cost of recompilation. Programs that are hierarchical -- as programs built out of subunits must be -- are likely to require less recompilation than non-hierarchical programs.

This case study has pointed out the need for two tools that would ameliorate some of the problems resulting from various modular designs. One is a tool to explain the origin of all identifiers appearing in a compilation unit but declared elsewhere. These include identifiers made directly visible by context clauses and identifiers inherited by a

subunit from its parent unit. The other tool, to make it possible to understand how a global variable is manipulated, would produce a list of all places in the program that modified or referenced the global variable. The functions of both these tools can be met by a familiar cross-reference generator, provided that it has the ability to gather information from several compilation units at once.

THIS PAGE INTENTIONALLY LEFT BLANK

Section 3

CONCLUSIONS

3.1. Ada Lifecycle Design Methodology

Ada was designed to facilitate the use of modern software engineering techniques. Issues like modularity, top-down design, data abstraction, information hiding, fault-tolerant programming, division of labor, and rigorous definition of interfaces played a central role in the definition of the language. The novel features of Ada, in turn, are changing the way we look at software engineering. Much of the traditional wisdom about the software life cycle must be re-evaluated in light of Ada. This section describes the characteristics of a complete life cycle design methodology promoting effective and efficient use of Ada. The intent is not to propose a drastically new methodology optimized for Ada, but rather to analyze the impact of Ada on a fairly classical view of the software life cycle.

Definitions of the software life cycle vary in detail, but most recognize the following phases:

- Problem analysis. This is the definition of the problem to be solved. The solution may involve a combination of manual procedures, non-computerized equipment, computer hardware, and software. The analysis does not presume anything about the solution to be chosen. The output of this phase is a written statement of the problem to be solved and the constraints restricting possible solutions.
- Requirements definition. This is a division of the solution into computer hardware, software, and non-computer components, along with a statement of the capabilities each component is to provide. This phase produces a document defining completely and rigorously the input-output behavior and performance requirements of the software component, along with similar documents for the other components. In what follows, we are concerned with software requirements.
- High-level design. This is the formulation of a plan for fulfilling the software requirements. The major modules of the software and the functions they perform are identified. The interaction among the modules, including the data abstractions with which they communicate, is defined. This results in a module-by-module "blueprint" of the program.
- Low-level design. The high-level design is refined to determine how each component of the software fulfills its function. This is an iterative process in which the functional specification of a component is taken as input and the internal design of the component is produced as output. The internal design includes the functional specifications of lower-level components, and these serve as the input to subsequent iterations. The outcome is a definition of all the

algorithms and data structures.

- Coding. The ultimate product of low-level design is a complete software description at a level easily expressible in a programming language. Coding is the translation of this description into the programming language. This may involve further stepwise refinement, as in low-level design. Thus the dividing line between low-level design and coding is not sharp. The product of coding is executable program text.
- Unit testing. This is the compilation, testing, and debugging of individual software components. It results in code known to be legal and free from gross local bugs.
- Integration testing. This is the testing of individually validated software components in conjunction with each other, to be sure that they interact as specified in the design. Integration testing leads to a complete functioning program.
- Acceptance testing. This is the testing of the entire system, including its non-software components, to validate that it solves the problem identified in the problem analysis stage.
- Maintenance. Maintenance is the changing of an existing program to fix errors that are discovered once the program is in use, to improve performance, to accommodate changes in the environment, or to add new functions. Maintenance involves feasibility analysis, examination of existing code to determine the appropriate place to make a change and its expected impacts coding of the change, and revalidation of the modified program. Over the life of a program, maintenance typically accounts for two-thirds of the cost of the software.

The phases of the software life cycle do not follow one another in strict sequence. Rather, they are interleaved. Questions arising during requirements definition may lead to further problem analysis, and high-level design may reveal the need for further requirements definition. Low-level design may reveal problems in implementing the high-level design. Coding and unit testing generally occur together and often lead to reconsideration of low-level design decisions. Errors in high-level design may be discovered during integration testing, and acceptance testing may reveal flaws in problem analysis or requirements definition. Maintenance involves its own design phase, though one that is highly constrained by the existing program. In some cases, maintenance may involve redesign and recoding of a major software component.

This basic framework remains as valid with Ada as with any other programming language. However, use of Ada can be expected to change both the nature of these phases and the relative impact of each phase in the complete life cycle. The remainder of this section discusses the

implications of Ada for each phase.

3.1.1 Problem Analysis

At first glance, it would appear that the use of Ada should not influence problem analysis or requirements definition. These phases are both concerned with the definition of the problem to be solved (first by the system as a whole, and then by the software component of the system), not with the definition of the solution. Thus problem analysis and requirements definition are both fundamentally language-independent. Even the designation of a programming language at this stage is sometimes regarded as over-specification.

Still, an important mission of Ada is to facilitate the production of reusable software components. This may lead us to view the product of a system development in a new light. Traditionally, we have viewed a working system (including documentation) as the only product of system development. When Ada is being used, it is sometimes more appropriate to view the system as being the primary product, and software components reusable in future systems as secondary products. In this light, the specification of a programming language may be appropriate as early as the problem analysis phase. The full context of problem analysis includes not only the current problem on which attention is focused, but the anticipated need to solve other similar problems in the future.

A more difficult question is how to specify the nature of the reusable software components in a way that does not constrain requirements analysis and design. If the problem definition states that a generic Fourier transform function should be produced, the requirements definition is precluded from allocating the computation of Fourier transforms to microcoded chips. All that can be properly stipulated at the problem definition phase is that any software built to perform certain specified functions is to be delivered, in the form of source code, as a reusable component.

3.1.2 Requirements Definition

At the requirements definition phase, once it is determined which functions will actually be implemented in software, it is appropriate to describe the characteristics that make a particular software component reusable. This description consists of a list of variations that the component must be able to accommodate. It does not stipulate whether the variations are to be accommodated by generic parameters, subprogram parameters, input data, or some other mechanism.

Previous investigations [Sof82] demonstrated the use of Ada as a language for stating requirements. The danger of this approach is that it is an invitation to code prematurely. Furthermore, the use of Ada to state requirements precludes the use of graphics. The designers observed during the project described in [Sof82] found the use of graphics extremely helpful.

3.1.3 High-Level Design

Traditionally, the product of software design has been the specification of subprograms and their calling hierarchy. The product of Ada software design should be the package structure of the program. This consists of both the package specifications and context clauses showing how they relate. If major components in a high-level design (for example, a data base management system) are to be procured off the shelf rather than developed independently, this should be stated in the design.

Package structure conveys certain information not conveyed by a calling hierarchy. For example, case study 2.4.1, "The Use of Exceptions," explains that fault tolerance must be carefully built into a system at the design stage. The high level design must stipulate how to handle exceptions arising from hardware failures or unanticipated software errors. The existence of system-wide exceptions is conveyed by their declaration in package specifications. The protocols for raising and handling an exception must be provided separately, perhaps as comments describing the subprograms and tasks that raise and handle the exception.

A calling hierarchy does not distinguish between subprograms providing fundamental services and subprograms used to implement those services. Package structure describes the division of a program into modules providing sets of facilities to other modules. Subprograms are described only if they are among the facilities provided by a package. Subprograms used only to implement a package's facilities ought not to be considered during high-level design. Indeed, such subprograms are described in package bodies, not package specifications, so they are not reflected in the package structure.

In the work leading to [Sof82] there were indications that a calling hierarchy provides a limited, and perhaps distorted, "big picture." It was hypothesized then that a similar chart, showing the with relationships, might be more useful. Examples of such charts can be found in case study 2.5.4, "Library Units Versus Subunits."

On the basis of current experience it is not sufficiently clear to what extent the tasking structure should be determined at the high-level design stage. Conceptually, most of the major packages will operate concurrently, or at least largely independently. A purist might argue

that the fact that a package may have one or more tasks is all that counts. In practice, however, it may be necessary to make certain decisions. For example, in implementing an abstract data type or object, it is important to know whether the object must be guarded (by a monitor task) against concurrent access. Furthermore, some estimate of the real-time performance will be necessary at this stage, and that may be infeasible in the absence of information about the tasking structure.

3.1.4 Low-Level Design

Low-level design traditionally involves stepwise refinement of the high-level design until the design is sufficiently detailed to begin coding. Since an Ada high-level design consists largely of compilable Ada code, it is possible to perform this stepwise refinement in Ada. Then the distinction between low-level design and coding is blurred.

The main advantage usually cited for hierarchical program structure is based on information hiding. Hierarchically-structured programs have loosely-coupled modules, i.e., modules that interact in limited ways. Stepwise refinement should proceed in such a way as to maximize information hiding. When a new data abstraction is required, it should be implemented as a private type, in a package whose sole purpose is the implementation of the data abstraction. Examples of this approach are found in case studies 2.2.2, "Implementation of Set Types", 2.2.5, "Recursive Type Definitions", 2.5.1, "Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output", and 2.5.2, "Information Hiding".

A designer begins stepwise refinement by formulating bodies for the packages produced during high-level design. During this process, he will determine the need for lower-level facilities, and write the specifications of packages providing those facilities. Subsequent refinement steps involve the formulation of a body for a previously written package specification, possibly yielding new package specifications.

If a new exception is introduced during a refinement step, the ultimate refinement must completely account for the handling of the exception. Exceptions not planned for in the high-level design must remain invisible to other parts of the program hierarchy. Case study 2.4.1, "The Use of Exceptions", elaborates on this point.

It is not clear whether a refinement step should be able to introduce new tasks, or whether all tasks in a program must be accounted for in the high-level design. A related issue is the ease with which tasks, introduced in the design stage as a powerful abstraction mechanism, can be eliminated in the coding stage by mechanical transformation to produce an efficient implementation. Stylistically, a task is only a linguistic detail, and tasks should be freely introduced

at any stage; demanding that all tasks be identified up front is conceptually as restrictive as demanding that all arrays be so defined. Pragmatically, it is possible that excessive dynamic spawning of tasks will make the performance too hard to estimate.

This top-down scenario is too simple to cover all situations, however. In particular, system design should take into account the availability of general purpose, reusable software components (including those whose development was mandated in the requirements definition of the current project). This requires a "software literature search" before low-level design begins to identify what is available, and constant alertness during low-level design to the possibility that the required component has already been developed. A related issue is the problem of identifying operations that will be required at several places in a program. A strictly hierarchical approach would lead to duplication of coding effort. Even the most zealous supporters of hierarchical system structure acknowledge that certain operations may be regarded as generally applicable primitives and exempted from strict adherence to tree structure. Finding these primitives is a more difficult challenge in Ada because generic parameters provide greater opportunities for commonality. Often, two apparently independent program units can be shown to be instantiations of the same hypothetical generic unit. Designers talented in generalization should participate in regular reviews of the evolving low-level design to identify opportunities for avoiding duplicate coding.

Primitive facilities to be used several places in a program should be coded as library units and made available through context clauses. The hierarchy of program units developed by stepwise refinement can be realized either by library units and context clauses or by subunits and body stubs. The complex issues to be considered in choosing one of these approaches are discussed in case study 2.5.4, "Library Units Versus Subunits". (This case study also discusses hierarchical program structure and primitive operations.)

Case study 2.1.1, "Guidelines for the Selection of Identifiers," explains the importance of spelling each English word used in identifiers in a consistent way throughout a large system. It follows that the vocabulary to be used for building identifiers must be agreed upon during the design stage, before coding begins. A lexicon should be part of the low-level design document.

3.1.5 Coding

As suggested earlier, the coding of Ada programs can be largely accomplished during the low-level design stage. This is primarily because of the powerful abstraction mechanisms in Ada, which allow high-level operations to be expressed directly in the language rather than encoded in terms of primitive instructions. It is reasonable for a

design to produce only pseudo-code in cases where coding is rote and tedious. Examples are filling a large array aggregate with values specified elsewhere, or implementing all the operations for a private type once the data representation and abstract operations have been completely specified. In such cases, coding remains a distinct step.

It is important that programmers be aware of Ada coding paradigms so that they can make use of the full expressive power of the language. Sections 2.2 and 2.3 contain case studies dealing with such paradigms. It is only because Ada, used to its fullest potential, can express algorithms at a very high level that there is no clear distinction between low-level design and coding in Ada.

A danger inherent in any block structured language is that programs will become too deeply nested. This can make programs difficult to understand. Ada provides mechanisms for reducing the depth of nesting. The problem and some solutions are discussed in case study 2.5.3, "Reducing Depth of Nesting".

3.1.6 Unit Testing

Unit testing should be somewhat easier in Ada than it has been in previous languages, because of Ada's compile-time and runtime checks. The strong typing rules will allow some of the most frequent sources of subtle errors, errors that manifest themselves only at runtime in other languages, to be caught at compile time. Constraint checking at runtime can alert a programmer to the fact that his beliefs about the program's behavior are inaccurate. Furthermore, they cause incorrect programs to terminate in predictable and well-defined ways. Therefore, constraint checks should never be disabled, except in a limited way as a desperate optimization step in a thoroughly integration-tested program, after all other remedies to a performance problem have been exhausted.

Just as Ada eliminates many old sources of error, it may introduce some new ones. Visibility rules and the ability to overload subprograms, entries, and enumeration literals may lead to new and interesting bugs. So far, however, we have no experience to document.

Information hiding, which is very beneficial during most of the life cycle, can present certain difficulties during unit testing. The measures taken to hide the internal representation of data can make it difficult to produce diagnostic output describing that data. It may be wise for each package implementing a private type to include two diagnostic output procedures -- one to dump the internal representation of the data and one to print the data in an abstract form. The first routine will be used to debug the package implementing the data abstraction. The second routine will be used once the data abstraction appears to be debugged, and in writing the dump routines for larger data types in which the private type is used as a component type. These

routines may be "commented out" (each line preceded by "---") in the production version of the system, but they should be made available, at least in this form, to maintenance programmers.

Dumping the internal representation of a data type defined using access types presents special problems. It may be necessary to print the address of an allocated object together with its contents to reveal sharing patterns. If the access value links may be circular, special measures must be taken to avoid infinite loops.

3.1.7 Integration Testing

Integration testing is usually a point at which serious inconsistencies are discovered in program design. Errors requiring expensive corrections manifest themselves in what appeared to be a well-working program. Typical sources of integration test failure are mismatches between types and parameters in separately compiled units.

Ada's cross-compilation-unit consistency checking should greatly reduce the trauma of integration testing. Compilation-order rules require that a package specification be compiled before either the body implementing the package or any unit using the package. If any compilation unit is syntactically inconsistent with the interface declared in the package specification, the error will be caught at compile time. If stepwise refinement proceeds through successive creation of package bodies and specification of lower-level packages to implement those bodies, this consistency checking can be performed during the design stage, even before unit testing!

Integration testing does not disappear totally, however. Ada compilers cannot check that all users of a package correctly understand the semantics of a package's facilities, just the correct syntax for using them. For example, correct use of the package `Text_IO` requires a file other than `Standard_Output` to be open before it is written to, but violation of this condition is not a syntax error. While strong typing makes a larger portion of semantic errors into syntax errors, some semantic errors are likely to remain undetected. Furthermore, errors involving real-time interactions between software components and errors in the hardware-software interface still will not be detected until the components of the system have been assembled.

3.1.8 Acceptance Testing

The use of Ada should not have much impact on acceptance testing, except insofar as it increases the likelihood of a timely delivery of a system fulfilling its requirements. However, acceptance test failures are usually traceable to problems in analysis or requirements

definitions. The use of Ada is not expected to make such problems any more or less likely.

3.1.9 Maintenance

Well-designed large Ada programs will be easier to maintain than comparably-sized programs written in earlier programming languages. The package structure will strongly and clearly define the modular decomposition of the system and the interfaces between the modules. It will be easy to locate the part of a program performing a given function, and the "ripple effect" of changes to that program will be minimized. The walls erected in the original design to facilitate information hiding are not easily breached during maintenance, increasing the likelihood that the structure of the maintained program will remain coherent.

The price of Ada's strong inter-compilation-unit consistency checking is a set of stringent rules about order of recompilation. These rules protect the maintenance programmer from inconsistencies in the same way that they protect the original developers, and assure that object modules are as current as source modules. Nonetheless, they can be burdensome, and recompilation costs resulting from a simple change can be awesome. A careful original design can anticipate and minimize recompilation costs. This issue is addressed in case study 2.2.3, "Constant Array Declarations".

3.2 Research Topics Addressed

This section will discuss those research topics identified in [Sof82] which have been addressed in this report. Each topic is reviewed and cross-references to the case studies in this report are given.

3.2.1 Information Hiding - Data Abstraction

In [Sof82, A.2.4] it was noted that paradigms were needed to illustrate the use of packages in expressing different levels of abstraction. Information hiding is an important part of the design process. The relationship between the concepts of data abstraction and of separation of concerns and the Ada features which enforce them, namely packages and private types, are explored in several case studies, listed below:

2.2.2 Implementation of Set Types

- data abstraction

2.2.5 Recursive Type Definitions

- use of private types and packages to hide information

2.5.1 Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output

- preserving implementation freedom
- assumptions about underlying hardware and software

2.5.2 Information Hiding

- use of packages to provide a logical interface that hides the physical interface

2.5.4 Library Units Versus Subunits

- levels of abstraction
- global data

3.2.2 Postponing Design Details

The discussion in [Sof82, A.2.10] pointed out that a number of Ada features enable the designer to postpone design details of a system component until a clear, coherent picture of it has emerged. By specifying a logical interface between this and other system components, the design of these other parts can continue. These features include but are not limited to generics, private types, subunits, parameters and package specifications. Various facets of this issue are expanded upon in the following case studies:

2.1.2 Implementation of Set Types

- use of generics

2.1.3 Constant Array Declarations

- package specifications and bodies
- deferring commitment to the size of an array

2.5.1 Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output

- designing a package interface without constraining the implementation
- avoiding dependencies on underlying hardware and software

2.5.2 Information Hiding

- separation of logical from physical interfaces
- use of private types

3.2.3 Guidelines on Packages

In [Sof82, A.3.2] it was suggested that guidelines be developed to aid the designer in writing packages. Information hiding, the grouping of logically-related entities, modularity, reusability, size, and intuition all affect package design. Several case studies in the current report discuss these different aspects, and they are listed below:

2.2.2 Implementation of Set Types

- use of packages to provide a data abstraction
- generic packages
- use of private types in a package

2.2.5 Recursive Type Definitions

- use of private types in packages

2.4.1 The Use of Exceptions

- exception declarations in packages

2.5.1 Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output

- reusability and portability
- general purpose functionality

2.5.2 Information Hiding

- use of private types in packages
- encapsulation of a hardware-software interface
- designer's intuition

2.5.4 Library Units Versus Subunits

- package size
- designer's intuition
- modularity and functional cohesion
- information hiding
- use of nesting

3.2.4 Tutorial on Exceptions

[Sof82, A.3.3] identified the need to develop a methodology for the proper usage of exceptions. They are a powerful design tool, discussed in the following case studies:

2.4.1 The Use of Exceptions

- implementation of unusual control structures using exceptions
- use of exceptions to "idiot-proof" a reusable software component
- use of exceptions to respond to invalid input data
- use of exceptions to recover from hardware and software failures
- disciplines for handling and propagating exceptions
- roles of the designer and coder in determining use of exceptions

2.5.1 Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output

- providing information useful for recovering from exceptional situations

3.2.5 Dot Notation, Naming and Readability

In [Sof82, A.4.1] the effect of dot notation and naming conventions were briefly discussed. Some guidelines are suggested in the following case studies:

2.1.1 Guidelines for the Selection of Identifiers

- choosing descriptive identifiers

2.3.4 Use of Block Statements for Local Renaming

- use of renaming declarations to avoid complex names

3.2.6 Standard Coding Style

[Sof82, A.4.2] recommended that a set of standard coding conventions be adopted. Such a style guide would address questions of code format, naming conventions, comments, and so forth. All of the case studies in this report attempt to follow a consistent style. Furthermore, the case study listed below explicitly presents ideas on naming conventions for identifiers:

2.1.1 Guidelines for the Selection of Identifiers

- naming conventions

2.4.1 The Use of Exceptions

- rules forbidding a programmer from interfering with the high-level exception-handling design of a program

3.2.7 Simple Coding Paradigms

In [Sof82, A.4.4] it was observed that case studies recording simple examples of Ada-like usage were needed. This volume contains several such case studies:

2.2.1 Discrete Types

- if statements
- case statements
- arrays indexed by enumeration type

2.2.2 Implementation of Set Types

- Boolean arrays
- unconstrained array types
- positional array aggregates
- generic instantiations

2.2.3 Constant Array Declarations

- positional and named array aggregates
- constants
- unconstrained array types
- index constraints
- package initialization

2.3.1 Use of Slices

- slices of arrays
- subtypes
- for loops

2.3.2 Short Circuit Control Forms

- if statements
- logical operators
- while loops

2.3.3 Loop Statements

- different kinds of iteration constructs
- exit statements
- subtypes

2.3.4 Use of Block Statements for Local Renaming

- block statements
- renaming declarations

3.2.8 Handling Impossible States

In [Sof82, A.4.5] it was noted that research was needed on methods to identify and handle unexpected or presumably impossible system states. Part of the problem was perceived to be that of educating the project staff to follow standard practices for the reporting and

handling of such errors so that they are neither overlooked nor deliberately hidden. The following case studies address these issues:

2.4.1 The Use of Exceptions

- raising, handling, and propagation of an exception specifically designated to indicate an unanticipated software error

2.5.1 Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output

- limitations of exceptions in Ada

3.2.9 Hardware Error Detection

In [Sof82, A.5.1], the question was raised as to how hardware errors should be communicated to the software. The following case studies discuss this issue:

2.4.1 The Use of Exceptions

- mechanism for a task driving a device to detect a hardware problem and propagate an exception to users of the device
- examples of measures for responding to hardware malfunctions

2.5.1 Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output

- the inflexibility of the predefined Ada exception `Use_Error`

3.2.10 Reusable Software Modules

[Sof82, A.5.2] expressed the need to find ways to identify and generate reusable software. Several case studies discuss some of the aspects of specifying reusable software:

2.2.2 Implementation of Set Types

- an example of a generic package providing a fundamental data abstraction tool

2.4.1 The Use of Exceptions

- allowing software components to react in well-defined ways when improperly invoked

2.5.1 Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output

- Text_IO as an illustration of the difficulties in designing general-purpose software components

3.3 Areas for Future Research

This report has answered some questions and posed others. The development of an Ada methodology is an ongoing process, in which experience plays a crucial role. Since we have just barely begun to accumulate experience in the use of Ada, we still have much to learn.

This section briefly describes several areas in which further work is needed. This further work ranges from case studies demonstrating widely misunderstood language rules to open research questions. The topics have been divided into five areas — design issues, data abstraction issues, additional naming conventions, additional coding paradigms, and operational issues.

3.3.1 Design Issues

3.3.1.1 Guidelines for Preserving Implementation Freedom When Designing Packages

An important purpose of Ada packages is to distinguish between the facilities provided by a module and the implementation of those facilities. Especially when they declare private types, packages can give the implementor considerable freedom to implement facilities in ways unforeseen by the person who wrote the package specification. However, case study 2.5.1, "Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output," reveals that the specifications of the predefined Ada I/O packages make restrictive assumptions about the implementation and the environment. This establishes the need for guidelines to be followed by package specification designers to maximize the options available to package body writers.

3.3.1.2 Recognizing the Opportunity for Generic Solutions

Section 3.1, "Ada Life Cycle Design Methodology," addresses some of the problems in realizing the Ada goal of general, reusable software. One of the problems is recognizing that two or more apparently distinct problems can be solved with a single generic unit. Programmers and designers must be trained in the art of algebraic generalization. A good first step is the formulation of case studies illustrating how several different problems can be generalized to the point that they share one generic solution.

3.3.1.3 Fault-tolerant Program Design

Case study 2.4.1, "The Use of Exceptions," gave an example of the use of exceptions to respond to a hardware anomaly and an example of the use of exceptions to respond to and isolate the effects of an unexpected

software error. Future research is required to develop additional examples and extrapolate general principles of fault-tolerant program design. The proper response to hardware and software failure remains one of the most difficult problems facing the software designer.

3.3.1.4 Introduction of Tasks During Stepwise Refinement of a Design

Section 3.1, "Ada Life Cycle Design Methodology," left open the question of whether new tasks may be introduced during evolution of a low-level design or whether all tasks must be planned for in the high-level design. This is primarily a performance issue, and only experience using real implementations will provide the answer.

3.3.1.5 Alternative I/O Packages

Case study 2.5.1, "Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output," points out several deficiencies in the design of the Ada I/O packages. The redesign of the packages to meet these objections would be a worthwhile exercise.

3.3.2 Data Abstraction Issues

3.3.2.1 Specific versus General Types

Case study 2.2.4, "Record types," illustrated a situation in which a data structure could be implemented using either several record types or a single type with several variants. This is just one aspect of a larger problem — should data types be specific or general? For example to implement a tree that can have four different kinds of nodes, each allowed to have different kinds of subtrees, should a programmer declare four different data types, or one data type general enough to represent all four kinds of nodes? Specific data types make it easier to detect logically inconsistent data structures, usually at compile time, because violations of the data structure's consistency rules become violations of Ada's type rules. General data structures make it easy to write general and powerful data manipulation routines, such as a tree traversal subprogram. A case study illustrating the issue and discussing the tradeoffs would be extremely beneficial.

3.3.2.2 Abstract Data Types Implemented as Access Types

The message switch program referred to in many of the case studies contained several package specifications with a declaration of a private record type plus a visible declaration of an access value designating the record type. The operations provided by these packages dealt only with the access type. One's first reaction is that the access type should have been declared private and the record type declaration should have been completely hidden in the private part of the package. However, the sharing of data resulting from an access value assignment

3.3.3 Additional Naming Conventions

Case study 2.1.1, "Guidelines for the Selection of Identifiers," provides an example of conventions that can be used to govern the choice of identifiers in Ada programs. Experience should lead to the augmentation and refinement of those conventions, as well as the development of new conventions.

3.3.4 Additional Coding Paradigms

The case studies in Section 2.3 illustrate some uses of distinctive Ada features -- slices, short circuit control forms, the three forms of loops, and block statements. There are many uses of these features that were not illustrated, and other features that were not illustrated at all. Additional coding paradigms illustrating appropriate use of Ada would be important tools in teaching how one should use Ada rather than how one may use Ada.

Specific coding issues that deserve further treatment are discussed below.

3.3.4.1 When to Use use With with

Case studies 2.1.1, "Guidelines for the Selection of Identifiers," and 2.5.4, "Library Units Versus Subunits," both argue that overuse of use clauses introduce a multitude of identifiers that are declared in some other compilation unit. Use clauses make it possible to refer to those identifiers without even an expanded name indicating the origin of the identifier.

Most of the time, renaming declarations are a more appropriate way to provide a succinct local notation for names declared elsewhere. A renaming declaration provides information about the meaning of an imported entity and also indicates where the entity comes from. Furthermore, renaming declarations can document which of the entities provided by an imported package are actually used locally.

Banning use clauses entirely would be extreme. Rather, we should investigate where use clauses are more appropriate and when renaming declarations are more appropriate.

3.3.4.2 Justifiable Uses of the Goto Statement

It has been generally acknowledged for a number of years that goto statements tend to make programs hard to follow. Ada provides a number of features that make most conventional uses of the goto statement unnecessary. Nonetheless, Ada also provides a goto statement. Thus we

would then be hidden from the package user. Making the access type limited private would rule out such assignments. However, as case study 2.5.1, "Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output" points out, even limited private types may not completely hide the implementation as an access type. A new case study should be devoted to this specific issue.

3.3.2.3 Non-Access "Pointer" Types

Case study 2.2.5, "Recursive Type Definitions," demonstrates that a direct access file element key acts as a "pointer," indirectly identifying data. The case study suggests that there may be other cases of non-access-type data acting in this way. If so, there could be other recursive data structures that cannot be modelled using Ada recursive types. The identification of other forms of "pointer data" is left as a topic for further research.

3.3.2.4 Data Structures Involving External Files

Case study 2.2.5, "Recursive Type Definitions," points out that a direct access file in which each file element contains a link to another may be viewed as a recursive data structure. In general, the design of Ada pays much attention to data structures residing in primary storage, but little to data structures residing wholly or partly in files. The role of files in data modelling deserves closer scrutiny.

In particular, Ada requires all elements in a sequential or direct-access file to belong to the same type. To build a heterogeneous file, one must declare a record type with variants and use different variants for different file elements. There is a need for a case study illustrating this approach and examining its implications.

3.3.2.5 Use of Character Types

Character types -- enumeration types some of whose literals are character literals -- are a widely misunderstood feature of Ada. Common mistakes are regarding identically-written character literals of different types as somehow related and expecting instantiations of Enumeration_IO with character types to work the same way that Text_IO works for the predefined type Character (i.e., without reading or writing single-quote delimiters). In fact, character types are not very useful except to define alternative character sets like EBCDIC and FIELDATA in Ada. A case study should be written illustrating correct and incorrect, appropriate and inappropriate uses of character types.

must determine when the use of a goto statement in an Ada program is justified, if ever.

Case study 2.4.1, "The Use of Exceptions," suggests that exceptions can be used to simulate Zahn's construct [Zah74]. Experience could well reveal, however, that goto statements implementing Zahn's construct are more easily understood than exceptions. Case study 2.5.3, "Reducing Depth of Nesting," suggests the use of goto statements to avoid the need to nest conditional entry calls on the grounds that a disciplined use of the goto statement is easier to understand than a deeply-nested structure. Research is required to identify other situations in which a goto statement may be the most easily understood alternative available to the Ada programmer.

To place these results in perspective, it would be wise to accompany them with a case study on unjustified uses of the goto statement. This case study could take the form of a series of Ada programs written in the style of FORTRAN IV, accompanied by equivalent programs that do not use goto statements. Such a case study would be a valuable tool in retraining FORTRAN and assembly language programmers to use Ada.

3.3.4.3 Use of Procedures for Local Renaming

Case study 2.3.4, "Use of Block Statements for Local Renaming," discussed the use of block statements containing renaming declarations to avoid redundant evaluations of names. Procedures can be used in the same way. The statements containing multiple occurrences of the same complex name can be embedded in a procedure body, with all occurrences of a given name replaced by the same formal parameter. These two approaches should be compared to determine which is easier to follow.

3.3.4.4 Replication of Tasks

The message switch program studied to produce this report contained two nearly identical task bodies. Ada provides two mechanisms for replicating tasks -- task types and tasks declared in generic package specifications. Both of these approaches are superior to duplicate source code in terms of maintainability and size of the object program. A case study could illustrate this issue.

3.3.4.5 Subprograms With Variable Types of Parameters

Case study 2.2.2, "Implementation of Set Types," illustrates the use of a parameter belonging to an unconstrained array type to simulate a subprogram with an unlimited number of parameters of the same type. A positional aggregate of any length could be used as an actual parameter.

The program on which the case study was based used a procedure with a parameter belonging to a record type with discriminants in a similar way. There were a few different combinations of data that had to be

supplied on different calls. These corresponded to different variants of the record. Procedure calls in which the actual parameter was a record aggregate for one variant or another had the effect of simulating a procedure that could be called with variable numbers and types of parameters.

Ada provides a more direct way of doing this -- overloading. A case study could compare the two approaches.

3.3.5 Operational Issues

3.3.5.1 Implementation and Optimization Issues

Several case studies point out areas in which a state-of-the-art compiler could produce more efficient code than might be apparent. Case study 2.2.3, "Constant Array Declarations," mentions that use of a discriminant of type Positive as a bound in an index constraint might or might not require a prohibitive amount of space, depending on the implementation. Case study 2.3.2, "Short Circuit Control Forms," explains that most uses of short circuit control forms to reduce execution time are inappropriate, because this low-level optimization can easily be performed by a compiler. Case study 2.3.3, "Loop Statements," shows that loops can often be written most clearly by introducing apparent inefficiencies that can, in practice, usually be removed by an optimizing compiler. Case study 2.3.4, "Use of Block Statements for Local Renaming," discusses the avoidance of redundant evaluation of object names. Case study 2.4.1, "The Use of Exceptions," states that the use of exceptions to simulate Zahn's construct could be inefficient given a naive implementation, but that the code for exceptions that are declared and handled locally can easily be made quite efficient.

There is a need for further identification of areas in which a good implementation can drastically improve the performance of a particular Ada construct. The purpose of this investigation is twofold. First, it will reinforce the notion that Ada programmers ought to program abstractly, without presupposing that certain usages will be inefficient. Second, it will make Ada implementers aware of constructs arising in real Ada programs for which an effort can and should be made to generate highly optimized code.

3.3.5.2 Definition of Standard Functional Subsets for I/O

Case study 2.5.1, "Specifying Interfaces for General Purpose, Portable Software: A Study of Ada Input/Output," warns that the Ada standard [DoD83] may provide too much latitude in the implementation of input and output, allowing each compiler to implement its own functional subset. To preserve portability of programs, it is important to identify standard functional subsets of the Ada I/O facilities. Then

every compiler can be described as fully implementing one of the subsets. The Ada Compiler Validation Capability has implicitly assumed a hierarchy of functional subsets, but an explicit definition would be desirable.

3.3.5.3 Identification of Ada Programming Tools

Case studies 2.2.3, "Constant Array Declarations," and 2.5.4, "Library Units Versus Subunits," both discuss recompilation costs and highlight the desirability of a tool to determine when the actual change made to one compilation unit logically necessitates the recompilation of another compilation unit nominally dependent on it. Case study 2.5.4 also explains the need for a cross-reference tool that looks past compilation unit boundaries, just as the Ada scope rules do.

An important area for future research is the identification of other language-oriented tools that will be useful at various stages of an Ada program's life cycle. A related research topic is implementation strategies for the tools that are identified. Both aspects of this research will have a direct impact on the maturation of Ada Program Support Environments.

THIS PAGE INTENTIONALLY LEFT BLANK

Section 4

REFERENCES

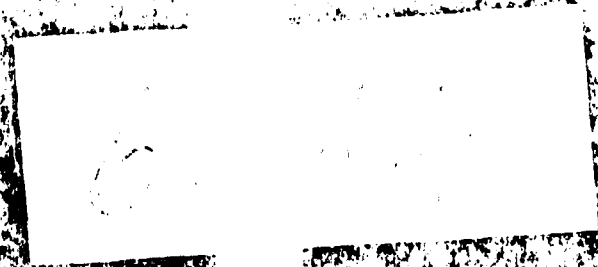
REFERENCES

- [Bel82] Belmont, Peter A. On the access-before-elaboration problem in Ada. Proceedings of the AdaTEC Conference on Ada, Arlington, Virginia, October 1982, 112-119
- [Car82] Carter, Breck. On choosing identifiers. SIGPLAN NOTICES 17, No. 5 (May 1982), 54-59
- [CWW80] Clarke, Lori A., Wileden, Jack C., and Wolf, Alexander L. Nesting in Ada programs is for the birds. Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language, Boston, December 1980, published as SIGPLAN Notices 15, No. 11 (November 1980), 139-145
- [DoD83] Military Standard: Ada Programming Language. ANSI/MIL-STD-1815A, United States Department of Defense, January 22, 1983
- [Gar83] Gardener, Michael R., Brubaker, Nils, Dahlke, Carl, Goodhart, Brian, and Ross, Donald L. Ada programming style. Intellimac, Inc., Rockville, Maryland, January 24, 1983
- [Hoa81] Hoare, C.A.R. Response to letters in "ACM Forum". Communications of the ACM 24, No. 7 (July 1981), 477-478
- [J&W74] Jensen, Kathleen, and Wirth, Niklaus. Pascal User Manual and Report, 2nd ed. Springer-Verlag, New York, 1974
- [Kaf82] Kafura, D., Lee, J. A. N., Lindquist, T., and Probert, T. Validation in Ada program support environments. Report AD-A124 765/9, Virginia Polytechnic Institute and State University, December 1982
- [Knu74] Knuth, Donald E. Structured programming with goto statements. ACM Computing Surveys 6, No. 4 (December 1974), 261-301
- [L&S79] Lauer, Hugh C., and Satterthwaite, Edwin H. The impact of Mesa on system design. Proceedings, 4th International Conference on Software Engineering, Munich, Germany, September 1979, 174-182
- [Sof82] Ada Software Design Methods Formulation Case Studies Report. SofTech, Inc., Waltham, Massachusetts, August 1982
- [Tur80] Turner, Joshua. The structure of modular programs. Communications of the ACM 23, No. 5 (May 1980), 272-277
- [W&S73] Wulf, W., and Shaw, Mary. Global variable considered harmful. SIGPLAN Notices 8, No. 2 (February 1973), 28-34

[Zah74] Zahn, Charles T., Jr. A control statement for natural top-down structured programming. In Programming Symposium: Proceedings, Colloque sur la Programmation, Paris, April 9-11, 1974, B. Robinet, ed. Lecture Notes in Computer Science 19, G. Goos and J. Hartmanis, eds., Springer-Verlag, New York, 1974

END

IMED



DTIC



AD-A140 818

ADA (TRADEMARK) CASE STUDIES III(U) SOFTECH INC WALTHAM
MA JAN 84 DAAB07-83-C-K514

44

UNCLASSIFIED

F/G 9/2

NL

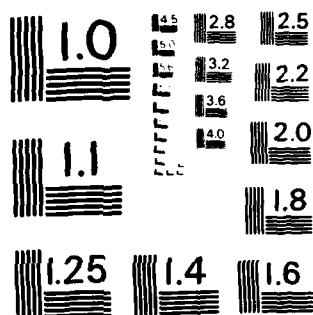


END

12-8-84

1710





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963-A

SUPPLEMENTARY

INFORMATION



DEPARTMENT OF THE ARMY
HEADQUARTERS US ARMY COMMUNICATIONS-ELECTRONICS COMMAND
AND FORT MONMOUTH
FORT MONMOUTH, NEW JERSEY 07703

REPLY TO
ATTENTION OF:

15 OCT 1984

Center for Tactical Computer Systems

Ms. Madeline Crumbacker
Defense Tactical Information Center
Cameron Station
Alexandria, Virginia 22314

Dear Ms. Crumbacker:

As per phone conversation with Ms. Andrea Cappellini, CENTACS on 11 October 1984, a copyright statement has been omitted on documents sent to DTIC and NTIS. Enclosed please find the copyright statement (Encl 1) that must appear in the enclosed list of document (Encl 2). If you have any questions, please contact Ms. Cappellini at 201-544-4280.

Sincerely,

James E. Schell
for JAMES E. SCHELL
Director, CENTACS

REPRODUCED AT GOVERNMENT EXPENSE

AD-A140818

REPRODUCED AT GOVERNMENT EXPENSE

Copyright by SofTech, Inc. 1984. This material may be reproduced by or for the U.S. Government pursuant to the copyright license under DAR clause 7-10A.9 (a) (May 81).









