Structured Hierarchical Ada Presentation Using Pictographs (SHARP) Definition, Application and Automation

WILLIAM E. BYRNE
SUSAN M. BRADSHAW
NEIL A. CRONIN
DAVID E. McDEVITT

Arthur D. Little, Inc.
Program Systems Management Co.
Acorn Park
Cambridge, Massachusetts

September 1986

Prepared For

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
DEPUTY FOR DEVELOPMENT PLANS
HANSCOM AIR FORCE BASE, MASSACHUSETTS 01731

ADA176990

LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

This technical report has been reviewed and is approved for publication.

MARK V. ZIEMBA, 2Lt, USAF
Project Officer, Software
Engineering Tools & Methods

ARTHUR G. DECELLES, Capt, USAF
Program Manager, Computer Resource
Management Technology (PE 64740F)

FOR THE COMMANDER

ROBERT J. KENT
Director
Software Design Center
Deputy for Development Plans
and Support Systems

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 36086-34	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-86-283
6a. NAME OF PERFORMING ORGANIZATION Arthur D. Little, Inc. Program Systems Management Co. — 6b. OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION Hq, Electronic Systems Division (XRSE)
6c. ADDRESS (*City, State, and ZIP Code*) Acorn Park Cambridge, Massachusetts	7b. ADDRESS (*City, State, and ZIP Code*) Hanscom AFB Massachusetts, 01731
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Deputy for Development Plans — 8b. OFFICE SYMBOL (*If applicable*) ESD/XRSE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-84-D-0011
8c. ADDRESS (*City, State, and ZIP Code*) Hanscom AFB Massachusetts, 01731	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. — PROJECT NO. — TASK NO. — WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
Structured Hierarchical Ada Representation Using Pictographs (SHARP) Definition, Application and Automation

12. PERSONAL AUTHOR(S)
William E. Byrne, Susan M. Bradshaw, Neil A. Cronin, David E. McDevitt

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	1986 September	348

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Ada Software Cost/Schedule Estimation Pictographs

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

This paper presents a methodology for representing a large and complex computer program using graphics and Ada-based annotated pseudo code. It describes the application of the graphical representation, referred to as Structured Hierarchical Ada Representation using Pictographs (SHARP), in the design and test of computer programs, and presents a concept of operation for generating the graphics in a computer aided manner. The resulting tool is considered important, since design and test costs account for over 60% of software development costs. The tool also applies to software maintenance, which typically exceeds the original development cost by more than 50%.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL M.V. Ziemba, 2d Lt, USAF	22b. TELEPHONE (*Include Area Code*) (6170 377- 2656 — 22c. OFFICE SYMBOL ESD/XRSE

ACKNOWLEDGEMENT

The definition of SHARP, and the investigation of its application and automation, has been sponsored by the Computer Software Systems Program Office, Software Design Center (XRSE), Electronic Systems Division (ESD), United States Air Force Systems Command, Hanscom AFB, Massachusetts 02173. Funding for the effort has been provided by the Air Force Computer Resource Management Technology Program, PE 64740F, Project 2526 - Software Engineering Tools & Methods.

Program Element 64740F is the Air Force engineering development program to develop and transfer into active use the technology, tools, and techniques needed to cope with the explosive growth in Air Force systems that use computer resources. The goals of the program are to: (a) provide for the transition of computer system developments in laboratories, industry, and academia to Air Force systems; (b) develop and apply software acquisition management techniques to reduce life-cycle costs; (c) provide improved software design tools; (d) address the various problems associated with computer security; (e) develop advanced software engineering tools, techniques, and systems; (f) support the implementation of high-order languages, e.g., Ada; (g) address human engineering for computer systems; and (h) develop and apply computer simulation techniques for the acquisition process.

TABLE OF CONTENTS

CHAPTER II

ADVANCED FEATURES OF SHARP

PART TWO: APPLICATION OF SHARP

CHAPTER III

BASIC ISSUES IN OBJECT-ORIENTED DESIGN WITH ADA

CHAPTER IV

STEPS FOR AN OBJECT-ORIENTED ADA-UNIQUE DESIGN

CHAPTER V

SHARP IN DoD SOFTWARE DESIGN DOCUMENTS

CHAPTER VI

TESTING OBJECT-ORIENTED ADA SOFTWARE

CHAPTER VII

ESTIMATING THE COST OF OBJECT-ORIENTED ADA SOFTWARE

CHAPTER VIII

TEACHING OBJECT-ORIENTED ADA

SUMMARY

This paper presents a methodology for representing a large and complex computer program using graphics and Ada-based annotated pseudo code. It describes the application of the graphical representation, referred to as Structured Hierarchical Ada Representation using Pictographs (SHARP), in the design and test of computer programs, and presents a concept of operation for generating the grahics in a computer aided manner. The resulting tool is considered important, since design and test costs account for over 60% of software development costs. The tool also applies to software maintenance, which typically exceeds the original development cost by more than 50%.

BACKGROUND

DoD has mandated the use of Ada in the implementation of mission-critical software. Standardizing to a single high-order language will contribute to lower software life-cycle costs, since, for example, fewer compilers will have to be developed and the labor force will not be subdivided among several languages. Furthermore, it promotes the development of tools applicable to the entire software life cycle, since with one language a large market will exist for each tool.

In addition, proponents of Ada expect that its technical features will also help reduce software development costs. For example, Ada packages can be used to encapsulate reusable software, for such things a hardware drivers, communication protocols, high and low level I/O, math functions and special purpose algorithms. The Ada packages and their contents can be made general purpose through the Ada generics capability, whereby the name of a program unit, and typically the definition of its types and the range of permissible values for passed parameters, are created during compilation. (The process of creating a particular instance of the generic program unit is referred to as generic instantiation.)

Another important technical feature of an Ada package is its information hiding capability. It can be used to provide the framework for the implementation of object-oriented designs, as means for controlling dependency relationships between variables, types and program units.

In the past, global parameters and routines were shared among many program units. If during development and maintenance any one of the global parameters or routines were modified to correct an error, the change could adversely affect many different parts of the computer program. When one error was corrected by a change, several others often were introduced. Seemingly innocent changes, at times, caused serious problems. However, by localizing design complexity using principles of object-oriented design, the effect of a change is trapped within the implementation of an object itself.

Concept of an Object-Oriented Design

With an object-oriented design, a large computer program is composed using multiple objects. Each object is a system component implemented in software using a set of operations unique to it and a local state defined

in a data structure. The unique operations are known only to the internals of the object implementation. Object implementations typically are independent and interact with only one or two other object implementations.

Object Implementations in Ada Packages

With Ada, objects can be implemented in Ada packages, which act as containers of data structures and other Ada program units. Like other Ada program units, an Ada package consists of a specification and a body. The contents of the specification are visible to other program units, while the contents of the body are not accessible by program units external to the body. Therefore, using Ada packages the complexities of object implementations can be hidden in the package's body.

Parameters passing between object implementations can be accomplished by communicating program units declared in the specification of the package. To the extent possible, the passed parameters should not include variables and flags used in the formulation of operations unique to the object implementation. In this way, coupling by parameters passed between objects can be avoided. If potentially coupling parameters are passed, with Ada they can be of private or limited private types. In this way, accessing program units have limited use of such passed parameters.

COSTS OF SOFTWARE DEVELOPMENT

The high cost of developing software constructed using coupled program units has proven to be the rule and not the exception. In addition, as we have already indicated, experience has shown the maintenance and improvement of such software over ten years or so, once it is put to use, may cost much more than the original development cost.

Cost models have been formulated and calibrated to estimate the cost of such highly coupled software. For example, the Constructive Cost Model (COCOMO) projects the cost to develop this software as a function of its size, type of application and characteristics of the development process. Although existing cost models have not been formulated or calibrated for software developed in an object-oriented manner, they can be indirectly used to project the costs. Figure i shows possible costs when comparing the software developed in a traditional (highly coupled) manner to software developed in an object-oriented manner using Ada.

The cost of software developed in a traditional manner was established by directly applying COCOMO. The cost of software developed in an object-oriented manner using Ada, was calculated using the following relationship:

$$\begin{aligned}(\text{Cost}) = &(\text{Design Cost}) + (\text{Object Implementation Cost}) \\ &+ (\text{Object Implementation Integration Cost})\end{aligned}$$

The design cost was established with COCOMO by assuming the cost to establish a traditional design is essentially equivalent to the cost to establish an object-oriented design.* The object implementation cost was established by summing the cost to develop each object separately.

* Design costs for both traditional and object-oriented designs would decrease significantly with an automated SHARP system, which is described in Chapter IX. These savings, however, cannot be quantified at this time and have not been considered in Figure i.

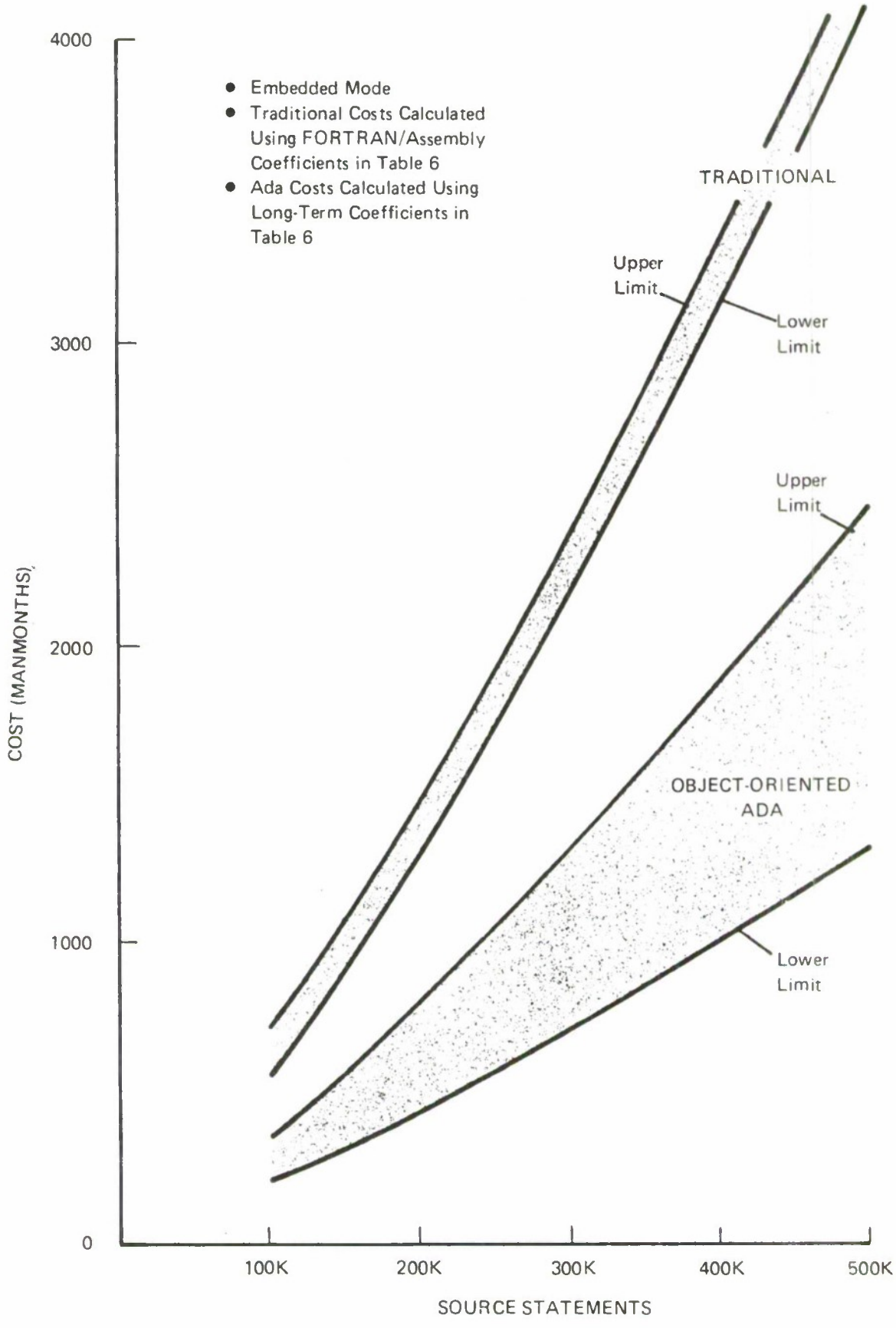


FIGURE i. SOFTWARE DEVELOPMENT COSTS*

*See Section 4.4 of Chapter VII for a description of the estimation technique.

The integration of object implementations is not accounted for by COCOMO. However, as a lower limit, this effort could be assumed to cost about the same as the cost to integrate an equal number of program units, since loosely coupled interfaces between object implementations should not be more complicated than typical program unit interfaces. The accuracy of this assumption decreases as the extent of coupling between the implementations increases. In the limit, the cost to develop strongly coupled objects approaches the cost to develop software in a traditional manner, where strong dependency relationships drive costs.

Although we can argue where between the limits the expected cost of object-oriented development using Ada should lie, most will agree that with respect to computer programs, "bulk is bad". Using object-oriented design principles, a "bulky" computer program can be developed as a set of relatively small object implementations rather than one large computer program with highly coupled routines. Each object can be developed independently in a cost effective manner, and then integrated with other object implementations.

As COCOMO and other cost models indicate, development costs increase more than linearly with the size of a computer program. We feel this is in large part, due to the increase in complex dependencies between variables, types and program units. Thus, by using relatively small object implementations and constraining their interaction, software development costs decrease.

This is especially relevant to extremely large computer software systems like those needed for the Strategic Defense Initiative, projected to consist of up to 50 million lines of source code. The complexity of dependency relationships for such large software systems will drive software costs very high, unless effectively controlled.

GRAPHICAL ABSTRACTION OF SOFTWARE DESIGNS

When written in accordance with effective style guidelines, most feel a computer program written in Ada is locally readable to those familiar with the Ada language. If we examine a fragment of Ada code, we can realize the design of that fragment. However, readable software means more than this. We must be able to easily understand the relationship of the fragment to the whole.

In the past, hierarchical block diagrams have been used to represent the interrelationship between program units. However, conventional hierarchical block diagrams are not adequate with Ada since they do not distinguish between the different kinds of Ada program units; they do not represent concurrent program unit execution that takes place in Ada; and they do not provide a mechanism for representing the unique capabilities of Ada to partition a large and complex computer program into understandable parts using object-oriented techniques.

Recognizing this problem, R.J.A. Buhr of Carleton University, Ottawa, Canada, has suggested the need for "blue-prints" of computer programs to be implemented in Ada. Although he concedes such software blueprints may not

be necessary during the development of small computer programs, Buhr argues that they are needed to effectively represent the structure of a large and complex computer program.

To illustrate this point, contrast the difference in design efforts undertaken by architects, who design large buildings, to home handymen, who may add a room to their home. The home handyman can proceed with minimal design information, making fragmented design decisions as he proceeds. A large architectural construction project, on the other hand, utilizes blueprints and other design documentation. The construction of a building is undertaken by many people who divide the construction project into a set of manageable parts and communicate using blueprints.

In the same manner, small software efforts undertaken by one or two people can proceed without partitioning or design structure information. However, as in construction, a plateau of project size and complexity is reached in software where it is not cost effective to proceed without explicit manageable parts and blueprints that represent the manageable parts.

The Need for Graphical Abstracts

We suggest that levels of abstracts are needed to represent the design of large and complex computer programs to be developed in a traditional or object-oriented manner and implemented using Ada. The abstracts can be pictorial representations of the design that comprise or concentrate within themselves the essential qualities of specific aspects of the design. Such abstracts, if readily readable, can help close the gap in communication between different members of a software development team. Such communication is critical to cost effective implementation of large and complex software systems.

In practice, pictorial abstracts are beneficial to both government and contractor personnel. In preparation for a software acquisition, the graphical presentation of large and complex Ada software helps instructors teach Ada in the context of the overall software system. During the acquisition of Ada software, contractor managers needs Ada abstracts to intellectually grasp the problem they must manage. A programmer can use Ada abstracts to help understand what it is that he must implement and communicate to a designer expansions or modifications he has introduced into the design.

Government reviewers need Ada abstracts to understand the design that they must ultimately approve. Review of many thousands of lines of pseudo-code or source code in a short period of time is often very difficult, if not impossible. In contrast, review of levels of the design provided in different pictorial abstracts is relatively easy.

Software engineers need abstracts to help present designs at design reviews and within design documentation. In this way, among other things, the consistency of variables passed between object implementations can be reviewed.

In the past, compilers did not check the consistency of variable and other declarations made in various parts of a computer program. Accordingly, a

program would compile easily and a unit test could quickly be initiated. Of course, problems associated with declaration inconsistencies had to be resolved as part of the testing process.

With Ada, compilers and linkers check the consistency of declarations. For example, the input/output parameters of a program unit must have their type (e.g., integer or floating point) defined, and checked at compilation and linking time. The consistency of these definitions are checked by matching the type definitions in the specification of a calling program unit to the type definitions in the specification of the called program unit.

Ada-unique diagrams and pictorial abstracts are needed to establish dependency relationships between object implementations used to establish a large Ada computer program. Failure to establish correct dependency relationships between the implementations will result in several time consuming compilation iterations during the integration of object implementations. Although incremental compilation is possible with Ada, all dependent pieces of a computer program must be recompiled at the same time. Thus, recompilation can be a significant effort.

The whole problem, of course, is magnified as the size of the Ada computer program grows. Such growth is possible today because of the capacity available in processing hardware. Today, technology offers potentially unlimited processing power and memory, thus, more and more complex applications are being undertaken.

Levels of Abstracts

Recognizing the need for graphical abstracts applicable to representing the design of computer programs, we have defined a set of Ada-unique abstracts that are applicable to both traditional and object-oriented Ada-unique designs. The graphical abstraction system is referred to as SHARP (Structured Hierarchical Ada Representation using Pictographs). It provides a notation that can be used to represent extremes in combinations of Ada program units, variables and types that a designer may choose, regardless of whether an object-oriented or more traditional design approach is being taken.

The graphics utilize pictographs to represent Ada program units. Specifically, a square is used to represent a subprogram, a parallelogram to represent a task and a rectangle to represent a package. In each case, the geometric figure is divided by a horizontal line into two parts, a narrow part representing the program unit's specification, and a wide part representing its body. The pictographs can be interconnected to represent the main program and its interface with external entities. For example, consider the diagram shown in Figure ii. In this diagram, the tasks shown are responsible for servicing a communication link, a terminal, a work station and an interfacing microprocessor. The small rectangle labeled P1 indicates a package utilized by the main program, made available through the Ada "with" clause.

The pictographs can be used to establish various other graphical options to represent the design of the program units declared in procedure MAIN. At a high level, graphical options can be used to represent Ada packages, which

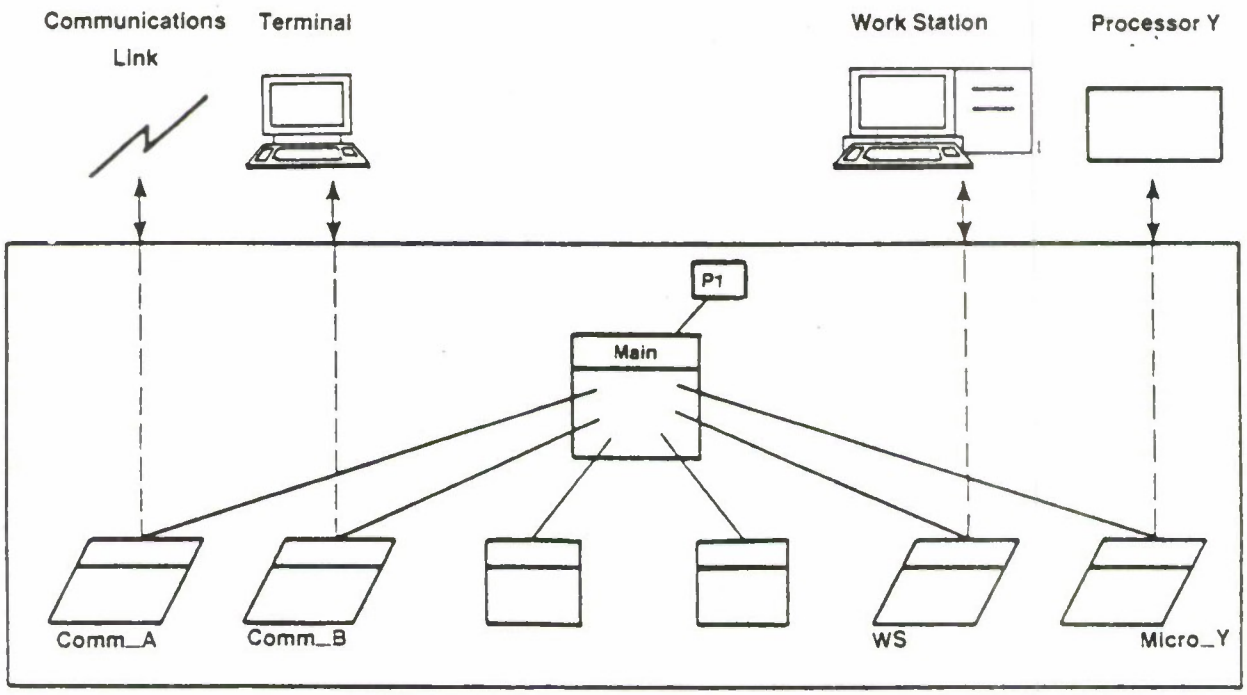


FIGURE ii. ABSTRACT FOR THE MAIN PROGRAM

can be used to encapsulate object implementations in the manner shown in Figure iii, where generic reusable software is represented by dashed lines. Such object implementations interact through communicating Ada program units visible in the package specification, as represented in the invocation abstract shown in Figure iv.

At an intermediate level, graphical options can be used to represent program units used to implement the internal complexities of an object implementation. Specifically as illustrated in Figure v, the structure of nested program units within a program unit declared in the specification of a package can be represented using a Hierarchy Diagram. The sequential set of subprogram calls within this program unit and the concurrent execution of Ada tasks within it can be represented using an Invocation Diagram.

At a yet lower level of design abstraction, options can be used to represent selective detail. As illustrated in Figure vi, these abstracts can be thought of as "blow ups" of entities identified in invocation diagrams. Abstracted detail of data flow between subprograms can be shown in a Subprogram Data Flow Diagram. Abstracted detail for task rendezvous can be shown in a Task Rendezvous Diagram. The visibility of type, constant and variable declarations can be shown in a Data Structure Diagram.

At the lowest level of abstraction, annotated pseudo code can be used to represent the bodies of each Ada program unit, as also illustrated in Figure vi. For each body, the pseudo code accounts for such things as logic, decisions, algorithms, program unit calls, input/output, generic instantiation and exceptions.

COMPUTER-AIDED SOFTWARE DEVELOPMENT

In order to implement large and complex computer programs in a cost effective manner, technological advancements in software development are necessary. Knowledge-aided design (KAD) systems that have been automated for Ada would undoubtedly significantly help reduce software development and maintenance costs. With them, software designers can rapidly generate abstracted design representations. The abstracts can be reviewed and the design representation iterated in order to, in some sense, optimize the design. Knowledge built into the KAD system helps inexperienced designers without extensive knowledge of Ada and object-oriented techniques. In this way, typical inefficiencies in design development and representation can be kept to a minimum.

Upon system turnover to users, the automatically generated graphical abstracts can be used to support software maintenance. The maintainer will be able to selectively produce abstracts that, in a systematic manner, zero in at the touch of a terminal key on parts of the program he must modify.

The abstracts will make the complexities of the design readily apparent, as opposed to culling thousands of statements in a a source code listing. The exclusive use of source code to maintain a large and complex computer program has proven to be very expensive, as we have already indicated. Such a KAD system could be developed as the first phase in the automation of SHARP, as illustrated by Item a of Figure vii.

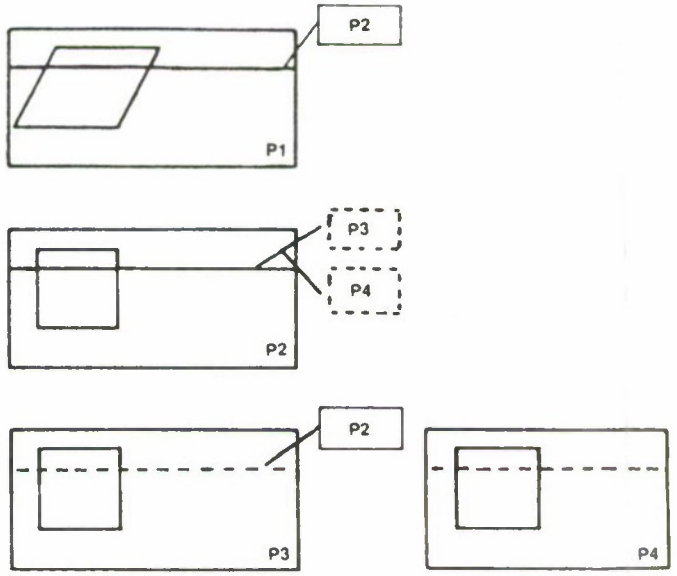


FIGURE iii. ABSTRACTS FOR OBJECT IMPLEMENTATIONS IN ADA PACKAGES

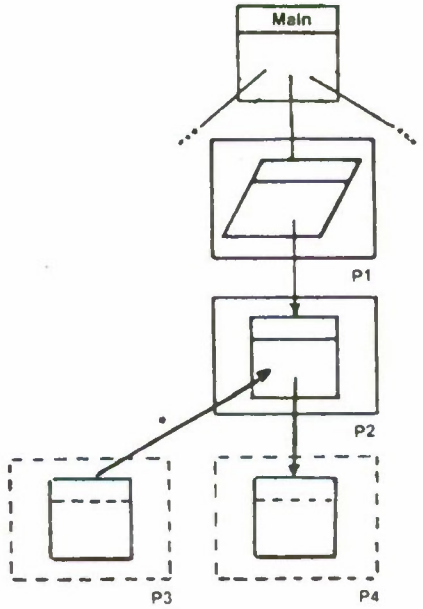


FIGURE iv. ABSTRACT FOR INTERACTION OF OBJECT IMPLEMENTATIONS

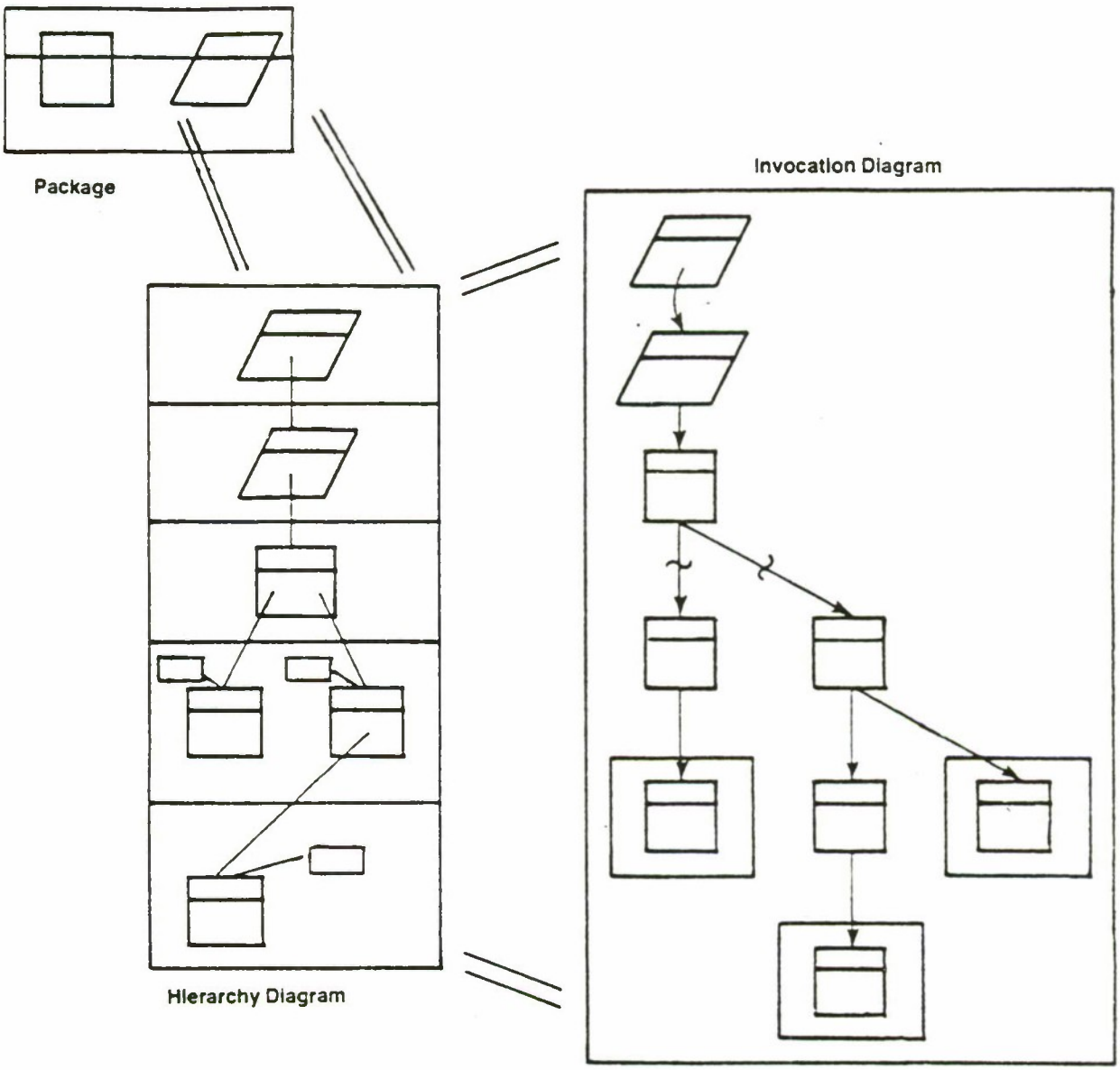


FIGURE v. REPRESENTING THE INTERNAL COMPLEXITIES OF AN OBJECT IMPLEMENTATION

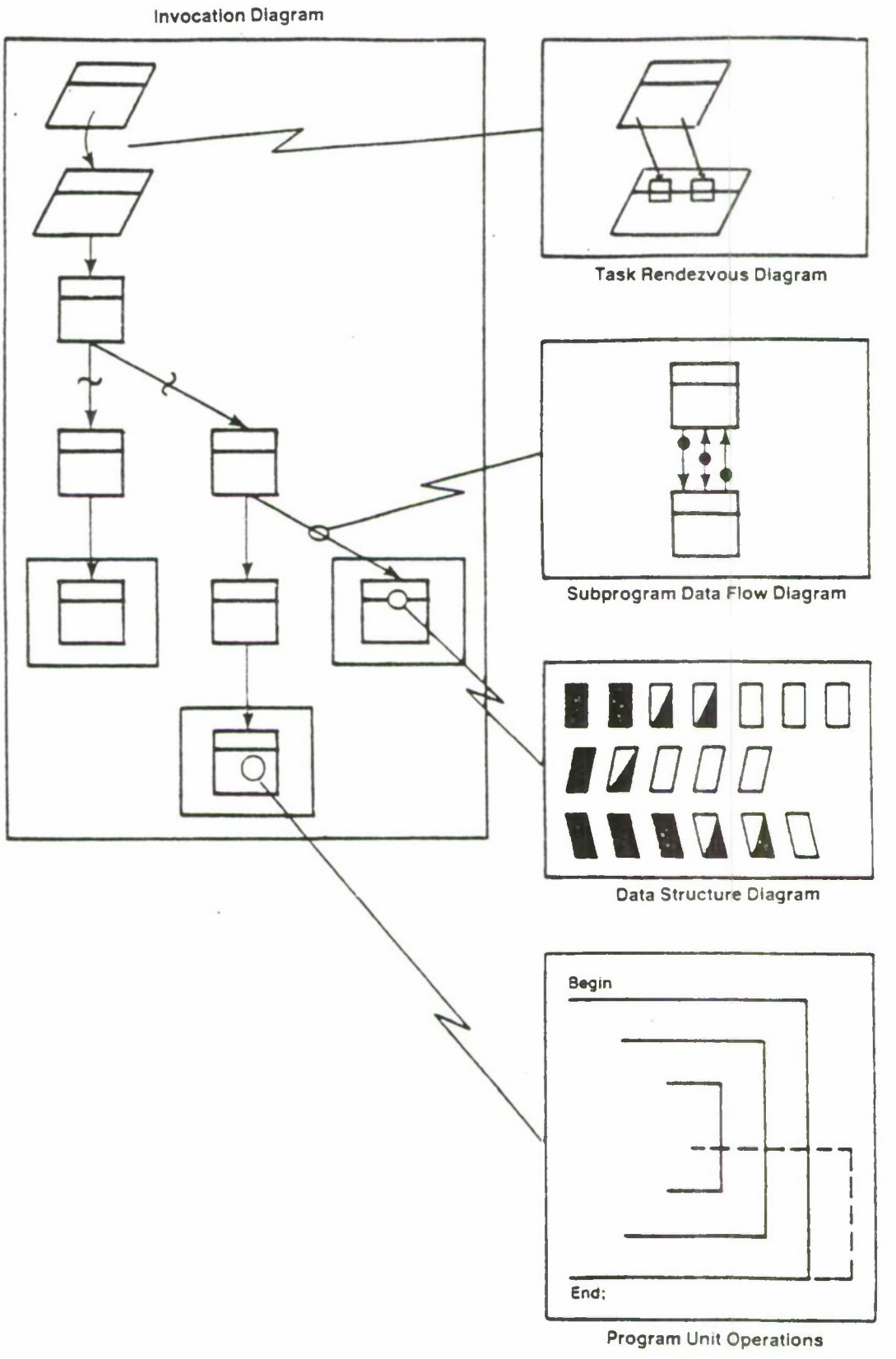
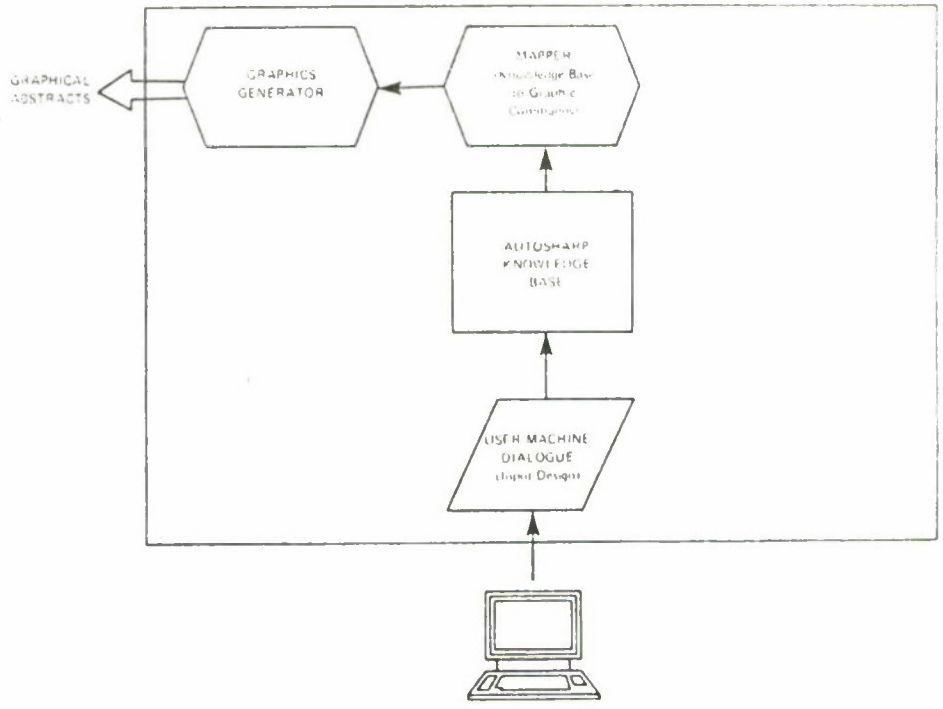


FIGURE vi. REPRESENTING SELECTED DETAIL IN AN OBJECT IMPLEMENTATION

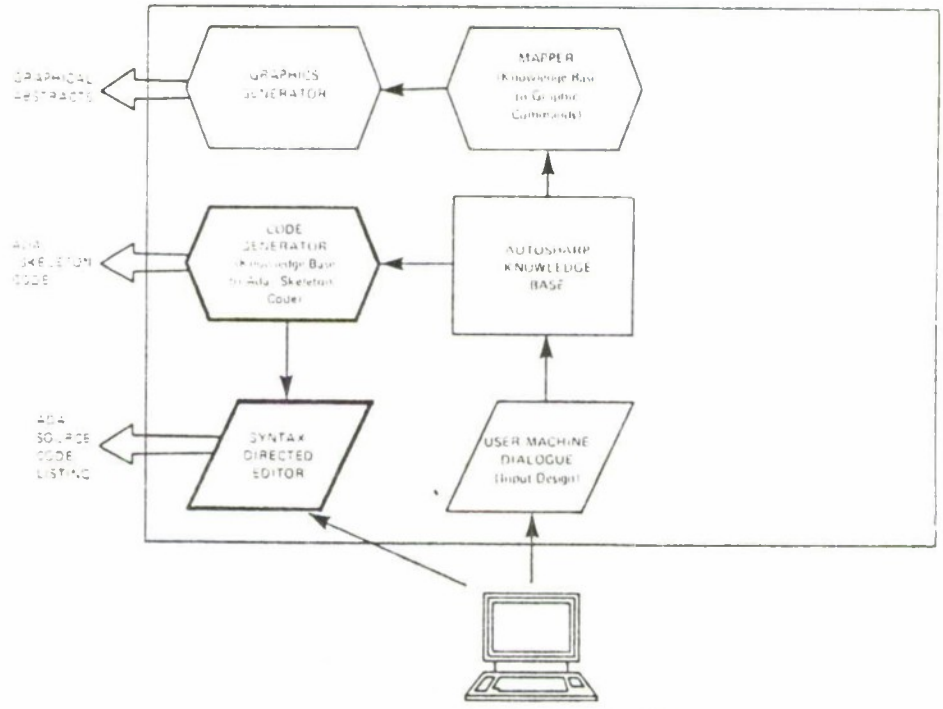
In addition to being used to establish design abstracts, the design knowledge base established by a user of a KAD system can also be mapped into Ada source code. The code would encompass aspects of the design directly accounted for in the design abstracts. This code could, in turn, be expanded and refined by a programmer using a syntax directed editor. Such an automatic programming capability could be developed as the second phase in the automation of SHARP, as illustrated by Item b of Figure vii.

Furthermore, the design knowledge base could also be mapped into a size metric. The size metric, along with user inputs on the attributes of the software acquisition, could be used as inputs to a cost estimation algorithm that projects the cost to build the large and complex computer program. In this way, the cost estimation problem is merged with the automated design process so that meaningful estimates can be made. Such a cost estimation capability could be developed as the third phase in the automation of SHARP, as illustrated by Item c of Figure vii.

Applying such a computer-aided software engineering tool in conjunction with object-oriented designs, is needed to help effectively acquire software and reduce software development and maintenance costs. The automated SHARP (AUTOSHARP) system is especially applicable to the development of large software systems like those required for the Strategic Defense Initiative. We feel the development and transfer of such technology into use will bring significant improvement in software productivity.

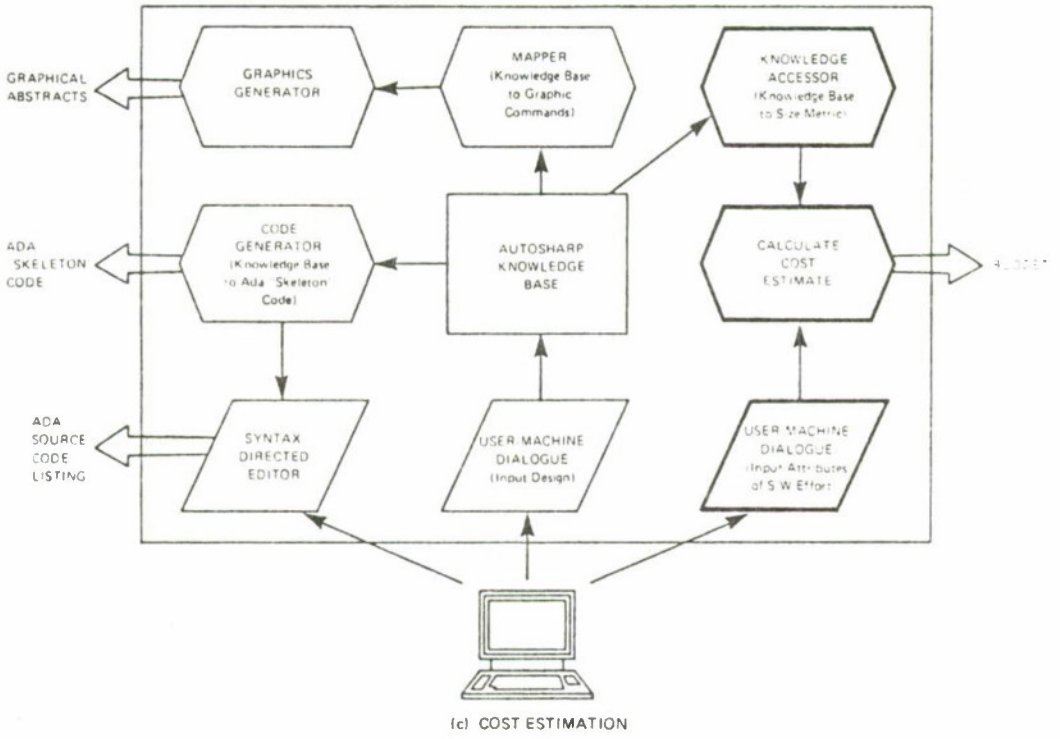


(a) KNOWLEDGE-AIDED DESIGN



(b) AUTOMATIC PROGRAMMING

FIGURE vii. COMPUTER-AIDED SOFTWARE ENGINEERING



(c) COST ESTIMATION

FIGURE vii. (CONCLUDED)

CHAPTER I

BASIC FEATURES OF SHARP

This chapter introduces Structured Hierarchical Ada Representation using Pictographs (SHARP). It describes the need for SHARP and the beneficiaries of SHARP. It establishes pictographs that can be used to represent Ada program units and describes the use of the pictographs to graphically represent the structure of an Ada computer program at a high level.

This chapter is very basic in nature and is meant as an introduction to those unfamiliar with the Ada language.

1 INTRODUCTION

1.1 WHAT SHARP IS

SHARP is defined by a criteria for establishing pictorial abstracts* of a large and complex computer program to be programmed using the DoD high order language Ada (ANSI/MIL-STD 1815A). Standards of the criteria dictate how to establish selective pictorial options, encompassing Ada unique versions of high and intermediate level abstracts for Ada packages, program unit hierarchy and program unit invocation. In addition, the pictorial options can be used to establish low level abstracts for Ada task rendezvous, data declarations and data flow between Ada subprograms. At the lowest level, options utilize annotated pseudo code to represent computer program logic and operations on variables. At this lowest level, SHARP junctions with design presentations that utilize Ada-based program design language.

The abstracts apply to traditional design approaches as well as object-oriented. The object-oriented approach is emphasized since it is the approach expected to be widely used in the development of large and complex Ada computer programs.

1.2 BACKGROUND

1.2.1 Projections for Software Acquisition Costs

The performance of defense systems has become increasingly dependent upon embedded computers. The development of software needed to program the embedded computers currently incurs a significant cost during system acquisition.

---

* In terminology of SHARP, an abstract is a pictorial representation that comprises or concentrates in itself the essential qualities of specific aspects of an Ada computer program (e.g., rendezvous between tasks or data flow between subprograms); synonym - pictorial compendium.

The acquisition of a single defense system can include the development of computer programs consisting of more than 600,000 instructions. Presently, software development costs during one system acquisition can exceed 60 million dollars. Moreover, the operation, maintenance, and continued improvement of the software over 10 years or so once a system is put to use can cost as much as 50 times the software development cost.[1]

An industry team, under the guidance of the Electronic Industries Association, has projected that by 1990 overall annual DoD software costs will reach 32 billion dollars, while annual hardware costs will reach only 6 billion dollars.

Since annual software costs have been projected to cost more than six times annual hardware acquisition costs by 1990, DoD has recognized the need to develop and implement capabilities that will reduce software acquisition and maintenance costs. Accordingly, as one step in controlling these costs, DoD has sponsored the development of Ada.

1.2.2 The Evolution of Ada

The military services, U.S. industry, and our NATO allies were intimately involved in the definition of requirements for Ada. During 1975 and 1976, preliminary requirements for Ada were distributed to a large audience for comment. In June 1976, a complete set of Ada requirements were published.

DoD used two criteria when establishing the requirements. First, Ada had to be a high quality product. Second, Ada was meant to accrue foreign acceptance and domestic acceptance outside the defense industry.

With this in mind, DoD released an international request for proposal (RFP), asking for a preliminary design of the Ada language. Four contractors were selected from those responding to the RFP for the preliminary design. They were Softech, Intermetrics, SRI International, and Honeywell.

Upon completion, the preliminary designs were distributed for comment. Eighty formal evaluation reports were received, having been submitted by DoD organizations, U.S. and European industry, plus the Ministries of Defense of the United Kingdom, France, and Germany. In April 1978 and in April 1979, public hearings were held to give software engineers throughout the world an opportunity to gain an understanding of the rationale for design decisions.

After extensive review, the Honeywell design was chosen. In 1983, the formal definition of the Ada language was standardized as ANSI/MIL-STD 1815A.

1.2.3 The Need for SHARP

When written in accordance with effective style guidelines, most feel a computer program written in Ada is locally readable to those familiar with the Ada language. If we examine a fragment of Ada code, we can realize the design of that fragment. However, readable software means more than this. We must be able to easily understand the relationship of the fragment to the whole.

In the past, hierarchical block diagrams have been used to represent the interrelationship between program units. However, conventional hierarchical block diagrams are not adequate with Ada since they do not distinguish between the different kinds of Ada program units; they do not represent concurrent program unit execution that takes place in Ada; and they do not provide a mechanism for representing the unique capabilities of Ada to partition a large and complex computer program into understandable parts using object-oriented techniques.

Recognizing this problem, R.J.A. Buhr of Carleton University, Ottawa, Canada, suggests in his book System Design with Ada[2] the need for "blueprints" of computer programs to be implemented in Ada. Although he concedes such software blueprints may not be necessary during the development of small computer programs, Buhr argues that they are needed to effectively represent the structure of a large and complex computer program.

To illustrate this point, contrast the difference in design efforts undertaken by architects, who design large buildings, to home handymen, who may add a room to their home. The home handyman can proceed with minimal design information, making fragmented design decisions as he proceeds. A large architectural construction project, on the other hand, utilizes blueprints and other design documentation. The construction of a building is undertaken by many people who divide the construction project into a set of manageable parts and communicate using blueprints, which present design information in a series of pictorial abstracts.

In the same manner, small software efforts undertaken by one or two people can proceed without partitioning or design structure information. However, as in construction, a plateau of project size and complexity is reached in software where it is not cost effective to proceed without explicit manageable parts and blueprints that represent the manageable parts.

1.2.4 Beneficiaries of SHARP

Pictorial abstracts of SHARP are beneficial to both government and contractor personnel. In preparation for a software acquisition, the graphical presentation of large and complex Ada software will help instructors teach Ada in the context of the overall software system. Specifically, it will help the instructor explain the notions of design abstraction and information hiding in conjunction with object-oriented or other design techniques; and how such designs are facilitated with Ada.

During the acquisition of Ada software, contractor and government personnel need Ada abstracts. Contractor managers need Ada abstracts to intellectually grasp the problem they must manage. Experience has shown that misunderstood projects will most likely go astray. Initial budgets tend to be insufficient and resource allocation during the course of the project may not be appropriate.

A programmer can use Ada abstracts to help understand what it is that he must implement. Also, the programmer needs a mechanism for communicating back to a designer expansions or modifications he has introduced into the design.

Government reviewers need Ada abstracts to understand the design that they must ultimately approve. Review of many thousands of lines of pseudo code or source code in a short period of time is often very difficult, if not impossible. In contrast, review of levels of the design provided in different pictorial abstracts will be relatively easy.

Later in the software life cycle, SHARP would also help the government in the maintenance of large Ada computer programs. With SHARP automated within a workstation, maintainers will be able to selectively produce Ada abstracts that zero in, at the touch of a terminal key, on the parts of a computer program that must be modified. The maintainer's learning curve will be faster with understandable abstracts complementing documentation and Ada source code.

Software Engineers need abstracts to present designs prepared using object-oriented design techniques, which can be uniquely implemented using Ada. For example, Ada abstracts are needed to present such designs taking into account mechanisms in Ada for (a) layers of packages, (b) levels of other program units and (c) hiding information within the packages and other program units. These Ada mechanisms are especially important because of the criticality of controlling complicated dependency relationships possible in large, complex computer programs.

In the past, compilers did not check the consistency of variable and other declarations made in various parts of a computer program. Accordingly, a program would compile easily and a unit test could quickly be initiated. Of course, problems associated with declaration inconsistencies had to be resolved as part of the testing process.

With Ada, compilers and linkers check the consistency of declarations. For example, the input/output parameters of a program unit must have their type (e.g., integer or floating point) defined, and checked at compilation and linking time. The consistency of these definitions are checked by matching the type definitions in the specification of a calling program unit to the type definitions in the specification of the called program unit.

Ada-unique diagrams and pictorial abstracts are needed to establish dependency relationships in a large Ada computer program. Failure to establish correct dependency relationships will result in several time consuming compilation iterations. Although incremental compilation is possible with Ada, all dependent pieces of a computer program must be recompiled at the same time. Thus, recompilation can be a significant effort.

The whole problem, of course, is magnified as the size of the Ada computer program grows. Such growth is possible today because of the capacity available in processing hardware. Today, technology offers potentially unlimited processing power and memory, thus, more and more complex applications are being undertaken.[2]

1.3 CHAPTER SCOPE

In this chapter, pictographs are defined to represent Ada program units and the interconnection of the pictographs is described in the representation of the high level design of an Ada computer program.

Section 2 describes Ada program units and comments on their use as the building blocks of an Ada computer program. It introduces the pictographs established by SHARP to graphically represent the Ada program units. Section 3 discusses utilizing Ada program units to establish the upper levels of a design for an Ada computer program. Section 4 states conclusions.

2 PICTOGRAPHS

The basic building blocks of an Ada computer program are the program units called subprograms, tasks, and packages. As suggested by Buhr, we can compare the use of these building blocks in the implementation of Ada computer programs to the implementation of electronic hardware. Hardware components are connected together using cables, plugs and sockets, all with well defined interface characteristics. Several of the hardware components can operate concurrently. Correspondingly, Ada can be conceptually thought of as program units connected together with well defined interfaces and with several of the program units operating concurrently.

This section provides an overview of the Ada program units and establishes pictographs that can be used to graphically represent them.

2.1 REPRESENTATION OF AN ADA SUBPROGRAM

2.1.1 Ada Subprogram Overview

As a basic building block of an Ada computer program, an Ada subprogram can be used to encapsulate a set of logically related operations on variables, data manipulations and other processing. This permits dividing sequential processing into manageable pieces. There are two kinds of subprograms -- procedures and functions.

The main program in Ada is an Ada procedure that is invoked upon activation of the Ada computer program. In addition, Ada procedures are nested within other Ada program units and invoked through a procedure call statement.

Ada functions are also nested within other Ada program units. However, in contrast to a procedure, a call to a function is embedded in an expression. Therefore, a function is invoked upon execution of an expression.

A subprogram consists of a specification and a body. The specification is a single Ada source instruction that establishes the name of the subprogram and the characteristics of its parameter passing.

The body implements processing to be undertaken upon execution. It consists of multiple Ada source instructions, which are clearly distinguishable from the specification.

2.1.2 Ada Subprogram Pictograph

SHARP utilizes a square to represent an Ada subprogram. The square is divided into a small narrow rectangle representing the subprogram's specification and a large rectangle representing its body, as illustrated in Figure 1.

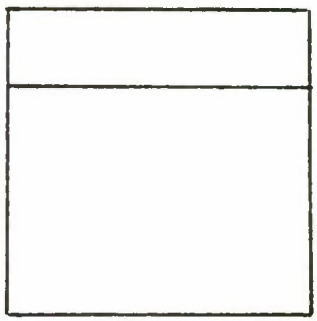


FIGURE 1. PICTOGRAPH FOR AN ADA SUBPROGRAM

2.2 REPRESENTATION OF AN ADA TASK

2.2.1 Ada Task Overview

Ada tasks are the building blocks of an Ada computer program that execute concurrently. Within a single processor, an Ada task operates in parallel with other Ada tasks in the time-slice sense of the word under control of an Ada run-time environment.

Ada tasks can be nested within the main program of an Ada computer program to account for all processing to be undertaken concurrently by that computer program. Ada tasks are also used to service interrupts, implement action queues, and implement other concurrent operations.

An Ada task consists of a specification and a body, in a manner similar to an Ada procedure. In contrast to the specification of a procedure which is a single Ada statement, the specification of a task consists of one or more Ada source statements that establish the name of the task and describe the characteristics of inter-task communication. Also, in contrast with procedures which are invoked upon request by a caller, task interaction is consummated by the callee (or acceptor) rather than the caller. The word "rendezvous" is used to describe such task interaction.

The body of a task implements processing to be undertaken upon execution. As is the case of a procedure, the body consists of multiple Ada source statements, which are clearly distinguishable from the specification.

2.2.2 Ada Task Pictograph

SHARP utilizes a parallelogram to represent an Ada task. The parallelogram is divided into a small narrow parallelogram representing its specification and a large parallelogram representing its body, as illustrated in Figure 2.

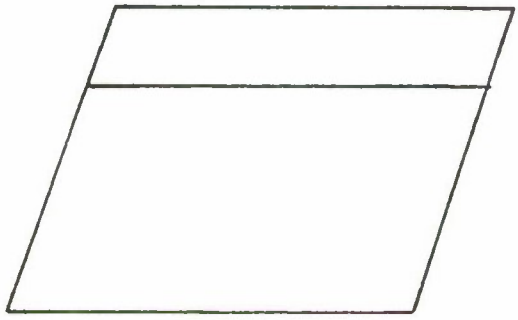


FIGURE 2. PICTOGRAPH FOR AN ADA TASK

2.3 REPRESENTATION OF AN ADA PACKAGE

2.3.1 Ada Package Overview

An Ada package is the program unit that acts as a container for other program units, data and data type declarations. An Ada package has its own specification and body. The specification consists of one or more instructions. It is used to identify the name of the package. It is also used to establish the identity of program units it contains that can be called by program units external to the package. The package specification may also include data and data type definitions that are accessible by other Ada program units.

The body of an Ada package contains the bodies of the program units declared in the specification. It also may contain data, data types plus specifications and bodies of program units that cannot be directly accessed by program units external to the package. The latter capability is basic to the implementation of object-oriented software designs with Ada. In this context, packages are used to encapsulate the implementation of objects, as discussed in subsequent chapters. Packages also serve as a mechanism for encompassing common program units and off-the-shelf reusable modules.

2.3.2 Ada Package Pictograph

SHARP utilizes a rectangle to represent an Ada package. The rectangle is divided into a small narrow rectangle representing the package's specification and a large rectangle representing its body, as illustrated in item a of Figure 3.

The specification of accessible program units within an Ada package are shown within the package's specification. The bodies of these program units program unit are shown within the package's body, as shown in item b of Figure 3. Ada program units hidden in the package's body and not accessible to program units external to the package, are not shown. The presence of variables, constants and type declarations are represented by rectangles enclosing slanted lines, as shown in item c of Figure 3.

3 APPLICATION OF THE PICTOGRAPHS

The designer of a large and complex computer program to be implemented in Ada, as an initial step in the design process, typically establishes concurrent processing threads or processes. With Ada, each process is established by a task declared in the main program. The representation of this level of an Ada design is discussed in this section.

3.1 PROCESSES

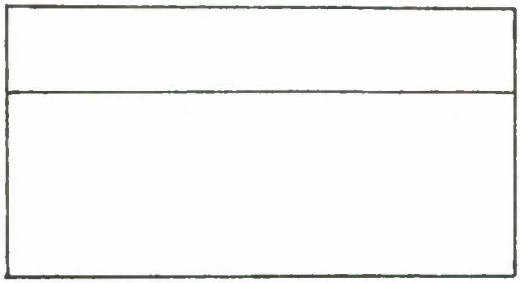
In a computer system, resident software typically has to satisfy multiple demands. For example, user commands and communication interface requests may simultaneously compete for a computer's processing time. A computer program must respond in a timely manner to the commands and requests, even when they are received at about the same instant in time.

A chain of modules can be written to implement the operations on variables, data manipulation, logic, exceptions and other processing needed to respond to each user command and communication interface request. In addition, chains of modules may have to be written to automatically initiate processing within a computer program on a periodic or some other basis. For example, a software built-in-test of equipment may be periodically initiated or processing may automatically consummate when a sensor value reaches a critical value.

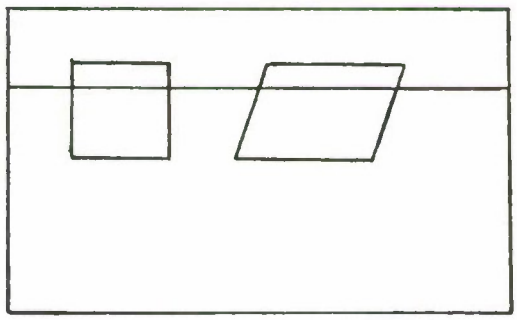
For high order languages like FORTRAN, these threads are typically referred to as processes that can be concurrently executed directly under operating system control (in a time slice manner), thus providing the timely response needed. In Ada, the concurrent execution can be accomplished using Ada tasks.

As a general rule, characteristics of processes include the following:

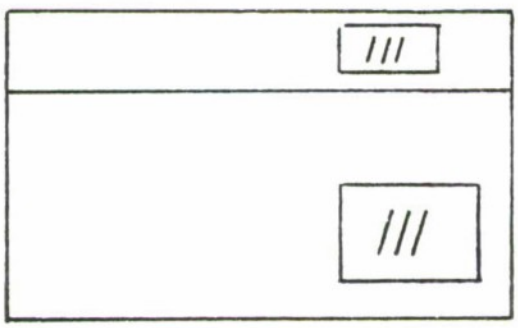
- Processes account for logic, operations on variables, data manipulation and other processing needed to (a) satisfy user commands and communication interface requests and (b) perform processing automatically initiated on a periodic or some other basis.
- Processes are loosely coupled.
- Processes can execute concurrently if necessary.



(a) BASIC PICTOGRAPH FOR AN ADA PACKAGE



(b) REPRESENTING ACCESSIBLE ADA SUBPROGRAMS AND TASKS



(c) REPRESENTING DATA STRUCTURES IN AN ADA PACKAGE

FIGURE 3. PICTOGRAPHS FOR ADA PACKAGES

3.2 REPRESENTATION OF THE HIGH LEVEL STRUCTURE OF AN ADA COMPUTER PROGRAM

With Ada, the implementation of processes can be accomplished using Ada tasks declared in the main program, as well as directly with the operating system. Each task can implement the requirements assigned to a specific process. This approach facilitates greater portability of the Ada computer program and does not necessitate knowledge of operating system configuration to establish processes. However, it may not be as memory and time efficient as directly using the operating system.

The set of Ada tasks can be graphically represented using the pictographs of SHARP. For example, Figure 4 shows Ada tasks declared in the main program to service communication links, multiple terminals, work stations and an interfacing microprocessor. In the diagram, straight lines are drawn from the body of the main program to the specifications of process tasks, and dotted lines are drawn from the tasks to a geometric figure introduced to represent the external entity the task must interact with.

The diagram also represents main program access to an Ada package through the Ada "with" clause. In Figure 4, the main program access to package TEXT_IO is represented by a small rectangle with a line drawn from it to the specification of the main program.

Furthermore, the diagram indicates subprograms nested in the body of procedure MAIN. The designer might want to use procedures, for example, to establish initial conditions at the start of execution of the large Ada computer program; and to establish restart conditions.

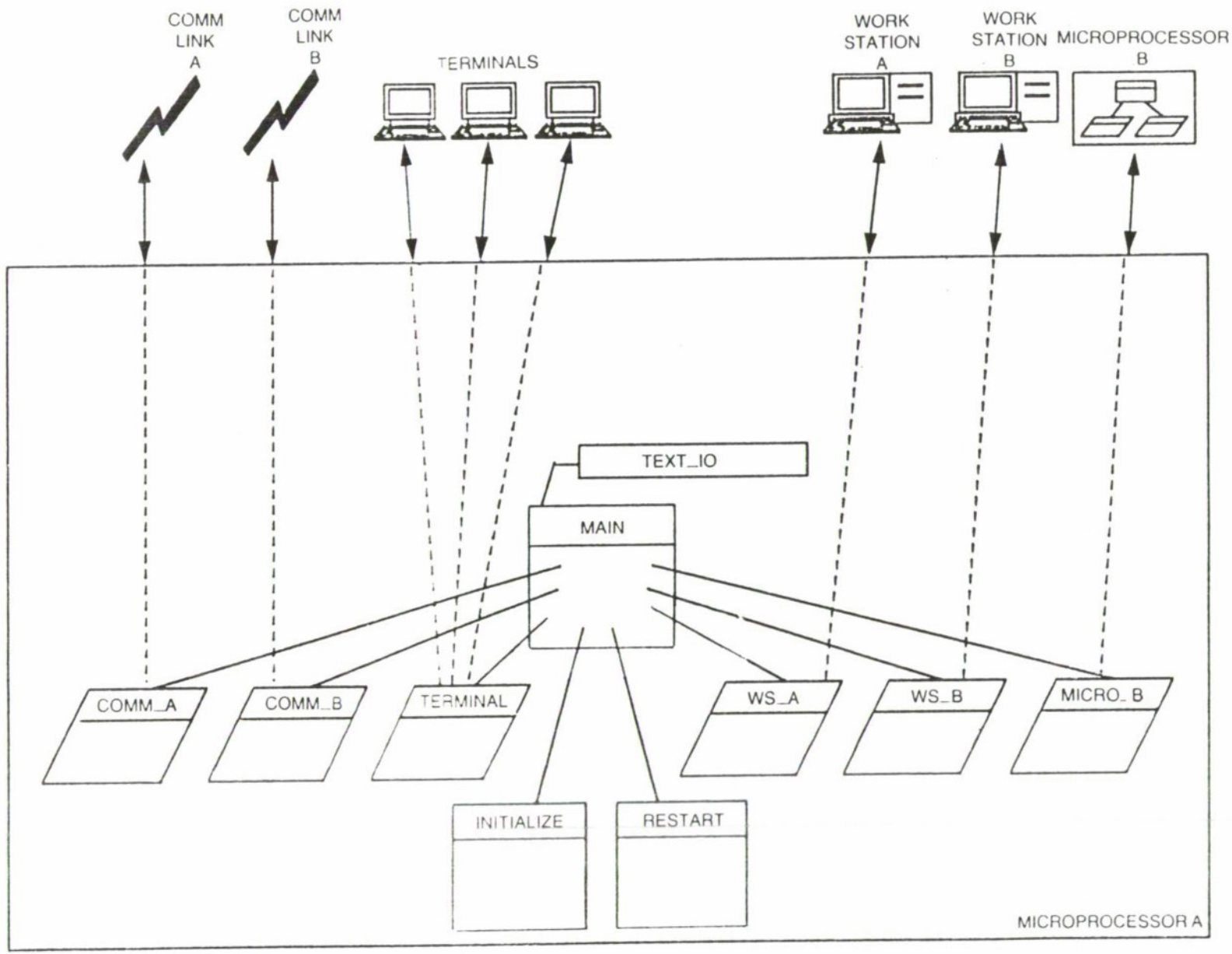


FIGURE 4. REPRESENTING THE MAIN PROGRAM

CHAPTER II

ADVANCED FEATURES OF SHARP

This chapter describes a graphical technique for presenting the object oriented design of large and complex computer programs to be implemented using Ada. The graphics present abstracts (i.e., pictorial overviews) of Ada computer programs, each of which applies to a different level of design. At the highest level, the abstracts represent Ada packages that may be used to encapsulate major partitions of a large Ada computer program. These abstracts potentially account for large amounts of code, possibly 10,000 source statements or more.

At an intermediate level, the abstracts can be used to represent the hierarchy and invocation of program units nested, for example, within a program unit declared in the specification of a package. The hierarchy diagram represents levels of nested program units and an invocation diagram represents concurrent processing with Ada tasks as well as sequential calls to multiple subprograms. These abstracts may account for as many as 5,000 source statements.

At a lower level, abstracts can be used to represent design detail. These abstracts can be envisioned as "blow ups" of entities identified in an invocation diagram. The abstracts represent details associated with task rendezvous, data flow between program units and data structures. At the lowest level, annotated pseudo code is used to represent operations, logic, input/output, generic instantiation and exception handling within the body of a program unit.

The SHARP abstracts can be used to pictorially represent all extremes of an Ada design. They can be generated using a software graphic package. In this way, we can generate Ada abstracts in a timely manner and easily iterate them to update or optimize an Ada design.

1 INTRODUCTION

1.1 SOME IMPORTANT FEATURES OF ADA

Ada provides a complete set of general purpose language features. As a programming language, Ada requires that both algorithms and data structures are specified precisely, and that the consistency in the use of variable types is verified by a compiler. Ada facilitates the construction of very large programs by providing extensive facilities for program unit modularization.

As a basic and important capability, Ada provides a framework for the construction of large programs using object-oriented design techniques, principally due to its capability to hide information in a set of Ada packages. Each package can be used to encapsulate a data structure local to one or more objects and operations unique to the objects.

A package consists of two parts - a specification and a body. Only entities contained within the package's specification can be accessed by program units external to it. The body of a package is used to implement complex operations on variables, logic and manipulation of data unique to the objects contained in the package.

A package thus allows controlled access to the results of complex and potentially lengthy programming operations and logic. As such, it provides a powerful mechanism for program modularization. A large computer program (e.g., 600,000 lines of code) can be composed with a set of packages, each with a controlled interface to other packages. In this way, potentially devastating dependency relationships across the large computer program can be controlled by careful design of package interfaces.

The development of each package itself may be a very complex and difficult job. Ada provides facilities to implement the bodies of program units declared in the specifications of packages, in accordance with the principles of software abstraction and information hiding.

As an example of software abstraction, a small and easily understood portion of the program unit is implemented at one level, while the implementation of the rest of the computer program is deferred to other levels through calls to Ada subprograms and activation of Ada tasks. At each of the other levels, the process is repeated.

1.2 PROGRAM DESIGN LANGUAGE

As discussed in subsequent chapters, designers of large and complex Ada computer programs can establish a set of packages to encompass its major parts, must carefully design each package interface so as to constrain potentially complex dependency relationships, and can design the internals of each package in an abstracted manner. Aspects of the computer program's design can be presented using Ada code (or a subset or superset of Ada code). For example, Ada code for high level program units and their specifications have been referred to as Ada Program Design language (PDL). However, Ada-based PDL may be difficult to read by those not familiar with typing and and other aspects of the Ada language. As stated in the document Program Manager's Guide to Ada [3] :

> 'Program managers, hardware designers, and communications engineers who are unfamiliar with PDL may find it difficult to review design documentation that consists largely of PDL."

In addition, some feel that Ada-based PDL by itself is not sufficient to represent the design of a large and complex computer program. When written in accordance with effective style guidelines, most feel a computer program written in Ada is locally readable (to those familiar with the Ada language). If we examine a fragment of Ada code, we can realize the design of that fragment. SHARP agrees and, in fact, utilizes annotated pseudo code similar to Ada code (i.e., a form of PDL) to present the design of operations on variables, logic and other processing within the bodies of subprograms and tasks. However, this utilization of Ada-based pseudo code is only employed at the very lowest level of SHARP design abstraction. As stated in the Program Manager's Guide to Ada:

> "PDL does not totally bridge the gap between the system level specification and the coded program. The top level software system design must be expressed."

Futhermore, Ada-based PDL for a computer program written in Ada may, in essence, be nothing more than a first "cut" at the final Ada code itself. As stated in the Program Manager's Guide to Ada:

> "If the implementation language and the PDL are the same (i.e., full Ada), programmers will tend to begin coding before the design is complete and verified."

Clearly, higher level abstracts are needed to present the set of packages used to compose a large Ada computer program. Abstracts are also needed to present the design of each package body. These abstracts must represent only essential aspects of Ada at a much higher level than that provided by PDL. The pictorial notation of SHARP can be used to establish such high level abstracts in a concise manner.

SHARP was developed recognizing that lower level abstracts are also needed, when a design reaches maturity and a programmer can become involved in the implementation of the design. Accordingly, SHARP provides lower level abstracts to represent such things as task rendezvous, data flow between subprograms, generics, and the visibility of information in data structures. However, each of these abstracts is at a substantially higher level than PDL.

1.3 CHAPTER SCOPE

Section 2 describes the selective pictorial abstracts of SHARP. Section 3 provides generic examples of the use of SHARP, in the context of a designer presenting an Ada software design prepared in an abstracted manner.

2 LEVELS OF SHARP ABSTRACTS

2.1 INTRODUCTION

As discussed in Chapter I, Ada facilitates the implementation of real-time computer programs embedded within weapon systems. As such, Ada computer programs must respond in a timely manner to independent commands, even when they are received at about the same instant in time. Ada tasks can be declared within the main program to service each command concurrently, under control of an Ada run-time environment.

Chapter I introduces graphical representation of the main program and its interface with external entities, as shown in Figure 4. In this diagram, the tasks shown are responsible for servicing a communication link, a terminal, a work station and an interfacing microprocessor.

With SHARP, various options can be used to present abstracts of the overall computer program design at a series of abstracted levels. At the highest level of design abstraction, options are used to represent Ada packages, which typically encapsulate major components of a large and complex computer

program (e.g., the components used to implement the process tasks shown in Figure 4). A catalog of these packages can be shown using the Ada Package Catalog (Option A). Each individual package is represented using an Ada Package Content Diagram (Option B). Each package may access other packages, which may, in turn, access other packages. This can be thought of as layers of packages, which can be represented using Option B as illustrated in Item a of Figure 5, where dashed lines indicate Ada generics.

At an intermediate level of design abstraction, options are used to represent program units encapsulated in a package, as illustrated in Item b of Figure 5. Specifically, the structure of nested program units within a program unit declared in the specification of a package can be represented using the Hierarchy Diagram (Option C). The sequential set of subprogram calls within this program unit and the concurrent execution of Ada tasks within it can be represented using the Invocation Diagram (Option D).

At a yet lower level of design abstraction, options can be used to represent selective detail. As illustrated in Item c of Figure 5, these abstracts can be thought of as "blow ups" of entities identified in invocation diagrams. Abstracted detail of data flow between subprograms can be shown in a Subprogram Data Flow Diagram (Option E). Abstracted detail for task rendezvous can be shown in a Task Rendezvous Diagram (Option F). The visibility of type, constant and variable declarations can be shown in a Data Structure Diagram (Option G).

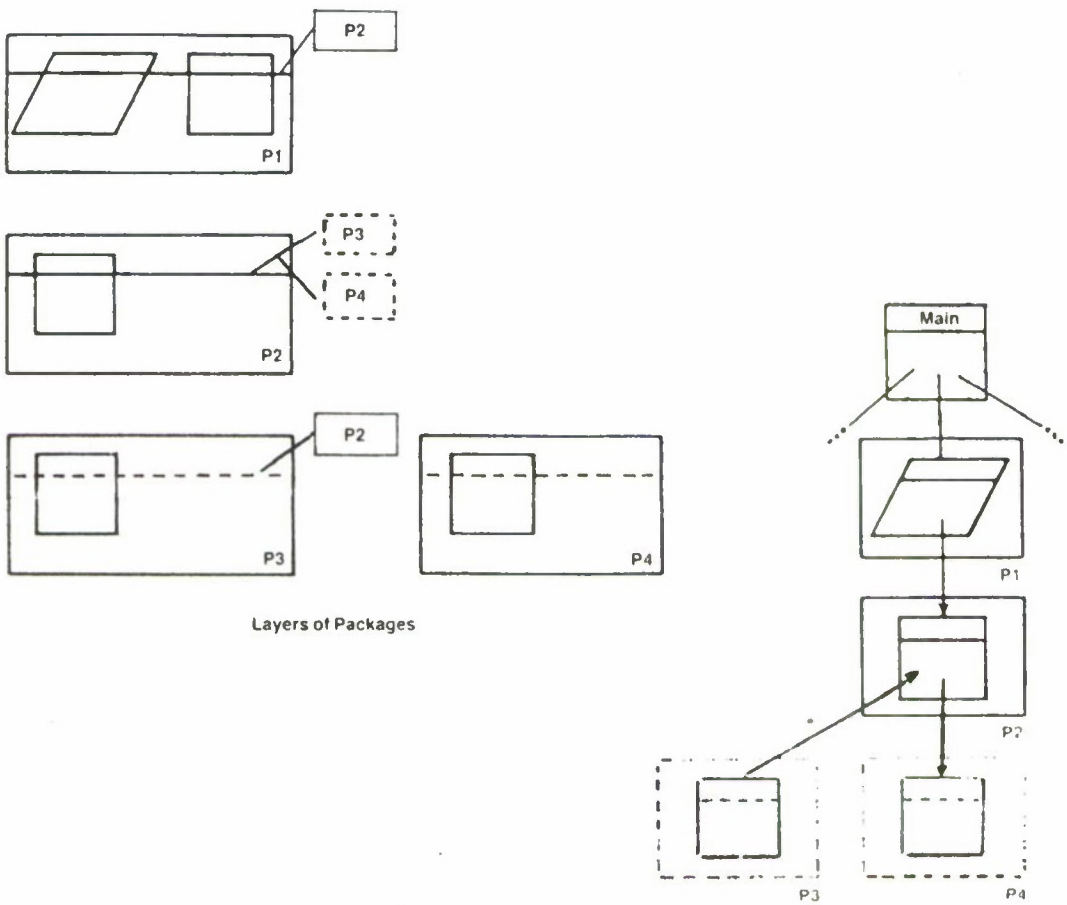
At the lowest level of abstraction, SHARP Annotated Pseudo Code (Option H) can be used to show program unit operations, logic and other processing within program unit bodies, as illustrated in Item d, Figure 5. This represents the junction of SHARP with traditional design presentations using PDL. A Data Structure Detail Glossary (Option I) can be used to represent data structure details not accounted for in a Data Structure Diagram.

2.2 HIGH LEVEL SHARP ABSTRACTS

2.2.1 Principles of Object-Oriented Design With Ada

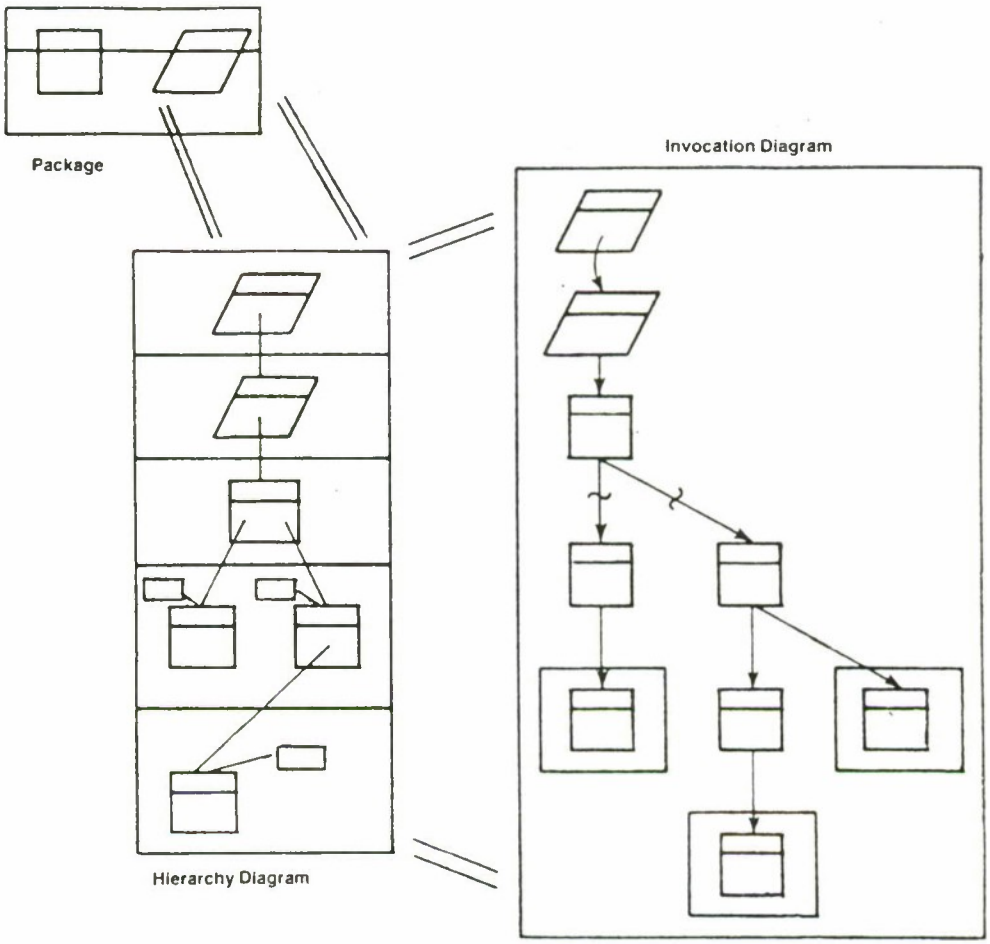
Using principles of object-oriented design, a large computer program is composed with pieces, each associated with one or more objects. An object accounts for a subset of software requirements. It is implemented using a unique set of operations and a local state defined in a data structure. The unique operations and local state are known only to the internals of the object implementation. Parameters may be passed from one object implementation to another. However, care must be taken in selecting passed parameters so as not to introduce interobject dependencies. To the extent possible, passed parameters should not include variables and flags used in the formulation of the local state and operations unique to the object implementation. In this way, undesirable coupling between object implementations can be avoided.

Requirements are assigned to objects so as to make their implementation independent and self sufficient, and in a sense, mimic real world objects, such as alarm clocks and telephones. Such real world objects make



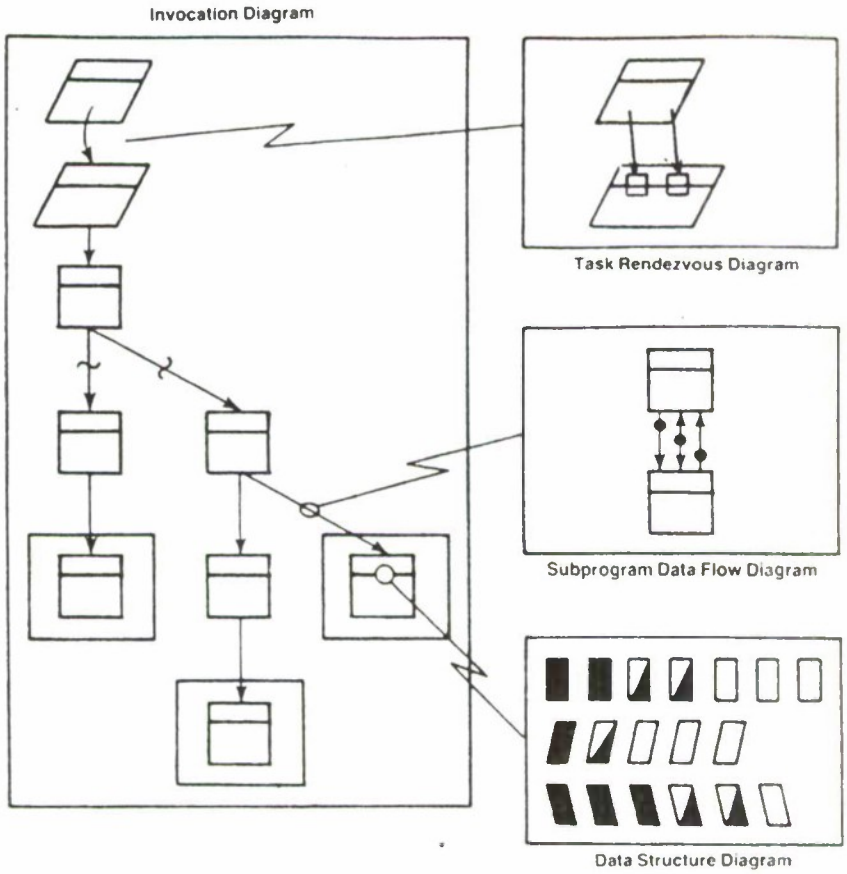
(a) LAYERS OF ADA PACKAGES

FIGURE 5. INTERRELATIONSHIP OF PICTORIAL OPTIONS OF ADVANCED SHARP



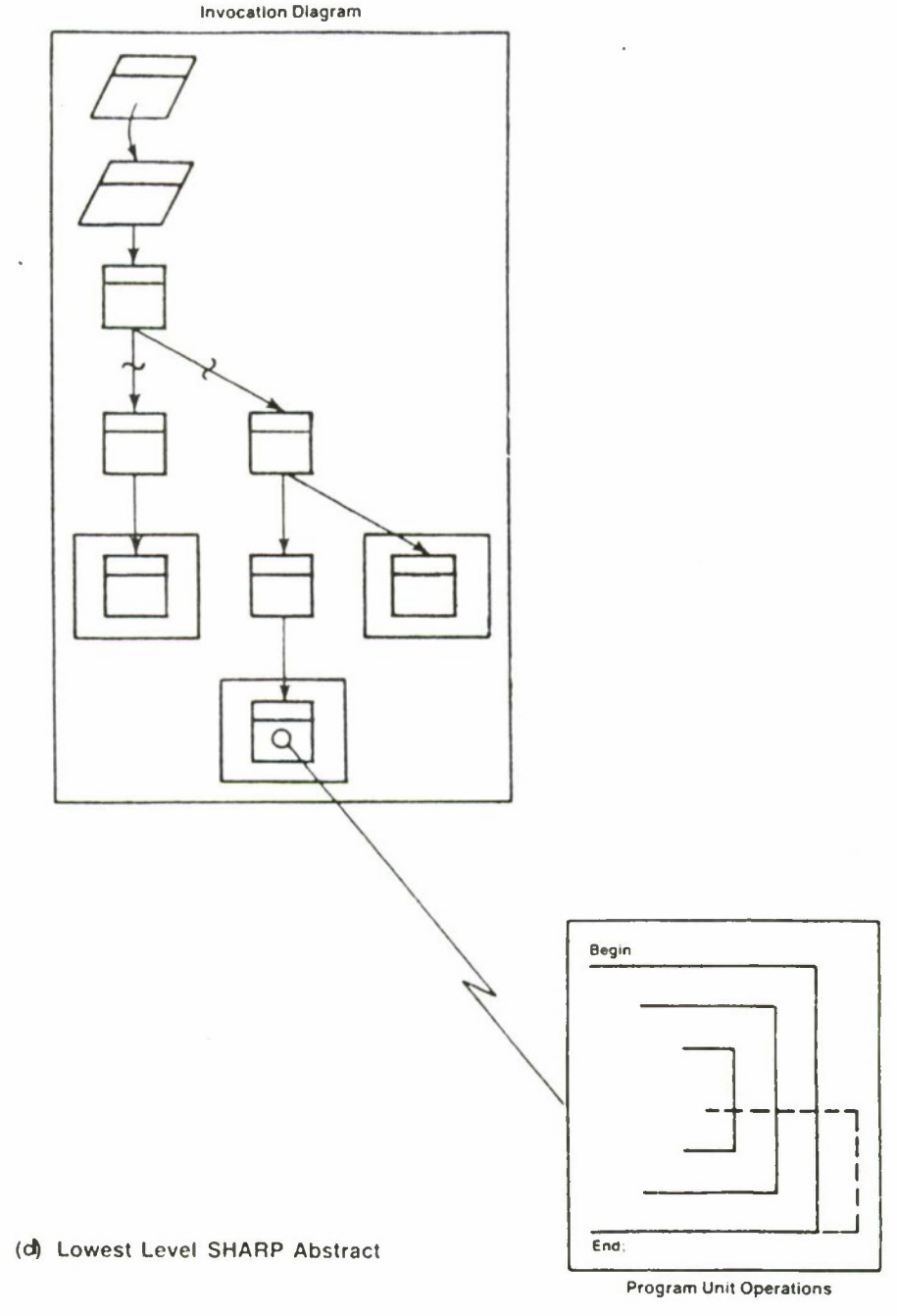
(b) Intermediate Level SHARP Abstracts

FIGURE 5. (CONTINUED)



(c) Lower Level SHARP Abstracts

FIGURE 5. (CONTINUED)



(d) Lowest Level SHARP Abstract

FIGURE 5. (CONCLUDED)

available to users a small number of basic operations (e.g., set time, enable and disable alarm, dial number, answer, hang up), while hiding from users implementation details. Hiding information is good for real world objects because it prevents interference from other objects (such as dust or grit in the case of clocks and telephones), and minimizes the number of places to look when something goes wrong. (One doesn't disassemble the alarm clock when the telephone fails to ring.)

Object implementations are very much like real world objects. They exist as relatively independent units, which can be combined together to build larger object implementations. The combination of objects then becomes a new object that presents a simple interface in the form of a collection of operations that can be performed on (or by) that object. The fact that an object may have to communicate with hundreds of its composite objects in order to accomplish an operation is completely hidden.

Older languages provide only subprograms and hence support only procedural and functional abstraction. However, as Cattage, Horowitz and Musser point out in Current Trends in Programming Methodology, "The nature of abstractions that may be conveniently achieved through the use of subroutines is limited".

With Ada, objects can be implemented using packages and tasks. However, some experts do not recommend extensive use of Ada tasks due to slow rendezvous execution times and potential difficulties in testing.

Accordingly, Ada packages are important to the implementation of object-oriented designs. Information hidden within each package limits dependency relationships between objects. Only parameters declared in program units, contained within the specification of a package, can be passed from one package to another.

Designers of large and complex Ada computer programs should choose to use an object-oriented approach, as opposed to older functional type design approaches. As Grady Booch suggests, a large software system should be built with layers of abstraction.[4] He feels that each layer should account for collections of objects. Furthermore, Booch feels that because objects may be independent and autonomous, there undoubtedly will be several threads of control active simultaneously throughout a system. When using Ada to implement an object-oriented design, Booch associates objects with Ada packages and tasks, and suggests that classes of objects should be associated with packages that export parameters of private or limited private types. By class of objects, Booch means a set of similar but unique objects. By restricting exported parameters to private or limited private, the user of the package has limited use of the passed parameter. For example, if the parameter is private, the user is excluded from applying operations on the parameter other than those operations defined within the package specification.[7] The only exception to this rule is assignments and tests for equality and inequality, which can be made. If the parameter is limited private, assignments and tests for equality are no longer automatically available. The use of private and limited private is relevant to parameters passed between object implementations that may couple the implementations. Such restrictions apply to passed parameters used in the formulation of the local state and operations unique to an object implementation.

When used in conjunction with object-oriented designs, some reusable Ada packages will themselves be object implementations (e.g., I/O device drivers, signal processing algorithms and database management systems). Other reusable Ada packages will be used to help construct object implementations (e.g., mathematical functions and data structure routines).

2.2.3 Ada Package Content Diagram (Option B)

2.2.3a Purpose

An Ada Package Content Diagram provides a pictorial abstract of an Ada package. Specifically, it is used to represent program units declared within a package's specification, the existence of data structures, other packages nested in the package's body, and package's accessed through the Ada "with" clause.

2.2.3b Description

As described in Chapter I, an Ada package is the program unit that acts as a container for various Ada entities, including subprograms, tasks, exceptions, and declarations of variables, types, subtypes, and constants. It consists of a specification and a body. The specification can be thought of as defining the contractual rights of a user, specifying visible entities within the package that the user can reference. Program units declared in the body of an Ada package are completely hidden from, and inaccessible to any users outside of the package. Similarly, types, constants and variables declared within the body of a package are hidden and inaccessible. Accordingly, packages are used extensively in designs where information hiding is to be introduced.

The Ada package content diagram represents subprograms and tasks declared in the package's specification, as shown in Figure 7. The Ada package content diagram does not represent subprograms and tasks nested with the bodies of program units visible in a package. Rather, these program units are represented by intermediate and lower level pictorial options of SHARP, described in Sections 2.3 and 2.4, respectively.

As also shown in Figure 7, the Ada Package Content Diagram can be used to represent packages nested directly in the subject package, and to represent the use of the Ada "with" clause to access other packages. The specification of a package, or its body, can access one or more other packages using the "with" clause. These packages can, in turn, access other packages. The result can be envisioned as layers of packages, as illustrated in Figure 8. In this figure, generic Ada packages are indicated by dashed lines.

Layers of packages are discussed in conjunction with object-oriented design in the example provided in Section 3.3. As described in this section, the interaction of object implementations can be shown using a SHARP Invocation Diagram (defined in section 2.3.2) for communicating program units (e.g., the visible program units declared in the specification of a package used to encapsulate an object implementation).

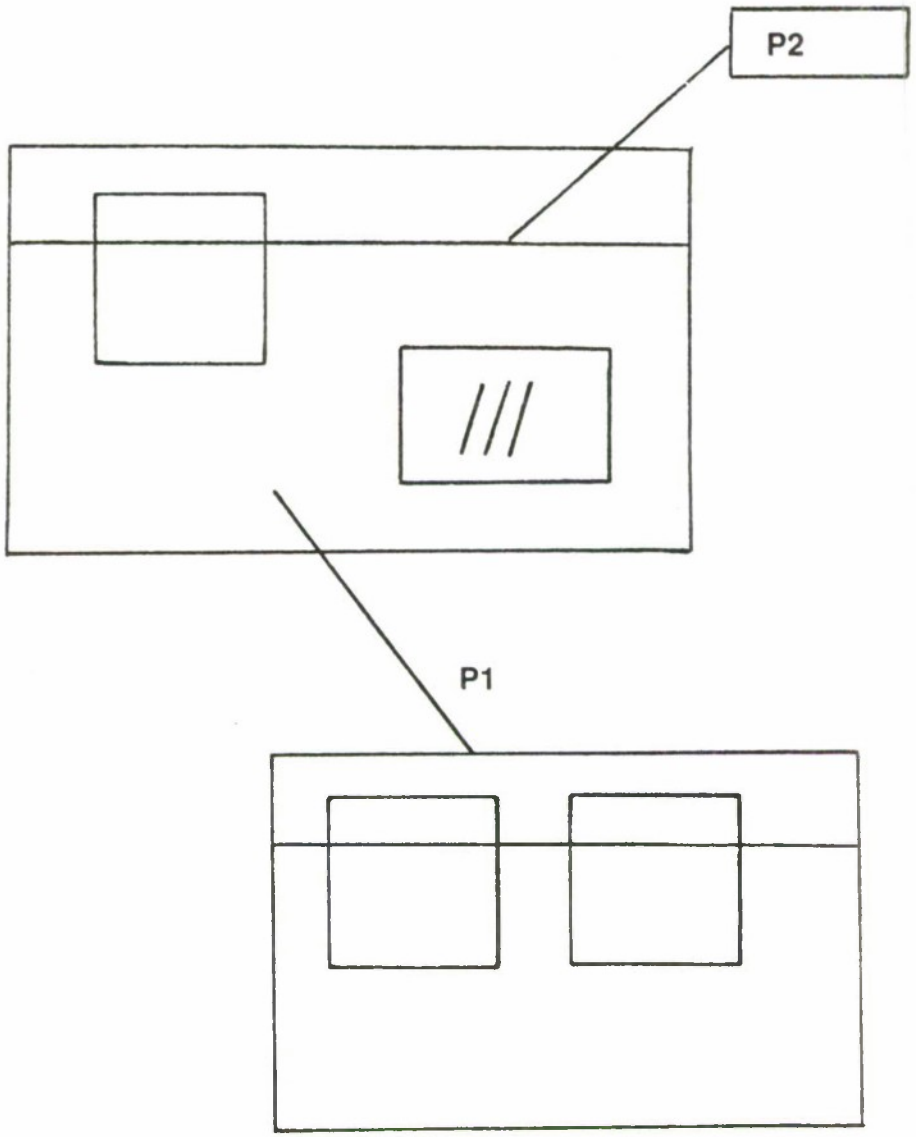


FIGURE 7. ADA PACKAGE CONTENT DIAGRAM (OPTION B)

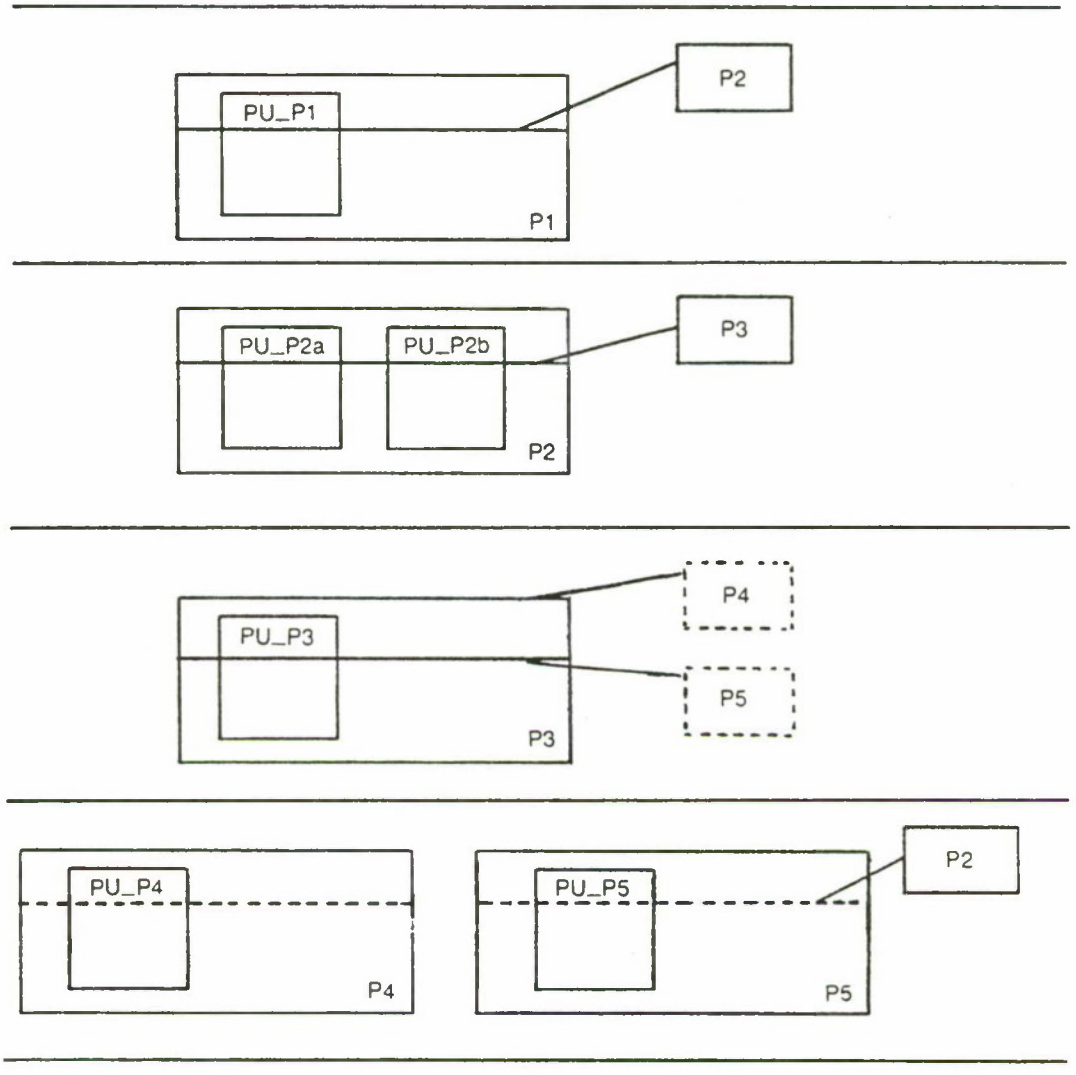


FIGURE 8. LAYERS OF ADA PACKAGES

2.3 INTERMEDIATE LEVEL SHARP ABSTRACTS

2.3.1 Hierarchy Diagram (Option C)

2.3.1a Purpose

A Hierarchy Diagram represents program units nested in a subject program unit (i.e., program units that are both declared and implemented). In addition to representing a subprogram or task declared in the specification of a package, it can be used, for example, to (1) represent program units to be nested in tasks declared in the main program, or (2) represent the nested set of program units used to implement the operational part of the main program or a package.

2.3.1b Description

A Hierarchy Diagram utilizes pictographs defined by Basic SHARP to represent program units. Specifically, a square is used to represent a subprogram, a parallelogram to represent a task, and a small rectangle to indicate a package "with" clause.

In the Hierarchy Diagram, nested program units are assigned to levels. The subject program unit is assigned to Level 1. Program units declared within the subject program unit are assigned to Level 2. In general, a program unit declared within a program unit at Level n is assigned to Level n+1.

As shown in Figure 9, a straight line is drawn from the body of a program unit at Level n to the specification of the nested program unit at Level n+1; and a straight line is drawn from the small rectangle indicating a package "with" clause to the program unit to which the clause applies.

The name of each program unit can be provided in the program unit's specification or body, or adjacent to the program unit. In Figure 9, each program unit is given the name PU (standing for Program Unit) followed by "underscore" and a program unit identifier. The identifier consists of the program unit's level number and unit letter, (e.g., PU_3b indicates unit b in Level 3). In practice, PU typically can be replaced by a name representative of the object or function implemented by the program unit (e.g., RADAR_TRACKER_2d or FFT_3e).

2.3.2 Invocation Diagram (Option D)

An Invocation Diagram displays the flow of control within a computer program. Specifically, it represents task activation, task rendezvous, and calls to subprograms, including subprograms contained in the specification of a package. The Invocation Diagram is essential in defining dependency relationships between program units, and therefore, is a useful tool in reviewing the complexity of such dependencies. Figure 10 provides an example of an Invocation Diagram associated with the Hierarchy Diagram shown in Figure 9. The following paragraphs describe the symbols used in conjunction with pictographs in an Invocation Diagram.

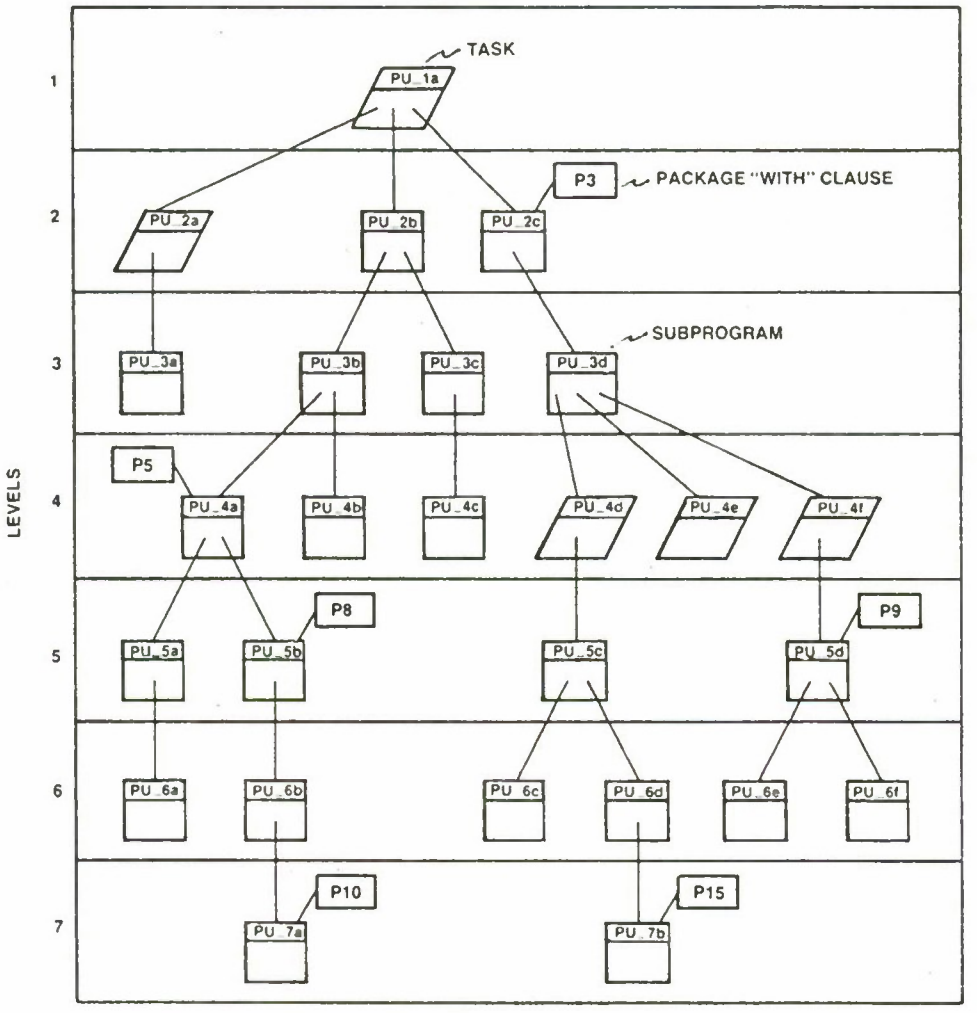


FIGURE 9. HIERARCHY DIAGRAM (OPTION C)

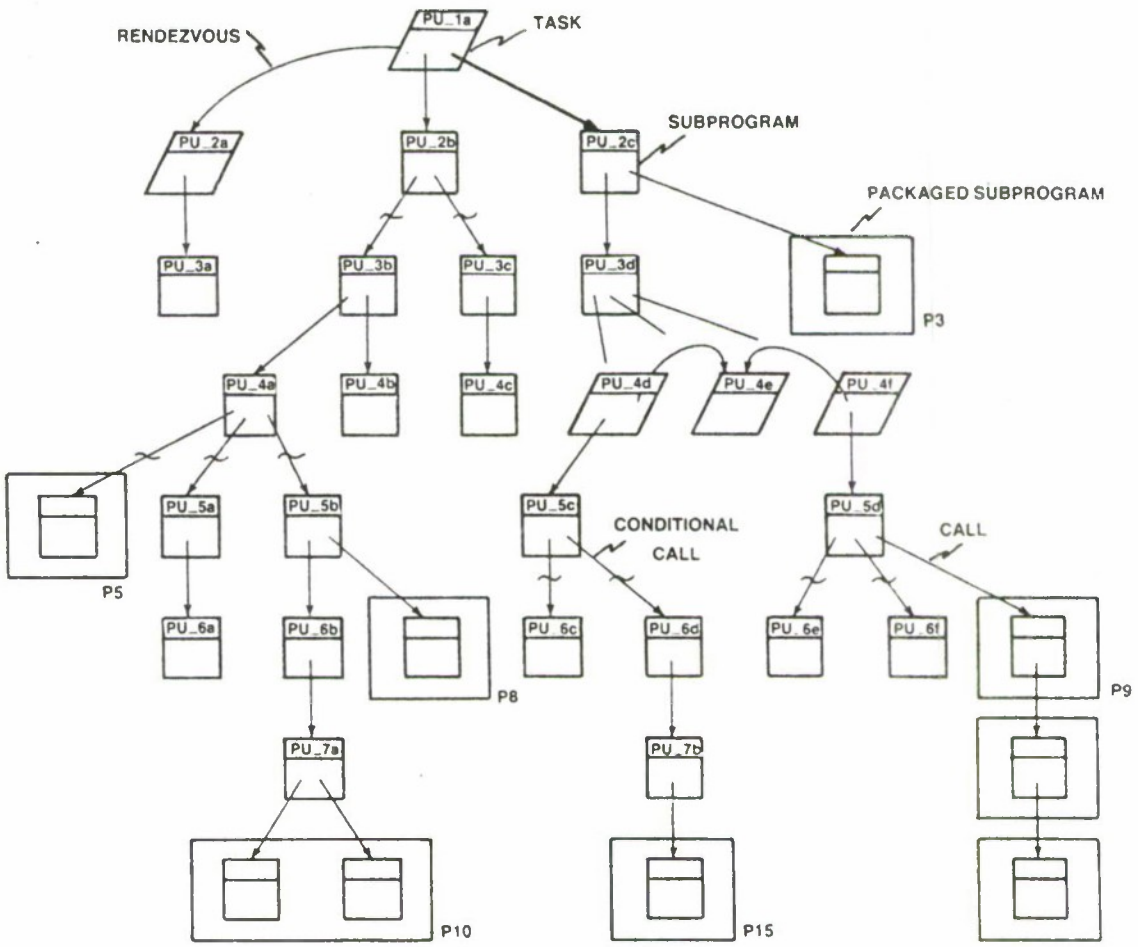


FIGURE 10. INVOCATION DIAGRAM (OPTION D)

An Invocation Diagram utilizes the pictographs defined by SHARP. However, in contrast with a Hierarchy Diagram, an Invocation Diagram directly indicates calls to program units contained in package specifications. A called subprogram or task belonging to a package is represented by a bounded pictograph, as shown in Item a of Figure 11.

As described in subsequent paragraphs, Invocation Diagrams represent the calling sequence of program units, recursive processes, and conditional subprogram calls. In addition to representing calls to program units declared in packages, Invocation Diagrams also represent other Ada unique characteristics such as:

- Task rendezvous
- Task activation
- Generic program units

2.3.2a Representing a Sequence of Program Unit Calls

Within an Invocation Diagram, arrows are drawn to the specifications of a called subprogram from the bodies of the calling units, as shown in Figure 11.

With the exception of program units involved in recursive processes or loops, if a program unit is called n times during the execution of a program, the program unit must be shown n times in the Invocation Diagram.

2.3.2b Representing Conditional Subprogram Calls

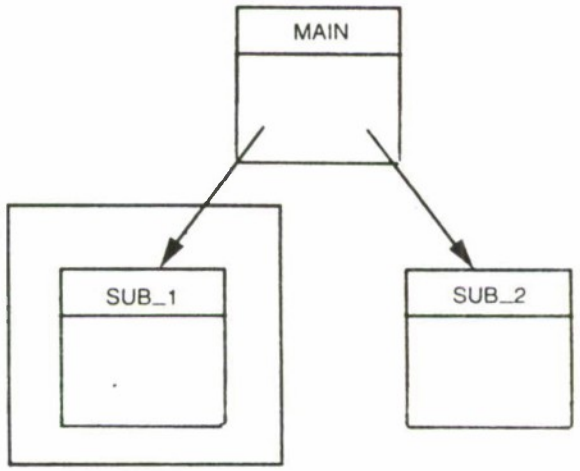
A call to a subprogram or task depending upon some transient condition (e.g., a 'select,' 'accept,' 'if' or 'case' statement) is pictorially represented in an Invocation Diagram. Specifically, the existence of the transient condition is indicated by a tilde placed on the arrow representing a program unit call, as illustrated in Item b of Figure 11.

2.3.2c Representing Recursive Processes

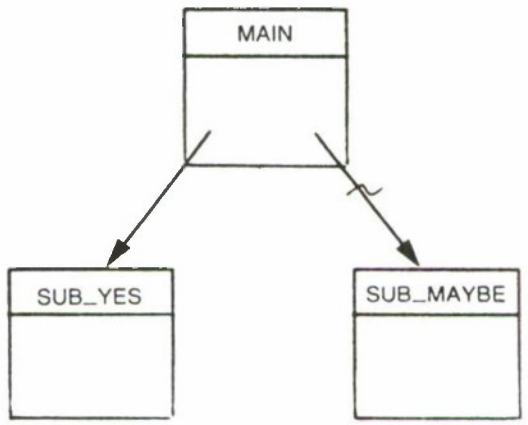
A program unit which calls itself recursively is represented by a semi-circular arrow, beginning at the bottom of the pictograph representing the program unit and ending on its side, as shown in Item a of Figure 12. Two program units that call each other recursively are flagged by asterisks placed adjacent to a double arrow, as shown in Item b of Figure 12. More than two program units involved in a recursive process are flagged by asterisks adjacent to a "feedback loop,"as shown in Item c of Figure 12.

2.3.2d Representing Task Rendezvous

Task rendezvous is represented by a curved arrow from the body of the calling task (or from a circle containing "H/W" if a hardware interrupt) to the specification of the acceptor task, as shown in Figure 13. Details of task rendezvous are pictorially represented in a Task Rendezvous Diagram, as described in Section 2.6.

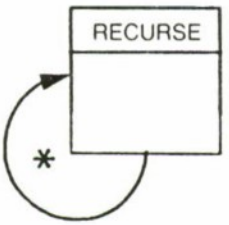


(a) Representing a Called Program Unit Belonging to a Package

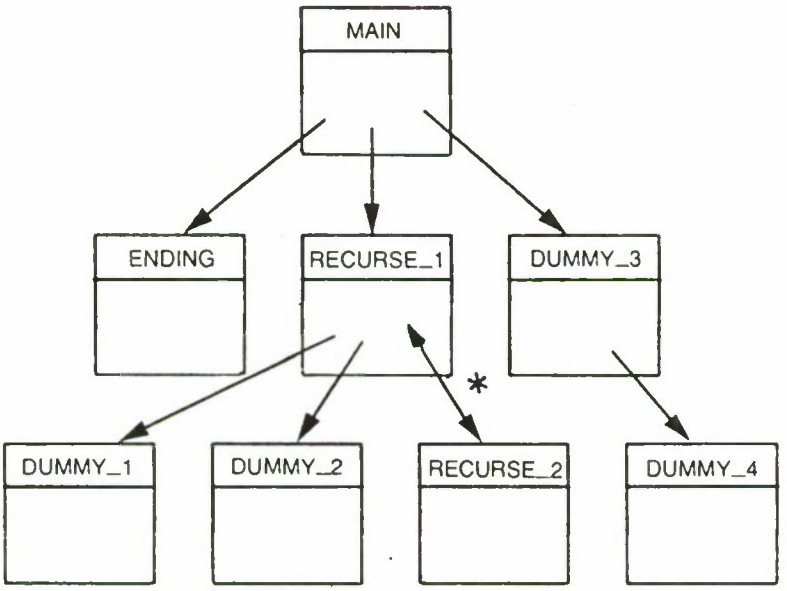


(b) Representing Conditional Subprogram Calls

FIGURE 11 REPRESENTING CALLS TO SUBPROGRAMS

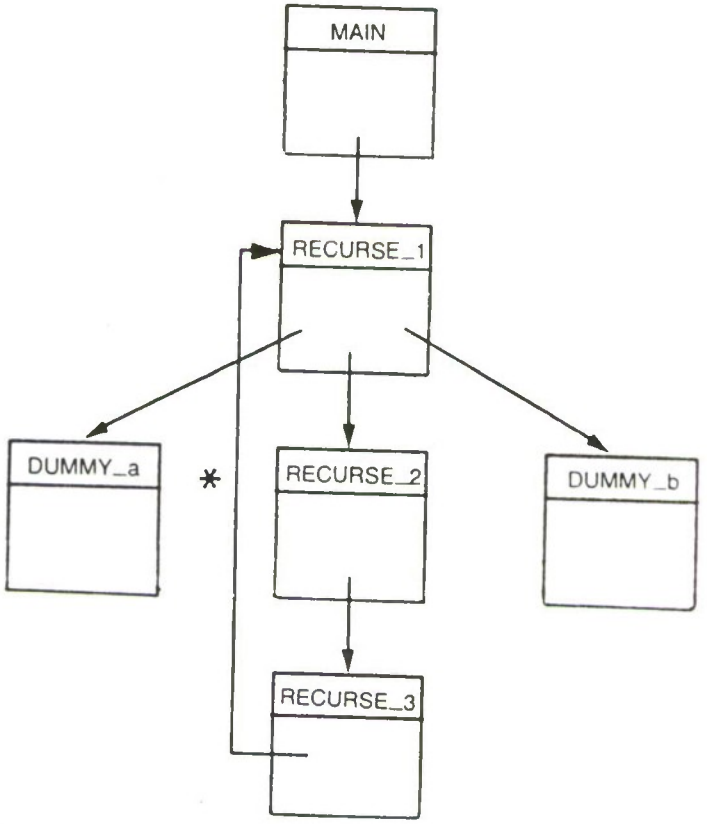


(a) Representing a Single Recursive Program Unit



(b) Representing Two Recursive Program Units

FIGURE 12 REPRESENTING RECURSIVE SUBPROGRAM CALLS



(c) Representing Recursive Program Unit Calls with More Than Two Subprograms

FIGURE 12 (CONCLUDED)

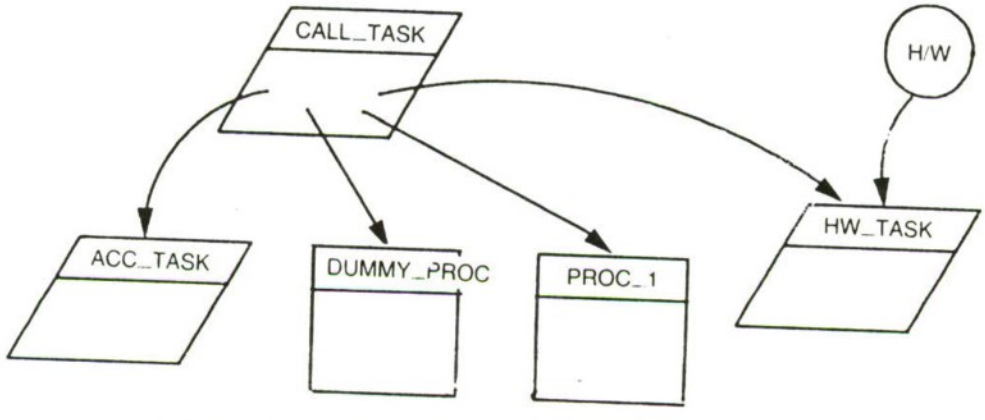


FIGURE 13 REPRESENTING TASK RENDEZVOUS IN AN INVOCATION DIAGRAM

2.3.2.e Pictograph for a Generic Subprogram

A generic subprogram is represented by a square with a dashed line used to establish the narrow part representing the subprogram's specification, as shown in Figure 14.

2.4 LOWER LEVEL SHARP ABSTRACTS

2.4.1 Subprogram Data Flow Diagram (Option E)

2.4.1a Purpose and Background Information

A Subprogram Data Flow Diagram represents data flow between a subject program unit and a specified caller. As background information, various diagrams have been used to represent data flow. For example, IBM has developed HIPO charts (i.e., Hierarchical Input, Processing, and Output), as illustrated in Item a of Figure 15. The wide arrow into and out of the center rectangle accounts for inputs and outputs, and the center rectangle accounts for processing performed on the input to produce the output.

Item b of Figure 15 shows another example of a Data Flow Diagram. Arrows show the data flow and the circles indicate the type of processing to be performed on the data.

2.4.1b Description

The Data Flow Diagram of Advanced SHARP pictorially represents data flow between a subject subprogram and a specified caller, by the modes established in the subject subprogram's specification. In Ada, the specification defines the mode of parameter passing, which is one of the following:

- 'in' (i.e., the value of a parameter is received and not modified)
- 'out' (i.e., the value of a parameter is created and exported)
- 'in out' (i.e., the value of a parameter is received, modified and exported).

As an example of a procedure specification, consider the following:

```
procedure  SAMPLE  (PAR1:  in INTEGER
                    PAR2:  in FLOAT
                    PAR3:  in out FLOAT
                    PAR4:  in out FLOAT
                    PAR5:  in out INTEGER
                    PAR6:  out FLOAT
                    PAR7:  out INTEGER) is
```

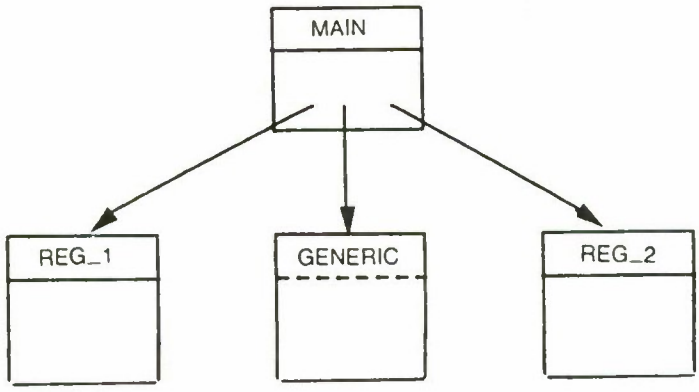
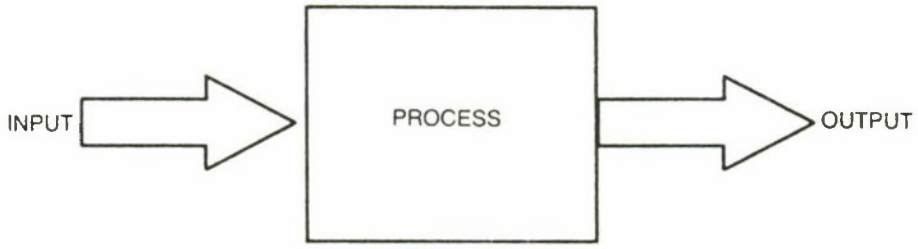
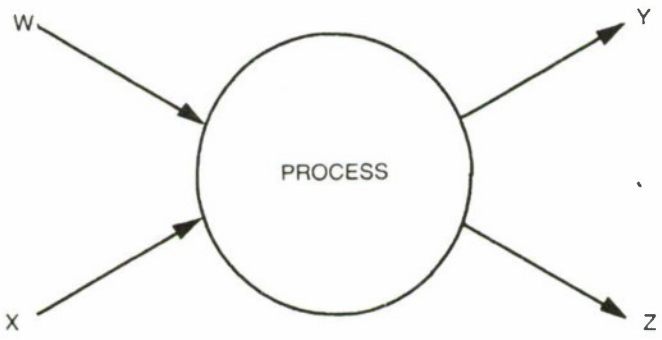


FIGURE 14 REPRESENTING GENERIC PROGRAM UNITS



(a) HIPO Charts



(b) Data Flow Representation

FIGURE 15. EXAMPLES OF TRADITIONAL DATA FLOW DIAGRAMS

Here a procedure named SAMPLE receives input parameters PAR1 and PAR2, which are not modified; receives input parameters PAR3, PAR4, and PAR5, which are altered and exported; and exports parameters PAR6 and PAR7, which have been created. With SHARP, this data flow is pictorially represented by a calling subprogram above the called subprogram, as shown in Figure 16.

Parameters received (i.e., the **in** mode) are shown as shaded circles on a directed line pointing to the called subprogram from the calling program unit. Parameters to be exported (i.e., the **out** mode) are shown as shaded circles on a directed line pointing to the calling program unit. Parameters received, modified and exported (i.e., the **in out** mode) are shown as shaded circles on a directed line pointing to both the calling program unit and the called subprogram.

2.4.2 Representing a Generic Subprogram in a Data Flow Diagram (Option E)

If the parameter being passed has a generic type, the circle on the directed line is not shaded, as illustrated in Figure 17. As an example, consider the procedure that exchanges two elements with a generic type:

```
generic
type ELEMENT is private
procedure EXCHANGER (FIRST, SECOND: in out ELEMENT);
procedure EXCHANGER (FIRST, SECOND: in out ELEMENT) is
  TEMPORARY:  ELEMENT
  begin
  TEMPORARY:=FIRST;
  FIRST:=SECOND;
  SECOND:=TEMPORARY;
end EXCHANGER;
```

The specific name of the generic procedure and the definition of type ELEMENT must be created prior to use of this procedure, which is referred to as instantiation. We may declare several instances of the generic program unit, as illustrated by the following:

```
o    procedure INTEGER_EXCHANGE is new EXCHANGER (ELEMENT=>INTEGER);

o    procedure FLOAT_EXCHANGE is new EXCHANGER  (ELEMENT=>FLOAT);
```

If these instantiations were in a calling procedure named CALLER, the Advanced SHARP Data Flow Diagram would be given as shown in Figure 17.
In addition to making the type of a passed parameter generic, Ada also permits the range of values permissible for a passed parameter to be generic for the **in** and **in out** modes. (The mode **out** cannot be used with a generic parameter.) For example, consider the following generic procedure:

```
generic
     ROW     :  in INTEGER:= 24;
     COLUMNS:  in INTEGER:= 80;
procedure MATRIX is
     o
     o
     o
end MATRIX;
```

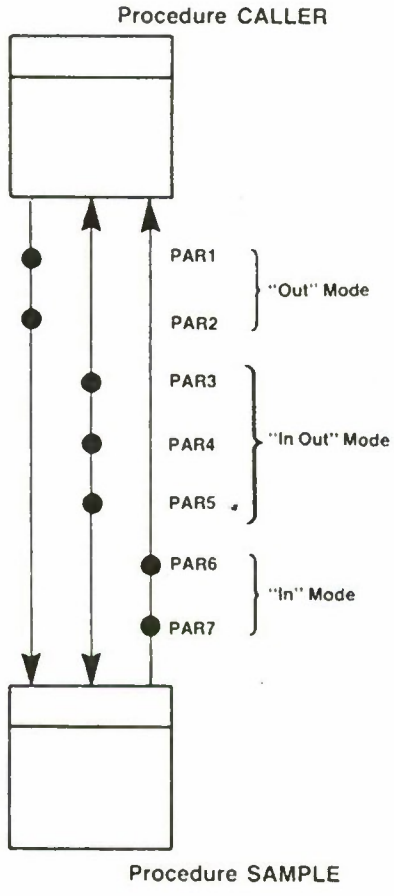


FIGURE 16. SUBPROGRAM DATA FLOW DIAGRAM (OPTION F)

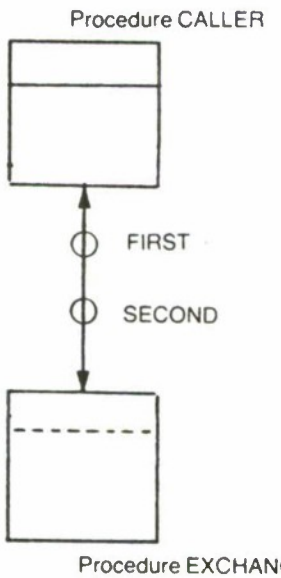


FIGURE 17. GENERIC PROCEDURE "EXCHANGER"

It can have several instances, such as

- o procedure SMALL_MATRIX is new MATRIX (ROWS=>5, COLUMNS=>10);
- o procedure LARGE_MATRIX is new MATRIX (ROWS=>130, COLUMNS=>230);

If these instantiations were in the calling procedure *MATRIX_CALL*, the Advanced SHARP Data Flow Diagram would be given as shown in Figure 18.

2.4.3 Task Rendezvous Diagram (Option F)

2.4.3a Purpose

The complexities of inter-task relationships may become so intricate that software quickly becomes difficult to understand and costly to maintain. Task Rendezvous Diagrams provide insight into such complicated task rendezvous.

Invocation Diagrams identify calling and acceptor tasks. However, these diagrams do not represent task entry points and give no information about the nature of the entry call (conditional, unconditional or time-conditional) or conditions of its acceptance. Task Rendezvous Diagrams supplement Invocation Diagrams by pictorially supplying this information.

2.4.3b Pictographs Used in a Task Rendezvous Diagram

A Task Rendezvous Diagram utilizes the SHARP pictograph for a task, the parallelogram; a pictorial representation for task entries, a small parallelogram which overlaps the representation of a task's specification and body; and a representation of interrupts from hardware, the circle. These pictographs are shown in Item a of Figure 19.

2.4.3c Representing Task Entry Points

A rendezvous between two tasks is initiated by one task calling an entry declared in another. As is the case for a procedure, parameters to be passed can be of the **in**, **out**, or **in out** modes.

If a task has many entry points (i.e., more than three), they may be represented as one long parallelogram with several lines drawn through it. In this case, the names of all entry points are shown adjacent to the task, as illustrated in Item b of Figure 19.

2.4.d Representing Access to Task Entry Points

Task entry calls are shown by three arrows drawn between the calling task and the acceptor task. Circles on these arrows represent parameters being transferred. Like the data flow diagram, each of the arrows identifies parameters which are either **in**, **out**, or **in out**. The nature of the parameters may be derived from the direction of the arrowhead(s), as explained in Paragraph 2.4.1b.

Item a of Figure 20 illustrates the following task:

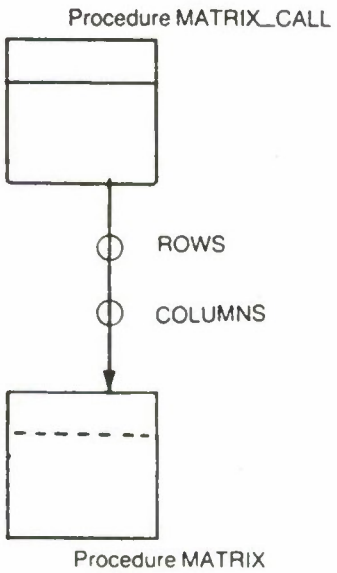
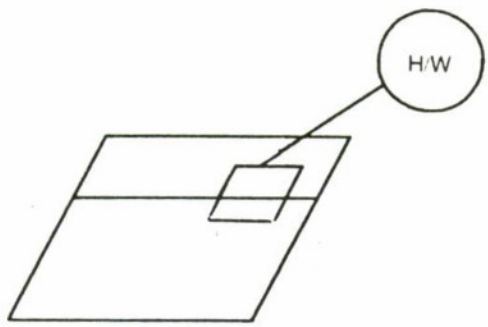
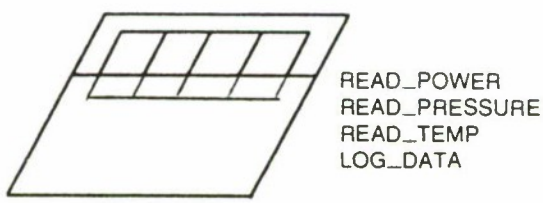


FIGURE 18. GENERIC PROCEDURE "MATRIX"

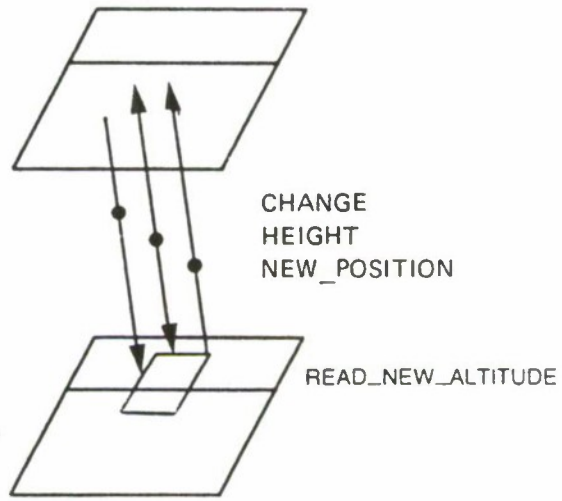


(a) PICTOGRAPHS USED IN A TASK RENDEZVOUS DIAGRAM

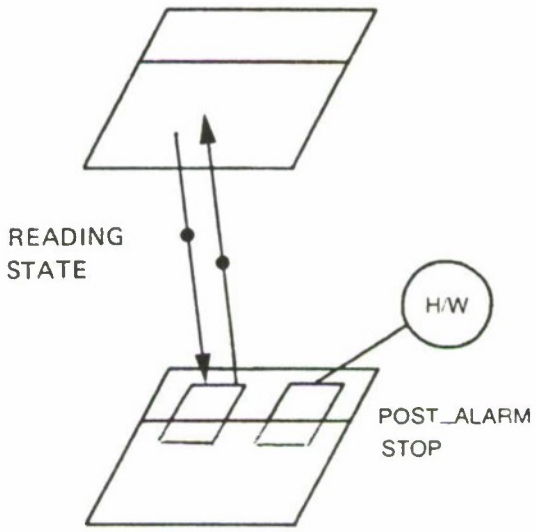


(b) REPRESENTING TASK ENTRY POINTS

FIGURE 19. TASK RENDEZVOUS DIAGRAM



(a) Representing Access to Task Entry Points (all modes)



(b) Representing Access to Task Entry Points (subset of modes)

FIGURE 20. RENDEZVOUS ENTRY POINTS

```
task ALTITUDE_CORRECT is
  entry READ_NEW_ALTITUDE (CHANGE: in REAL;
                           HEIGHT: in out REAL;
                      NEW_POSITION : out DIMENSION);
end ALTITUDE_CORRECTION;
    . . .
```

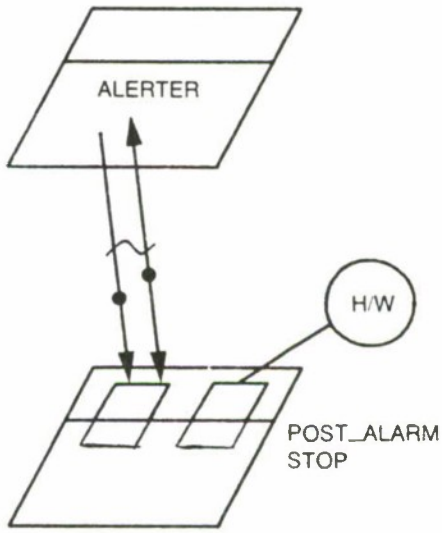
A task entry which does not have all three parameter modes may be pictorially represented with only one or two arrows. A task entry with no parameters may be represented by a straight line connecting it and the calling task. A task entry with only the **in** and **out** mode is shown in Item b of Figure 20 and the following example:

```
    . . .
task ALARM is
   entry POST_ALARM (READING:  in SENSOR_READINGS;
                     STATE   :  out STATUS);
   entry STOP;
end ALARM;
    . . .
```

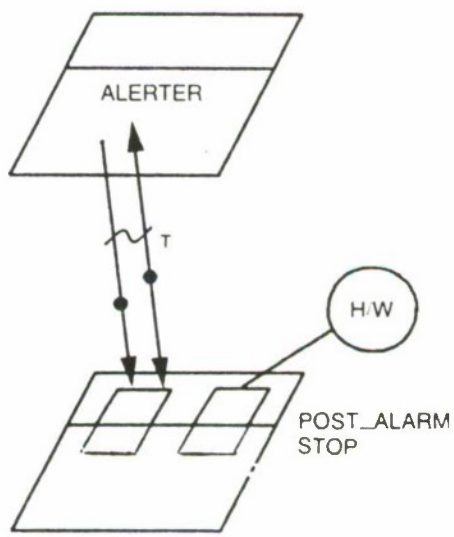
Thus far, we have been discussing unconditional entry calls; that is, patient caller tasks which will wait indefinitely for a rendezvous. With SHARP, we can also represent conditional and time-conditional task entry calls. A conditional entry call occurs when the calling task requests a rendezvous, receives no immediate response from the acceptor task and, therefore, takes an alternative action. A conditional task entry call is represented by a tilde placed on the appropriate arrow connecting the calling and acceptor tasks. Item a of Figure 21 illustrates the conditional call for the following example:

```
    . . .
task body ALERTER is
    . . .
  select
    ALARM.POST_ALARM (...);
  else
    . . .  --  some alternative action
  end select;
    . . .
end ALERTER;
```

A time-conditional call occurs when the caller task requests a rendezvous and waits T units of time for the rendezvous to occur; if there is no response, alternative action is taken. The time-conditional call is pictorially represented like the conditional call, with a 'T' adjacent to the tilde. Item b of Figure 21 illustrates the time-conditional call in the following example:



(a) Representing a Conditional Task Entry Call



(b) Representing a Time Conditional Task Entry Call

FIGURE 21. CONDITIONAL TASK ENTRY CALLS

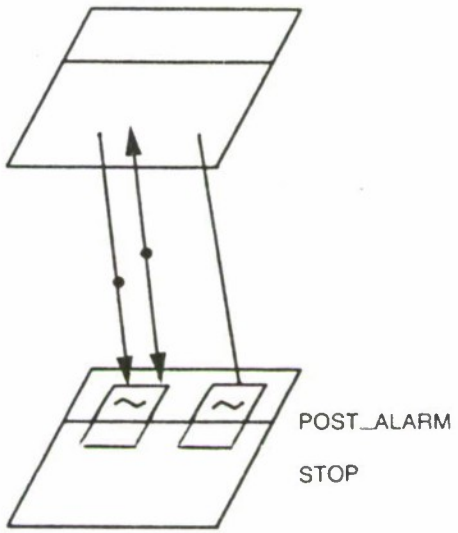
```
     . . .
task body ALERTER is
     . . .
  select
    ALARM.POST_ALARM (...);
    or delay T;
  end select;
     . . .
end ALERTER;
```

2.4.3e Representing Acceptance of a Task Call

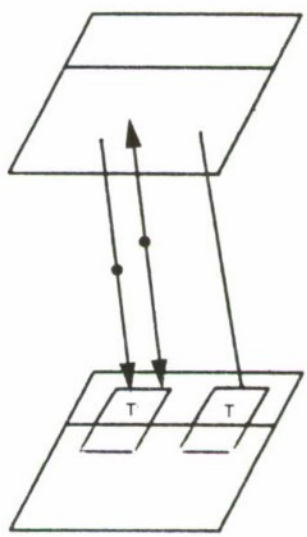
Like the caller task, the acceptor task may have conditions associated with a rendezvous. When this is the case, the entry shall be pictorially represented by a tilde inside the entry representation (within the task's body). Item a of Figure 22, and the following example, illustrate this situation:

```
. . .
task body ALARM is
. . .
  select
    when X =>
      accept POST_ALARM (...) do
. . .
      end;
  or
    when Y =>
      accept STOP do
. . .
      end;
  end select;
. . .
end ALARM;
```

Also, like the caller task, the acceptor task may be time-conditioned. In this case, affected entries shall be pictorially represented by a 'T' inside the entry representation (within the task's body). Item b of Figure 22 and the following example illustrate this case.



(a) Representing Conditional Task Acceptance



(b) Representing Time Conditional Task Acceptance

FIGURE 22. CONDITIONAL TASK ACCEPTANCE

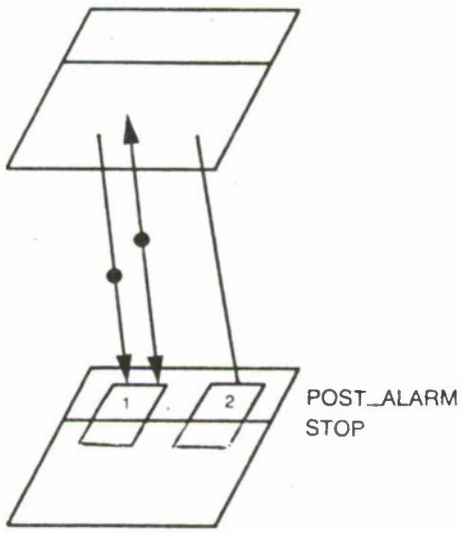
```
        . . .
task body ALARM is
     . . .
select
     accept POST_ALARM (...) do
     . . .
     end;
  or
    accept STOP do
          . . .
          end;
      or
          delay T;          -- timeout
end select;
     . . .
end ALARM;
```

In addition to the above two cases, acceptances of entry calls may be in fixed order (specified) or in time order (first come, first serve). Entries accepted in fixed order are represented by numbers within the affected entry points where '1' indicates first, as illustrated in Item a of Figure 23 and the following example:

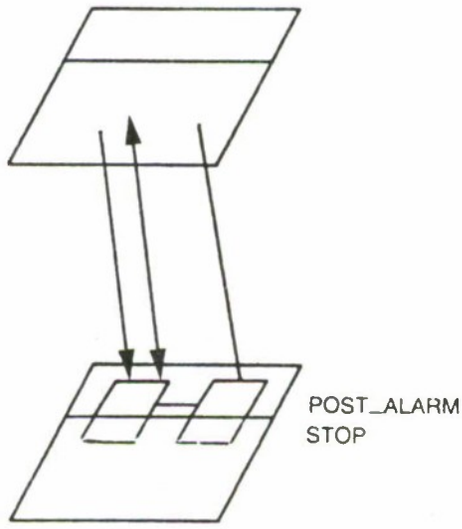
```
     . . .
     task body ALARM is
     . . .
        accept POST_ALARM (...);
     . . .
        accept STOP;
     . . .
     end ALARM;
```

Acceptance on a first arrival basis is represented by a line connecting the affected entry points, as illustrated in Item b of Figure 23, and the following example:

```
     . . .
     task body ALARM is
     . . .
       select
         accept POST_ALARM (...) do
     . . .
         end;
       or
         accept STOP do
     . . .
         end;
       end select;
     end ALARM;
```



(a) Representing Acceptance on a Fixed Order Basis



(b) Representing Acceptance on a First Arrival Basis

FIGURE 23. ORDER OF ENTRY CALL ACCEPTANCE

2.4.4 Data Structure Diagram (Option G)

2.4.4a Purpose

Data Structure Diagrams pictorially present an abstracted representation of type, constant and variable declarations within a package, and their visibility. These diagrams are a useful tool to a designer when trying to utilize the mechanism of information hiding to control dependency relationships between program units (e.g., for software maintainability and versatility) or to encapsulate security-critical software.

Several degrees of information hiding may be achieved by varying the use of package specifications and bodies, and through the use of private and limited private types. The Data Structure Diagram provides a pictorial representation which clearly illustrates visible, hidden, and private declarations within a package.

A Data Structure Diagram also distinguishes between discrete types (i.e., numbers and characters) from structured types (i.e., arrays and records); and represents discriminate types, access types and task types.

As shown in Figure 24, declarations of types, constants and variables are represented as follows:

- Upright narrow geometric entities for type declarations
- Right-slanted geometric entities for constant declarations
- Left-slanted geometric entities for variable declarations

2.4.4b Representing Items Declared in the Package Specification

Items may be declared and decomposed in the specification of a package. These items are therefore visible and available to the user. The user may access visible entities, utilizing either dot notation or the Ada 'use' clause. Pictorially, visible declarations are represented as unshaded versions of the geometric entities used to represent types, constants and variables. A number placed directly above each entity, is associated with a declaration name, as illustrated in Figures 24.

2.4.4c Representing Items Declared Private or Limited Private

Items can be declared private or limited private in the visible part of the package specification and refined in the private part. This restricts the use of such items. Specifically, if an item is **private**, the user can only apply to it operations defined within the package specification, and assignments and tests for equality. If an item is **limited private**, assignments and tests for equality are no longer automatically available. Pictorially, such declarations are represented by partially shaded geometric figures, as illustrated in Figure 25.

TYPE DECLARATIONS

1. AXIS_PAIRS
2. PRIORITY
3. BUFFERING
4. ALERT
5. NATURAL
6. DEGREES

CONSTANT DECLARATIONS

1. MAX
2. MIN
3. PROB_IN_POWER
4. PROB_OUT_POWER
5. PROB_IN_PRESSURE
6. PROB_OUT_PRESSURE
7. PROB_IN_TEMP
8. PROB_OUT_TEMP
9. ANGLE_1
10. ANGLE_2
11. POWER_VALUE

VARIABLE DECLARATIONS

1. ENVSIM
2. COUNTER
3. CURRENT_AXES
4. NEXT_AXES
5. RANDOM_NUM
6. DELTA_POWER
7. DELTA_PRESSURE
8. UPPER_POWER
9. UPPER_PRESSURE
10. LOWER_POWER
11. LOWER_PRESSURE
12. UPPER_TEMP
13. LOWER_TEMP
14. CURRENT_STATE
15. CURRENT_VALUE

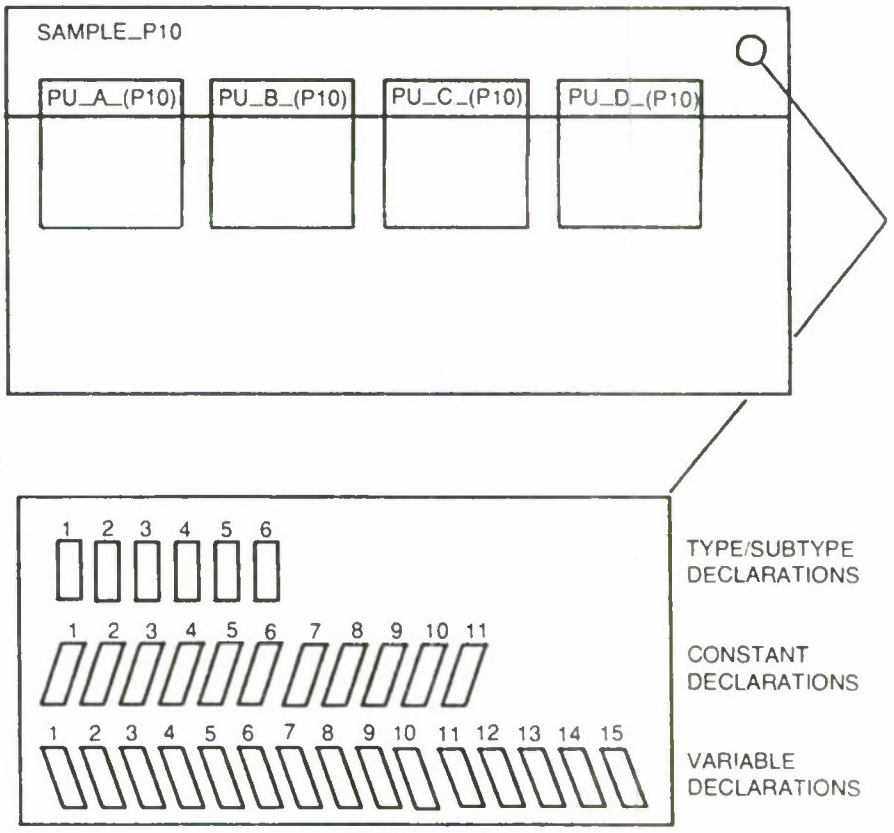


FIGURE 24. VISIBLE DECLARATIONS IN A PACKAGE SPECIFICATION

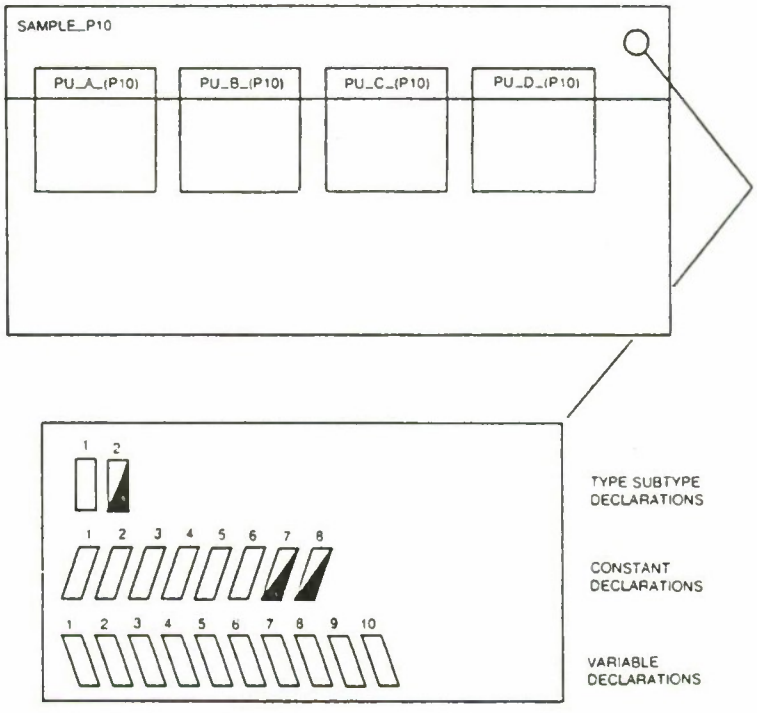


FIGURE 25. VISIBLE AND PRIVATE DECLARATIONS IN A PACKAGE SPECIFICATION

2.4.4d Representing Items Declared in the Package Body

Entities declared within the body of a package are inaccessible to program units external to the package. Such hidden declarations are represented by shaded geometric figures, as shown in Figures 26.

At design reviews or when teaching Ada, we may want to select other options for showing the visibility of declared entities. For example,

- Magnification of both the body and specification of a package can be used to show data structure detail of both hidden and visible entities declared directly in the package (i.e., not under any other program unit), as illustrated in Figures 27.

- Magnification of a package body and the bodies of program units within the package, to show all items declared within a package body (both directly and under a program unit), as illustrated in Figures 28.

- A combination of the previous options to show the visibility of all visible and hidden items declared both in the package directly and within all program units contained within the package, as shown in Figures 29.

2.4.5 Representation of Types

2.4.5a Representing Discrete Types

A discrete type, or scalar type, defines a set of values that have no components. They include integer, real and enumeration types. As shown in Item a of Figure 30, an integer type is represented by an upright narrow rectangle with the letter "I" underneath it; a real type with the letters "RL"; and an enumeration type with the letters "EN".

2.4.5b Representing an Array type

An array is a collection of components, with each component of the same type. An array type is represented by an upright narrow rectangle with the letters "AR" underneath it, as illustrated in Item b of Figure 30.

2.4.5c Representing a Record Type

A record is also a collection of components. However, records differ from arrays in that not all components must be of the same type. Also, the selection of a record component is always static and determinable at the compile time, whereas an array component can be made dynamically at run time (by evaluating an expression denoting an index value). A record type is represented by an upright narrow rectangle with the letter "RC" underneath it, as illustrated in Item b of Figure 30.

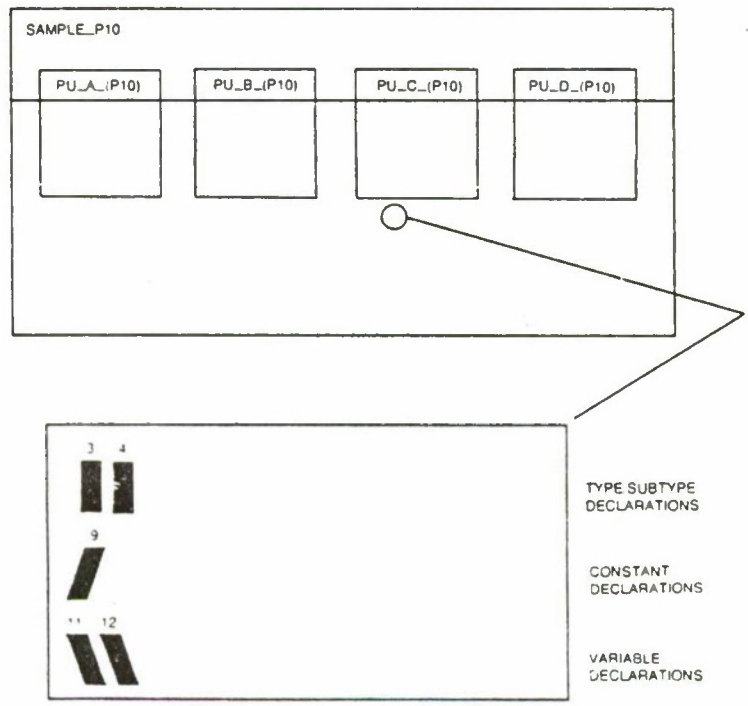


FIGURE 26. NON-VISIBLE DECLARATIONS IN A PACKAGE BODY

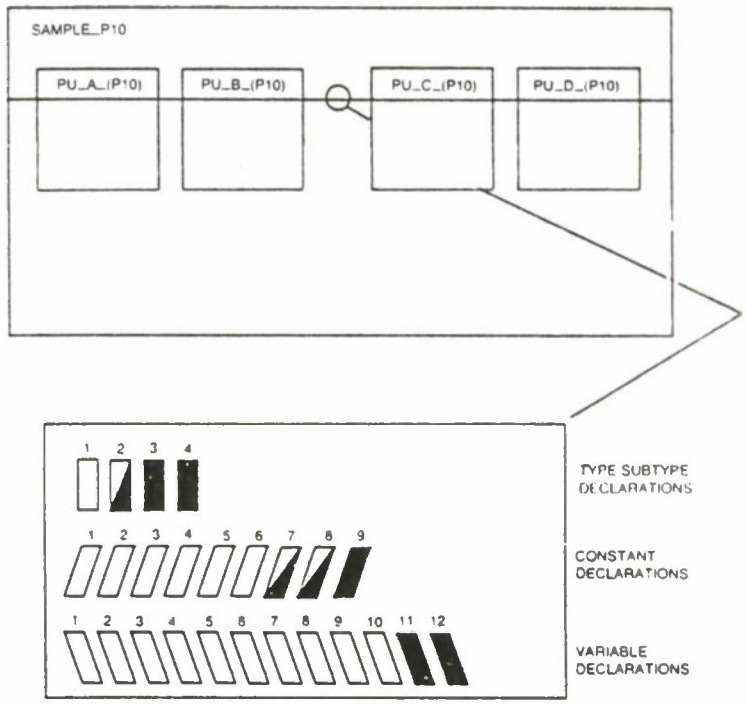


FIGURE 27. REPRESENTING VISIBLE DECLARATIONS IN A PACKAGE SPECIFICATION IN CONJUNCTION WITH NON-VISIBLE DECLARATIONS IN A PACKAGE BODY

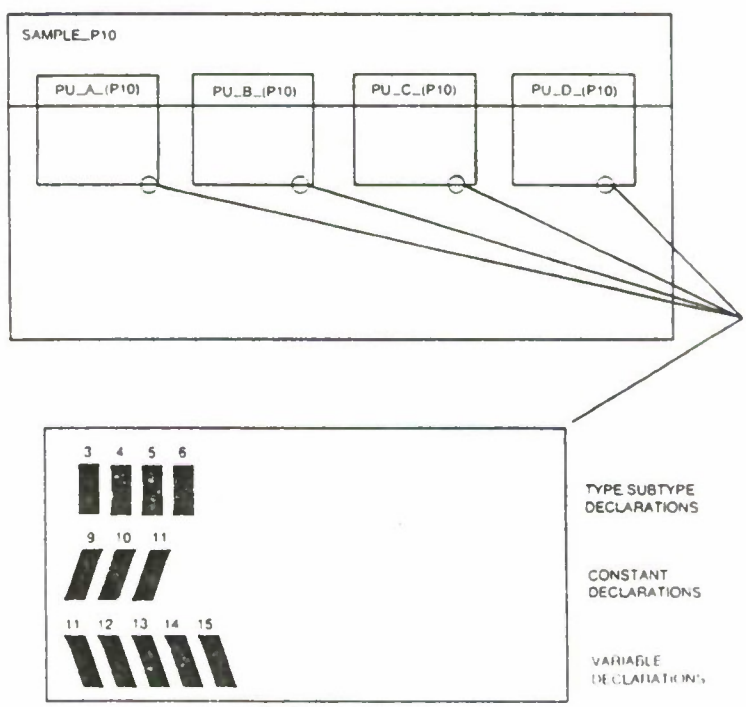


FIGURE 28. REPRESENTING NON-VISIBLE DECLARATIONS IN A PACKAGE BODY AND THE BODY OF A PROCEDURE CONTAINED WITHIN THE PACKAGE

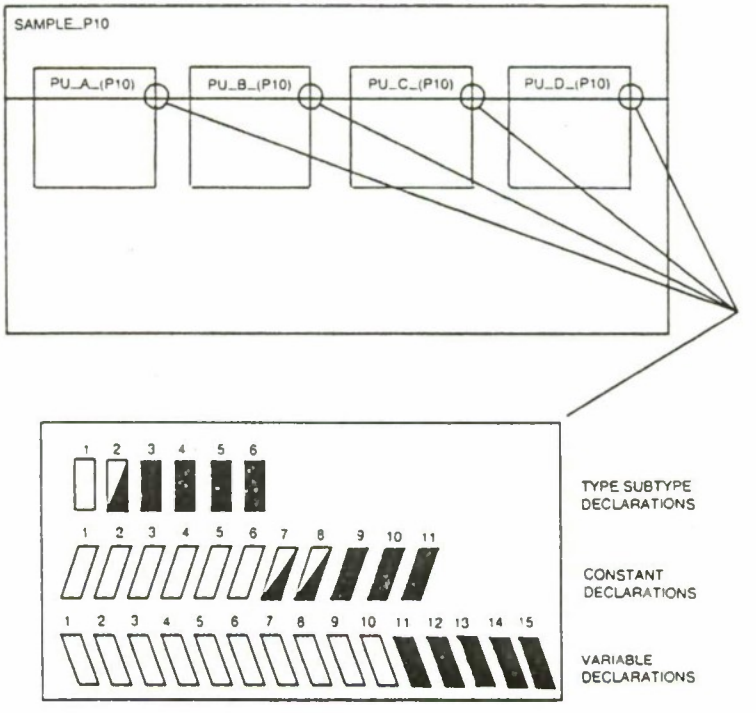
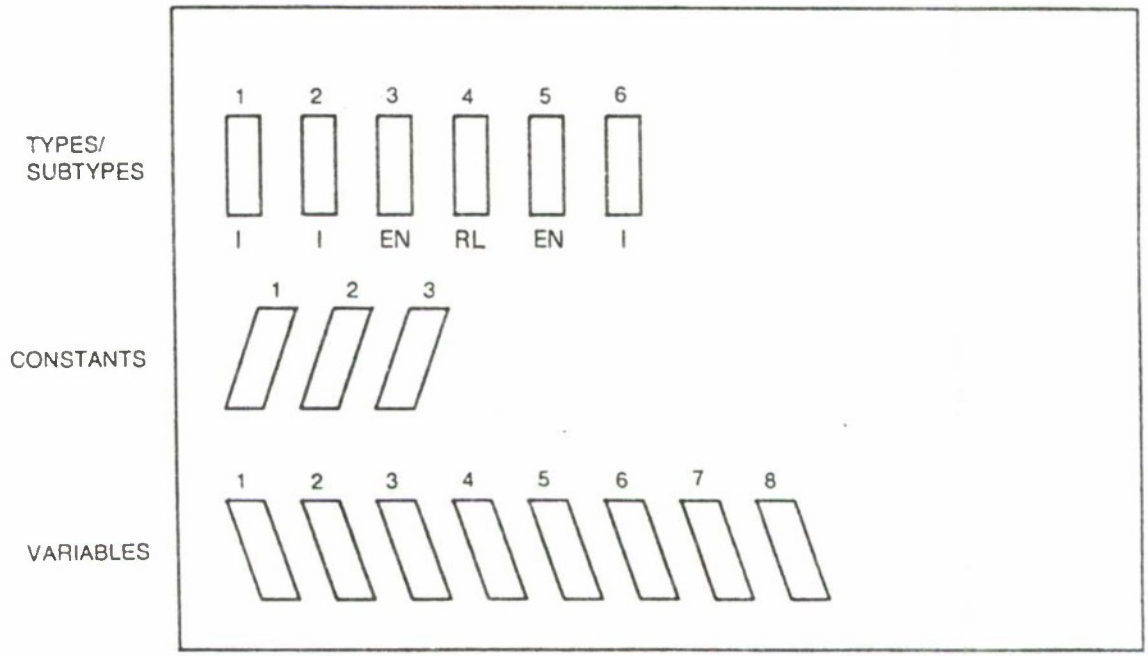
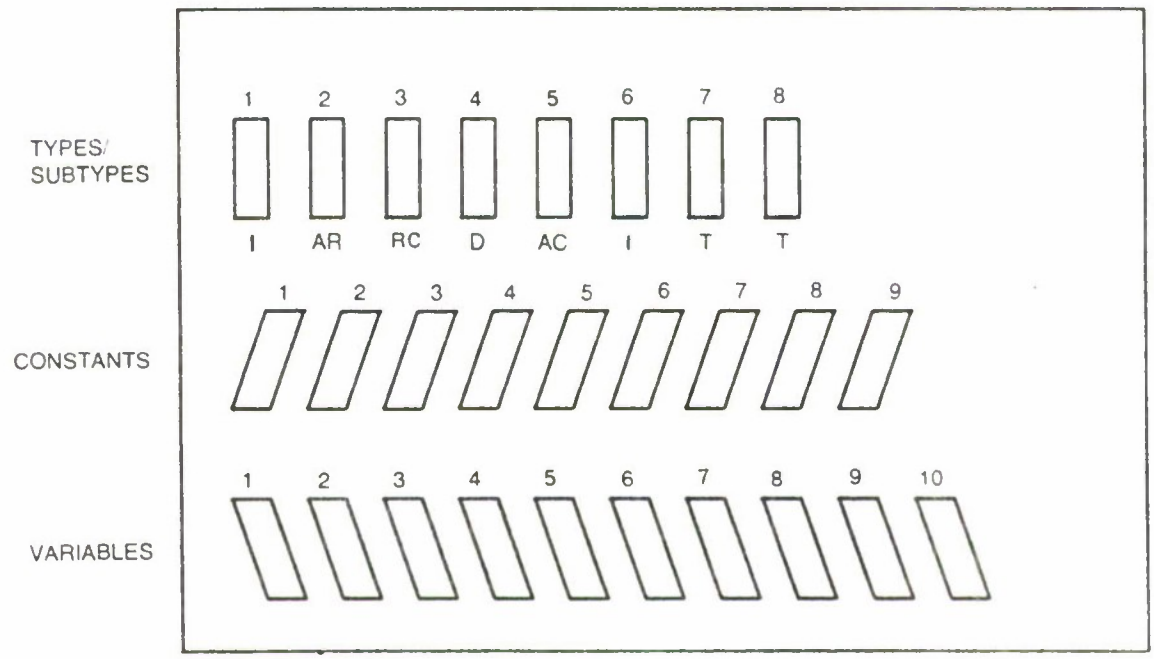


FIGURE 29. REPRESENTING ALL VISIBLE AND NON-VISIBLE DECLARATIONS WITHIN A PACKAGE



(a) REPRESENTING AN INTEGER TYPE, REAL TYPE AND ENUMERATION TYPE



(b) REPRESENTING AN ARRAY TYPE, RECORD TYPE, DISCRIMINATED TYPE, ACCESS TYPE AND TASK TYPE

FIGURE 30. REPRESENTING TYPES

2.4.5d Representing a Discriminated Type

Ada also facilitates a discriminated types, where variables of this type are called "discriminants." A record with discriminants may have (a) variant parts in which certain components are present only for certain values of the discriminant, and (b) array components whose bounds are fixed by the values of discriminants. Conceptually, a discriminated type can be thought of as introducing a set of record values, each value in the set having a different structure. A discriminated type is represented by an upright narrow rectangle with the letter "D" underneath it, as illustrated in Item b of Figure 30.

2.4.5e Representing an Access type

As an alternative to statically allocated data, Ada provides a mechanism for allocating variables dynamically during program execution. Since the storage locations used for dynamic variables are not determined in advance, they cannot be referenced by a name but must instead be referenced indirectly via a so-called access type. Unknown amounts of data can be handled by dynamically allocating storage to each new datum when it is received. In this way, complex data structures can be built with components dynamically allocated. An access type is represented by an upright narrow rectangle with the letters "AC" underneath it, as illustrated in Item b of Figure 30.

2.4.5f Representing a Task Type

A task type is formed when the keyword task is followed by the keyword type. Elaboration of the corresponding task body defines what a task of that type does. It does not cause a task to be activated. Rather, tasks are activated separately by declaring variables of the task type. A task type is represented by an upright narrow rectangle with the letter "T" underneath it, as illustrated in Item b of Figure 30.

2.4.5g Representing the Type of a Variable and Constant

The type of a variable or constant can be represented as follows:

- If the type is predefined, then the first letter of the type (e.g., I for INTEGER) is placed under the geometric representation of the variable or constant.

- If the type is defined, the letter "T" followed by the type glossary number is placed under the geometric representation of the variable or constant.

For example, in Figure 31, variables number 1 and 3 are of the predefined type INTEGER, variables number 2 and 4 are of the predefined type BOOLEAN, variable 5 is of defined type T1, variables 6 and 8 are of defined type T2, and so forth.

2.5 LOWEST LEVEL SHARP ABSTRACTS

At a low level of abstraction, SHARP junctions with traditional design representation using Ada-based pseudo code and a glossary of data structure detail. This is the lowest level of design abstraction with SHARP.

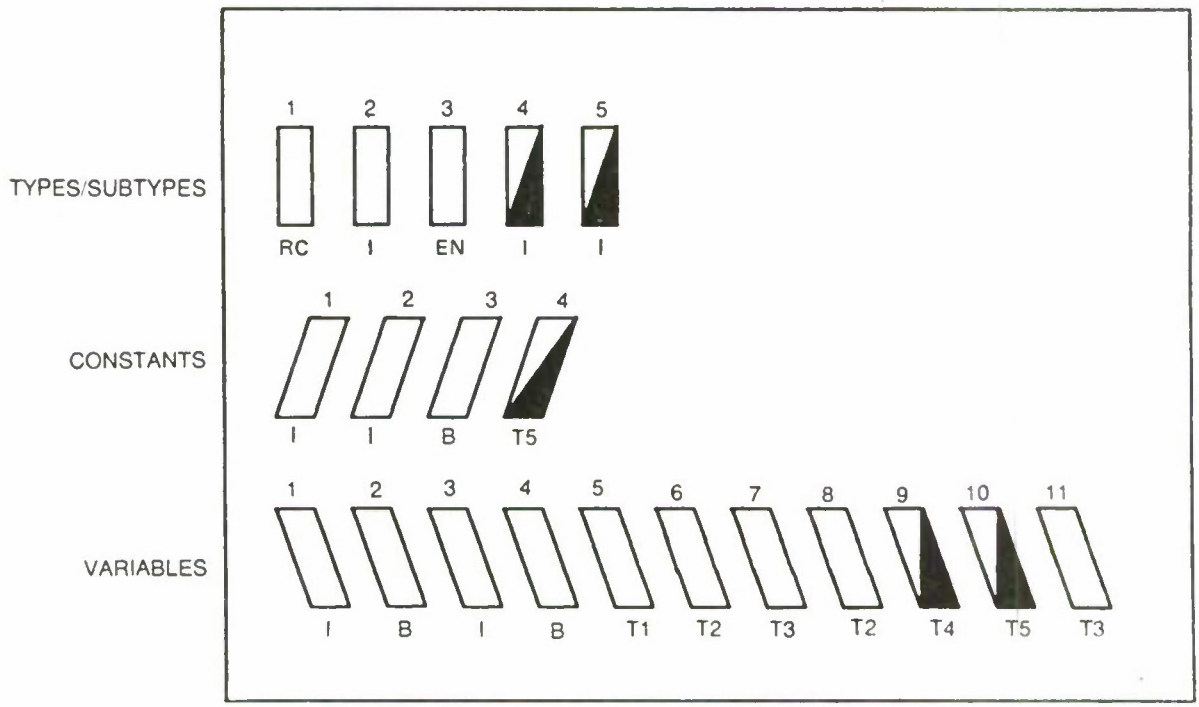
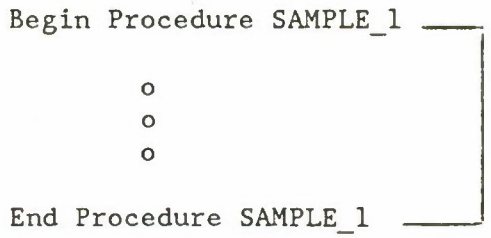


FIGURE 31. REPRESENTING THE TYPE OF A VARIABLE AND CONSTANT

2.5.1 Annotated Ada-Based Pseudo Code (Option H)

SHARP utilizes annotated pseudo code to represent the design of program unit bodies. SHARP criteria include general standards for the pseudo code and its annotation. Standards of the criteria require that the name of the subject program unit is clearly shown, and the beginning and end of the pseudo code for the subject program unit is bracketed as follows:



In addition, the standards require the pseudo code to account for the following:

- Logic and decisions
- Algorithms
- Program unit calls and I/O
- Generic instantiation
- Exception handling

The standards require that the design associated with these factors must be presented using certain Ada key words and annotation, as described in the following paragraphs.

2.5.1a Logic and Decisions

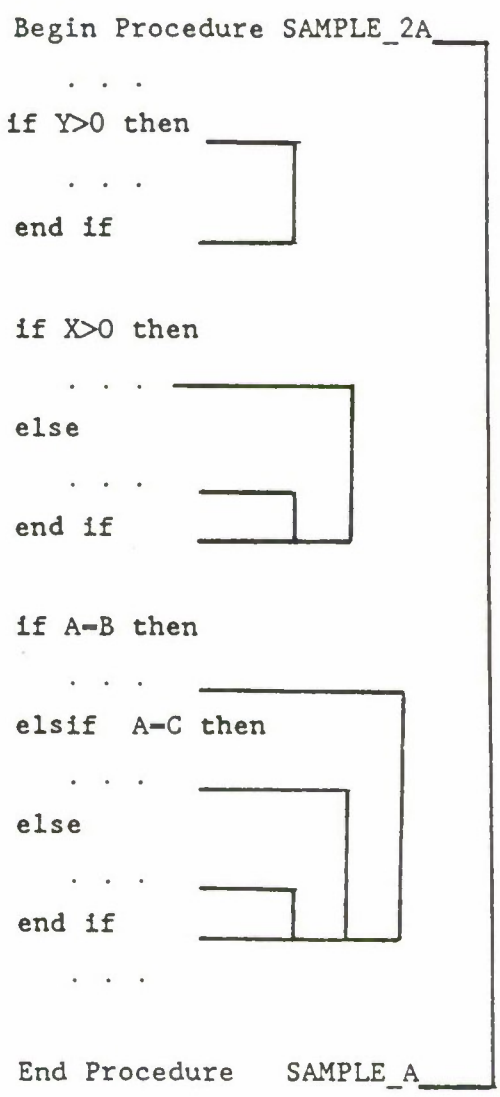
SHARP criteria states that Ada control statements must be used to represent the design for logic and decisions made in the body of a program unit. For example, the **if** and **case** statements are used to provide conditional control (i.e., the selection of one of a number of alternate actions).

The **if** statement selects a course of action depending upon the truth value of one or more conditions. In Ada, there are three basic forms of the **if** statement:

- **if-then**
- **if-then-else**
- **if-then-elsif**

In each case, the **if** statement is terminated with an **end if** clause. SHARP pseudo code annotates the **if** statements with brackets bounding the beginning and end of the statement, as illustrated in Item a of Figure 32.

The case statement provides for the selection one of a set of multiple alternative actions, as a function of the value of an expression. Only one of the alternative actions is taken. SHARP pseudo code annotates the case statement with brackets as illustrated in Item b of Figure 32.



(a) USE OF "IF" STATEMENTS

FIGURE 32. ANNOTATED PSEUDO CODE

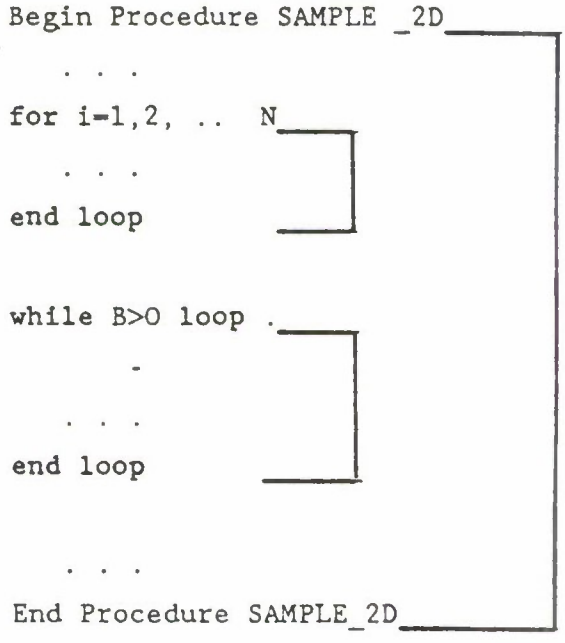
```
Begin Procedure SAMPLE_2B
   . . .
case  TEST   is
   when PASS  Call Procedure CONTINUE
   when FAIL  Call Procedure RESTART
end case
   . . .
End Procedure  SAMPLE_2B
```

(b) USE OF THE "CASE" STATEMENT

```
Begin Procedure SAMPLE_2C
   . . .
loop
   . . .
end loop

loop
   . . .
        loop
        . . .
        end loop
        . . .
end loop
   . . .
End Procedure SAMPLE_2c
```

(c) USE OF THE "LOOP" STATEMENT

FIGURE 32. (CONTINUED)



(d) USE OF THE "FOR" AND "WHILE" STATEMENTS

FIGURE 32. (CONCLUDED)

Repetitive execution of action is accomplished in Ada using the loop statement. The basic loop is accomplished using a loop and end loop statement. To leave a loop, an exit statement is used. SHARP pseudo code annotates the loop statement with brackets, as illustrated in Item c of Figure 32.

To repeat a loop for a specific number of times, the basic loop can be preceeded by a for iteration clause. Also, another form of iteration can be accomplished with the **while** statement, whereby a sequence of statements is repeated as long as some condition is true. SHARP pseudo code annotates the for and **while** statements with brackets, as illustrated in Item d of Figure 32.

2.5.1b Algorithms

SHARP criteria state that all mathematical algorithms to be implemented in the body of a program unit must be clearly shown in the pseudo code. The criteria makes no constraints on how the formulation is presented. Therefore, either the notation of mathematics or a programming language is applicable.

2.5.1c Program Unit Call

SHARP criteria state all program unit calls made in the body of a program unit must be shown by the pseudo-code, but does not require specification of parameters passed in conjunction with the call. The parameter passing is shown by SHARP Data Flow Diagrams (for subprograms) and by SHARP Task Rendezvous Diagrams. As illustrated in Figure 33, SHARP does require that annotation be provided to identify the package a program unit is to be implemented in, if it is to be other than the package (or task) of the subject object implementation.

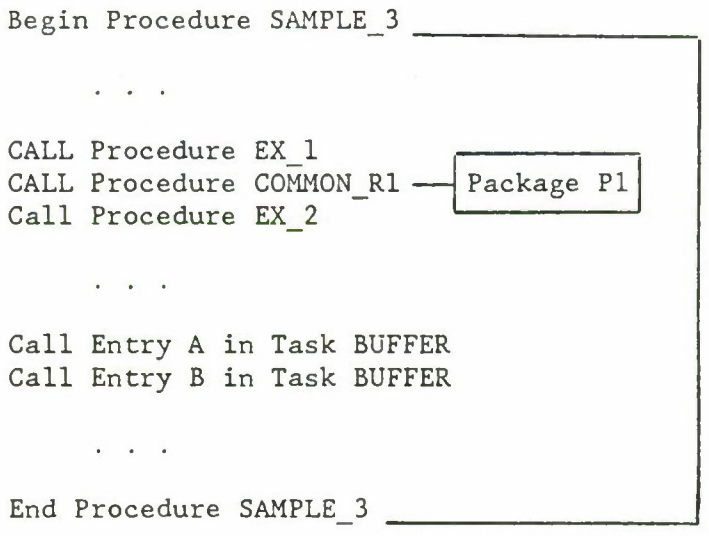


FIGURE 33. SHARP PSEUDO CODE FOR REPRESENTING PROGRAM UNIT CALLS

2.5.1d Generic Instantiation

The specific parameters passed between object implementations is represented by SHARP Data Flow Diagrams. This diagram represents parameters being passed as shaded circles on directed lines.

If the program unit being called is generic, the line introduced to partition a pictograph into a narrow part (i.e., to represent the program unit's specification) and a wide part (i.e., to represent the program unit's body) is dashed, as illustrated in Figure 34. In Ada, the specific name of the generic procedure and the definition of unspecified types must be created prior to calling a generic program unit. This is referred to as instantiation.

In addition to making the type of a passed parameter generic, Ada also permits the range of values permissible for a passed parameter to be generic for the **in** and **in out** modes of parameter passing. (The mode **out** cannot be used with a generic parameter.) Several instances of such values can be established in generic instantiations.

SHARP criteria state that generic instantiation must be represented in the SHARP pseudo code for program unit bodies. As shown in Figure 35, this pseudo code is bracketed by dashed lines.

2.5.1e Input/Output

The high level input/output (I/O) facilities provided in Ada are not, as in other languages, supplied in the form of additional language constructs. Rather I/O is accomplished using the predefined packages SEQUENTIAL_IO, DIRECT_IO and TEXT_IO. Their package specifications give a precise description of the I/O facilities provided. TEXT_IO is used to read in data generated by humans and to write out data to be read by humans.

SEQUENTIAL_IO and DIRECT_IO are used for data written and read by a computer and not a human (e.g., to store data on disc or magnetic tape). They are generic Ada packages. For example, to declare files for elements of a given type, an instance of the package must be declared and the required element type must be specified as an actual generic parameter. Such generic instantiation is represented using annotated pseudo code, as we have already explained. Generic instantiation for this package is explained in Chapter 15 of An Introduction to Ada.[2]

Unlike SEQUENTIAL_IO and DIRECT_IO, TEXT_IO is not generic but is an ordinary Ada package. It is accessed using the Ada 'with' clause.

In SHARP pseudo code, calls to program units contained in package TEXT_IO is presented in the manner described in Paragraph 2.5.1c. Instantiation of program units contained in packages SEQUENTIAL_IO and DIRECT_IO is presented in the manner described in Paragraph 2.5.1d.

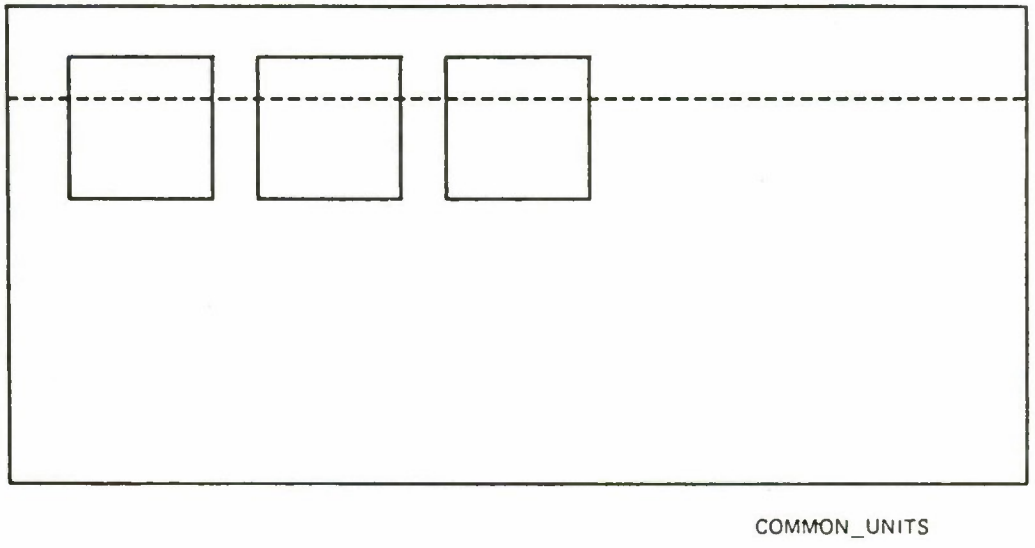


FIGURE 34. SHARP REPRESENTATION OF A GENERIC PROGRAM UNIT

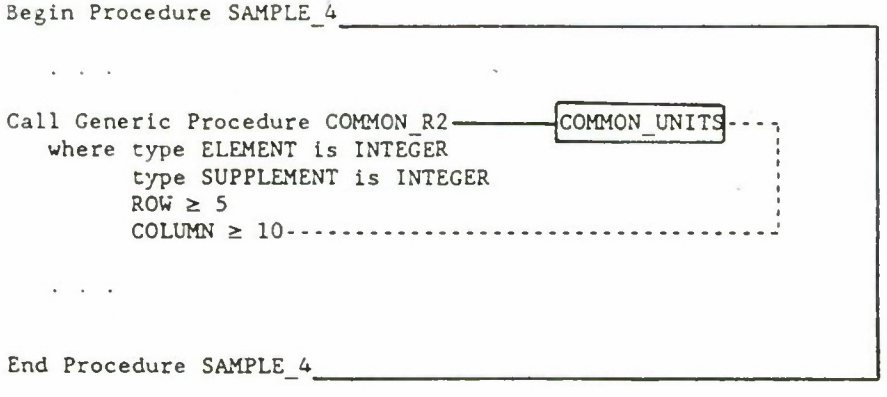


FIGURE 35. SHARP REPRESENTATION OF A GENERIC INSTANTIATION

2.5.1f Exception Handling

Ada provides an explicit mechanism for detecting and responding to an anomaly. The anomaly, for example, could be associated with erroneous input data or overflow conditions. SHARP criteria require that the design of a program unit's body must specify the detection of the anomaly and the course of action taken after the occurrence of the anomaly.

In Ada, the detection of the anomaly causes normal program execution to be suspended and control transferred to an exception handler. Once the exception handler has completed its processing, control transfers to code following the exception handler code.

Figure 36 provides an example of annotated SHARP pseudo code used to specify the detection of an anomaly and action taken in an exception handler. SHARP criteria requires that (a) the pseudo code for the anomaly detection must be introduced by the key words 'raise exception' and must be bracketed as shown in the figure; (b) the pseudo code for the action taken upon occurrence of the anomaly must be introduced by the key words 'exception handler,' must be concluded with the key word 'end,' and must be bracketed as shown in the figure, and (c) an arrow must point from the bracket enclosing pseudo code for the anomaly detection to the bracket enclosing pseudo code for the exception handler.

2.5.1g Example

Figure 37 provides an example of annotated SHARP pseudo code.

2.5.2 Data Structure Detail Glossary (Option I)

A SHARP data structure diagram presents an abstracted representation of types (subtypes), constants and variables. It establishes the name and visibility of each. It indicates the type of each variable, including arrays, records, discriminants and dynamic variables (which are designated as access types). It represents types and variables declared as private.

To complete the description of a data structure, a glossary can be used to designate additional detail, including the following:

- The use of a variable and constant
- The range of values the design designates as permissible in a type and subtype
- The components associated with a record type
- The type of designated variables associated with an access type.

3 APPLICATION OF THE SHARP PICTORIAL ABSTRACTS

This section presents examples of general applications of the various pictorial options of SHARP. Examples of specific applications are provided in Chapters III and IV.

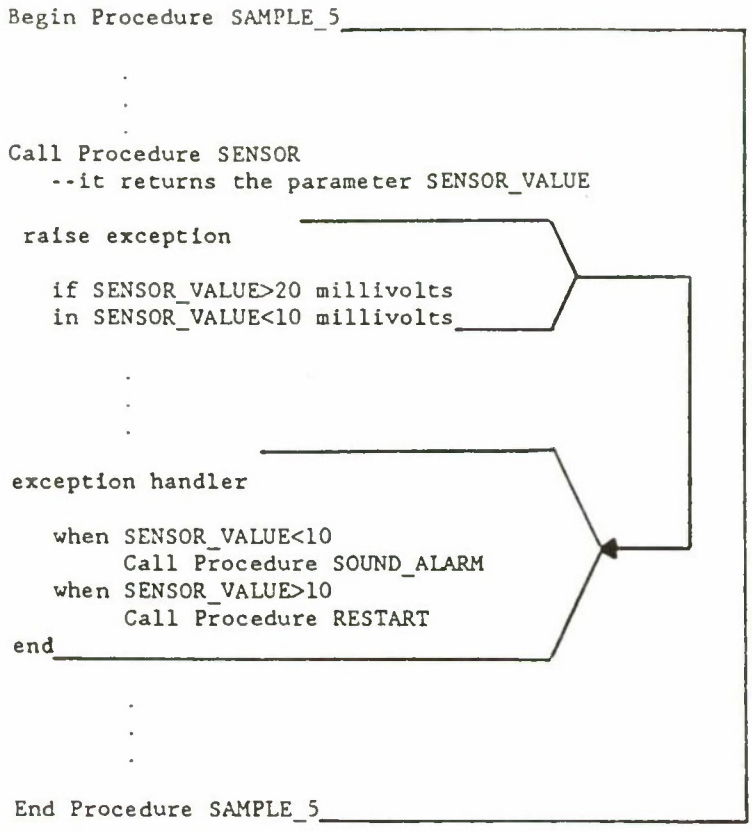


FIGURE 36. SHARP ANNOTATED PSEUDO CODE FOR RAISING AND HANDLING EXCEPTIONS

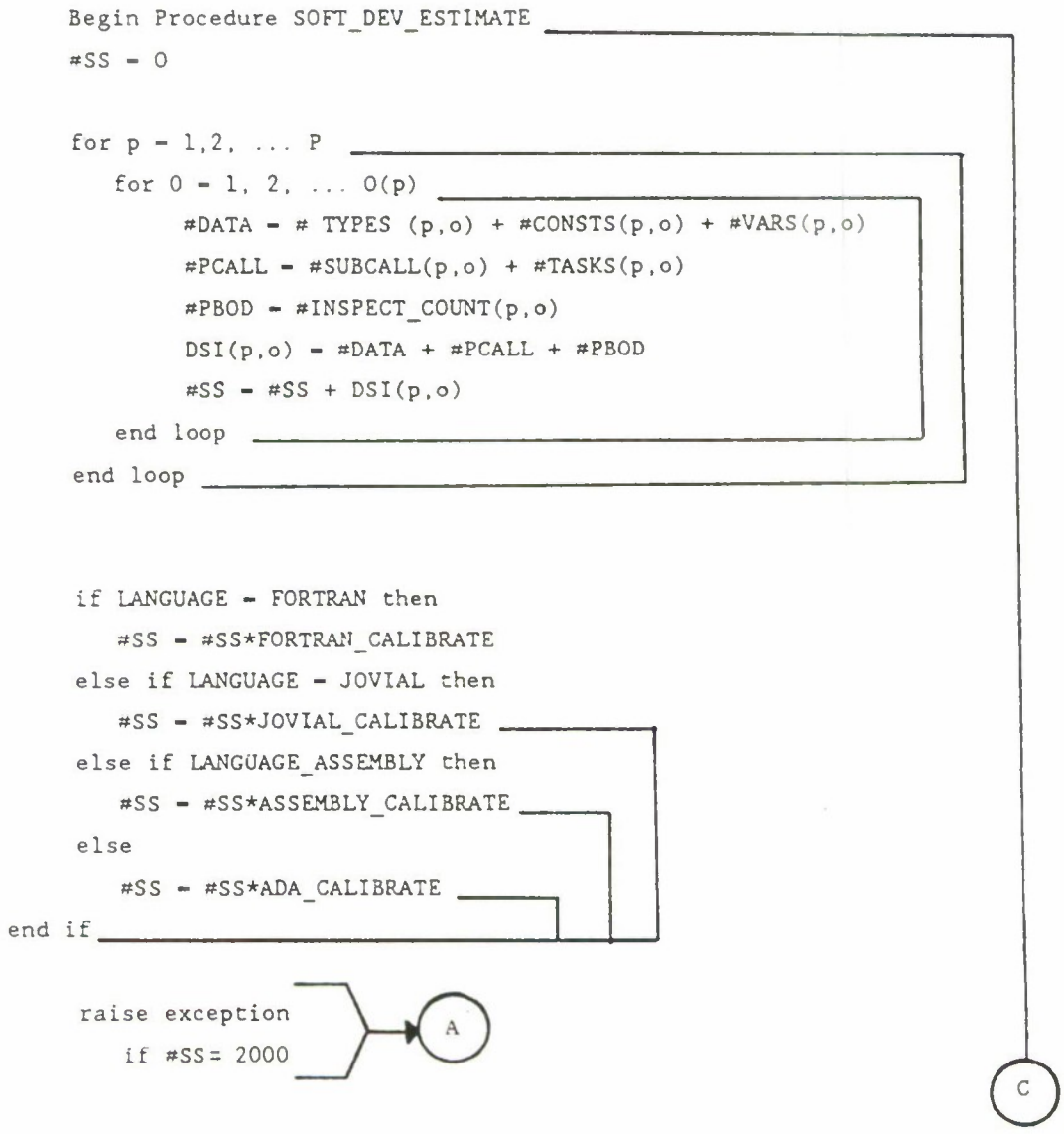


FIGURE 37. EXAMPLE OF ANNOTATED PSEUDO CODE

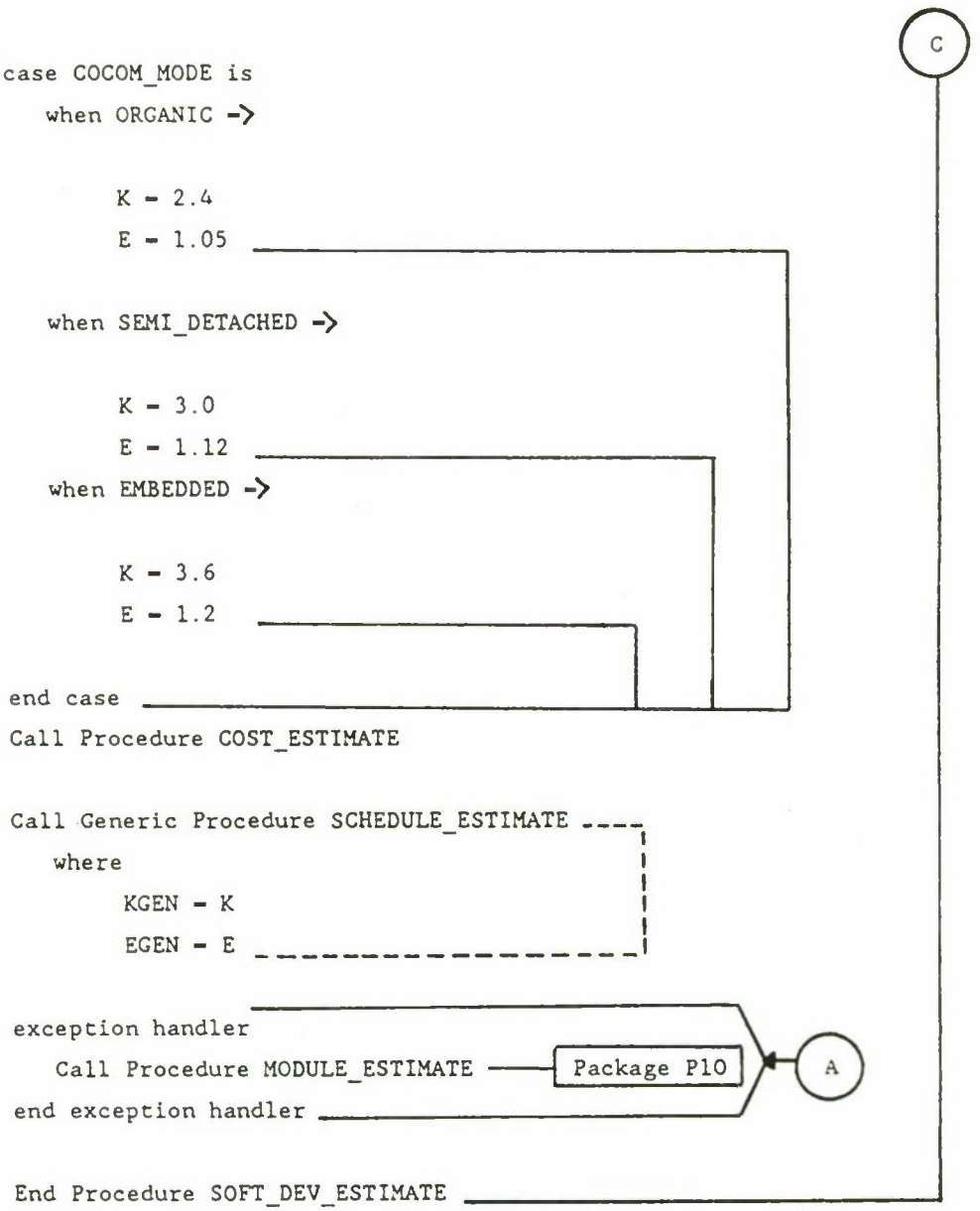


FIGURE 37. (CONCLUDED)

The examples are given in the context of a designer presenting his design requirements at a design review. The examples illustrate the versatility of the SHARP notation. The notation is shown to effectively represent extremes in combinations of program units that a designer may choose in the implementation of a large Ada computer program, regardless of whether an object-oriented or more traditional design approach is being taken.

3.1 EXAMPLE 1 - SOFTWARE ABSTRACTION WITH PROCESSES

As discussed in Section 3.1 of Chapter I, processes consist of chains of software modules introduced to (a) satisfy user commands and communication interface requests and (b) perform processing automatically initiated on a periodic or some other basis. Processes execute concurrently in the time slice sense.

With languages such as FORTRAN, processes execute concurrently under operating system control. With Ada, a designer can choose to implement each process within the body of an Ada task, declared in the main program.

Figure 38 provides an example of representing the tasks nested in the main program. As discussed in Chapter I, the diagram also indicates Ada packages the designer wants the main program to access, through the Ada "with" clause; and subprograms the designer wants nested in the operational part of procedure MAIN.

This high level diagram can be used by a designer to represent interfaces between the large Ada computer program and the "outside world." As a high level diagram, it provides a manager with a conceptual view of the functions of the large Ada computer program.

3.2 EXAMPLE 2 - SOFTWARE ABSTRACTION WITH PROCESS LEVELS

Since the requirements of a process are often complex, a designer typically will use additional abstraction in the development and presentation of his design. For example, a relatively small and easily comprehended portion can be implemented at one level, with the rest of process requirements implemented at other levels. At each other level, the technique is repeated, with abstraction of variables and manipulation of the data. The designer may choose to abstract the design using layers of packages, which is discussed in Section 3.3 of this chapter. The problem may lend itself to abstraction using Ada tasks, which is discussed in Section 3.4 of this chapter. Alternatively, the designer may choose to implement the design using a controlling structure of subprograms and tasks assigned to levels, which access packages using the Ada "with" clause.

With this approach, the body of each task, activated in procedure MAIN to account for a process, is abstracted by constraining the amount of detail within it to an easily understood amount. Excluded detail can be passed to the bodies of called program units. The called program units may be contained in an Ada package, which is made available through the use of the Ada "with" clause.

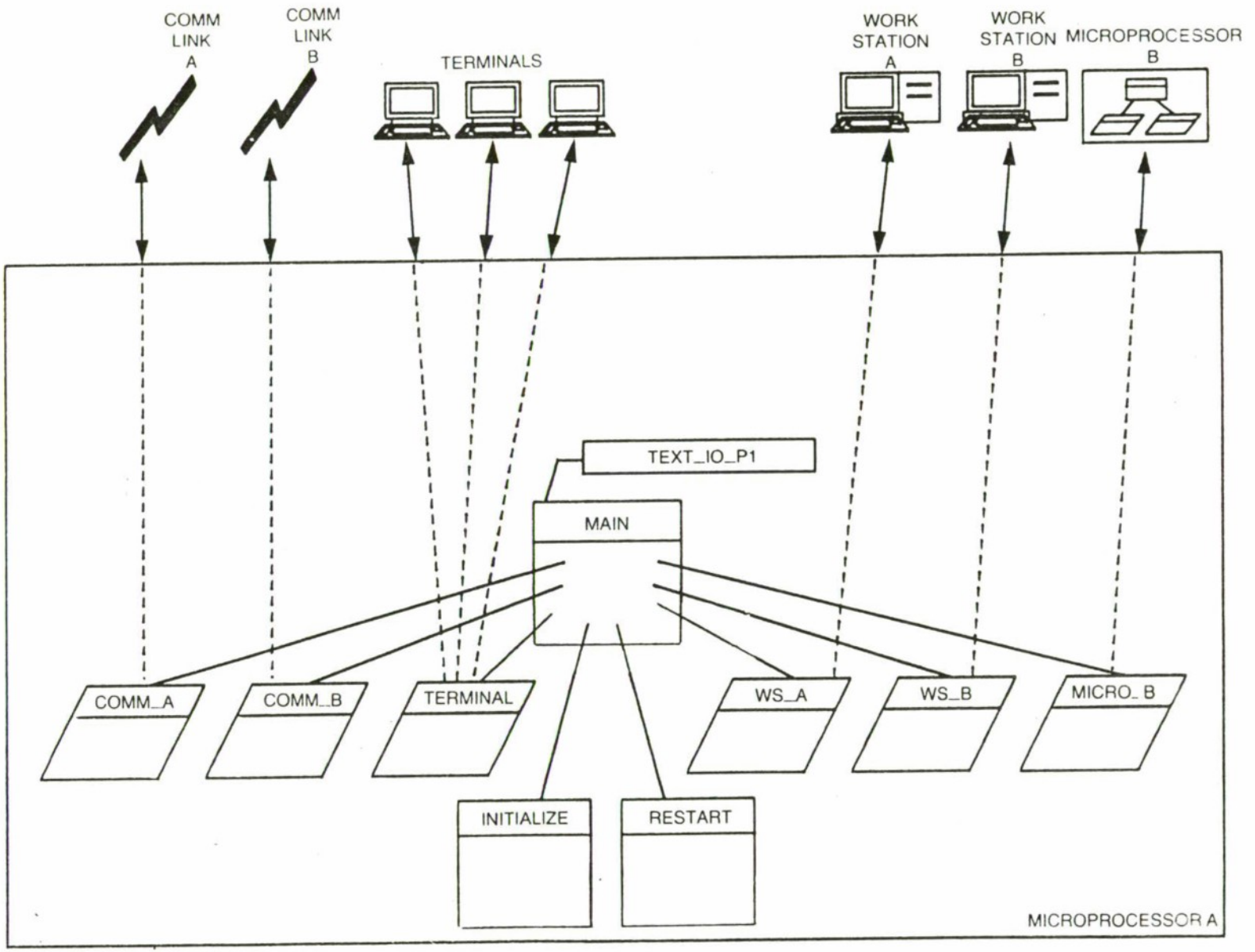


FIGURE 38. REPRESENTING THE MAIN PROGRAM

The deferred bodies of the called program units can be designed subject to the same constraints that applied in the design of the task's body. Therefore, these bodies are also constrained to an easily understood amount of detail, with lower detail moved again to called program units.

To illustrate the physical layout of Ada source code prepared in accordance with such an abstracted design, consider the sample computer program shown in the Appendix B. The main program appears first, followed by a set of program unit threads for each process. Implementation of this program would require thousands of lines of Ada code. The information provided in the listing in the Attachment requires several pages, even in its "skeleton" form. As shown in Figure 39, the same information can be provided in SHARP Hierarchy Diagrams (Option C) on one page. Therefore, this abstract can be used at design reviews to summarize thousands of lines of implementing code.

Figure 40 illustrates how the SHARP Invocation Diagram (Option D) can be used by the designer to show the sequence of calls for the program units identified in a hierarchy diagram. In addition, the designer can select a set of other SHARP options to provide Ada abstracts of lower level detail.

For example, Figure 41 illustrates how a designer can use the Subprogram Data Flow Diagram (Option E) to specify data flow into and out of program unit PU_A3a of Thread A. As another example, Figure 42 illustrates how a designer can use Annotated Pseudo Code (Option H) to specify conditions of procedure calls between Levels 3 and 4 of Thread C, flagged as conditional by the Invocation Diagram.

The designer might use the Task Rendezvous Diagram (Option F) to show details of rendezvous between tasks. Examples of these diagrams are provided in Section 3.4 of this chapter.

The designer also might use the Data Structure Diagram (Option G) to show, for example, the visibility of types, variables, constants and task types. Examples of these diagrams are provided in the next section.

3.3 EXAMPLES 3 - SOFTWARE ABSTRACTION WITH ADA PACKAGES

As we suggest in the previous section, a designer may choose to abstract the design using layers of packages. This approach can be used in conjunction with object-oriented design to minimize dependency relationships within a large Ada computer program, a potentially significant problem.

As discussed in Section 2.2 of this chapter, object-oriented design is a software development method in which a large computer program is composed with object implementations. Each object implementation consists of a set of operations unique to it and a local state defined in a data structure. The implementation of unique operations are known only to the object. The parameters describing the object and its state may be passed as parameters to other objects.

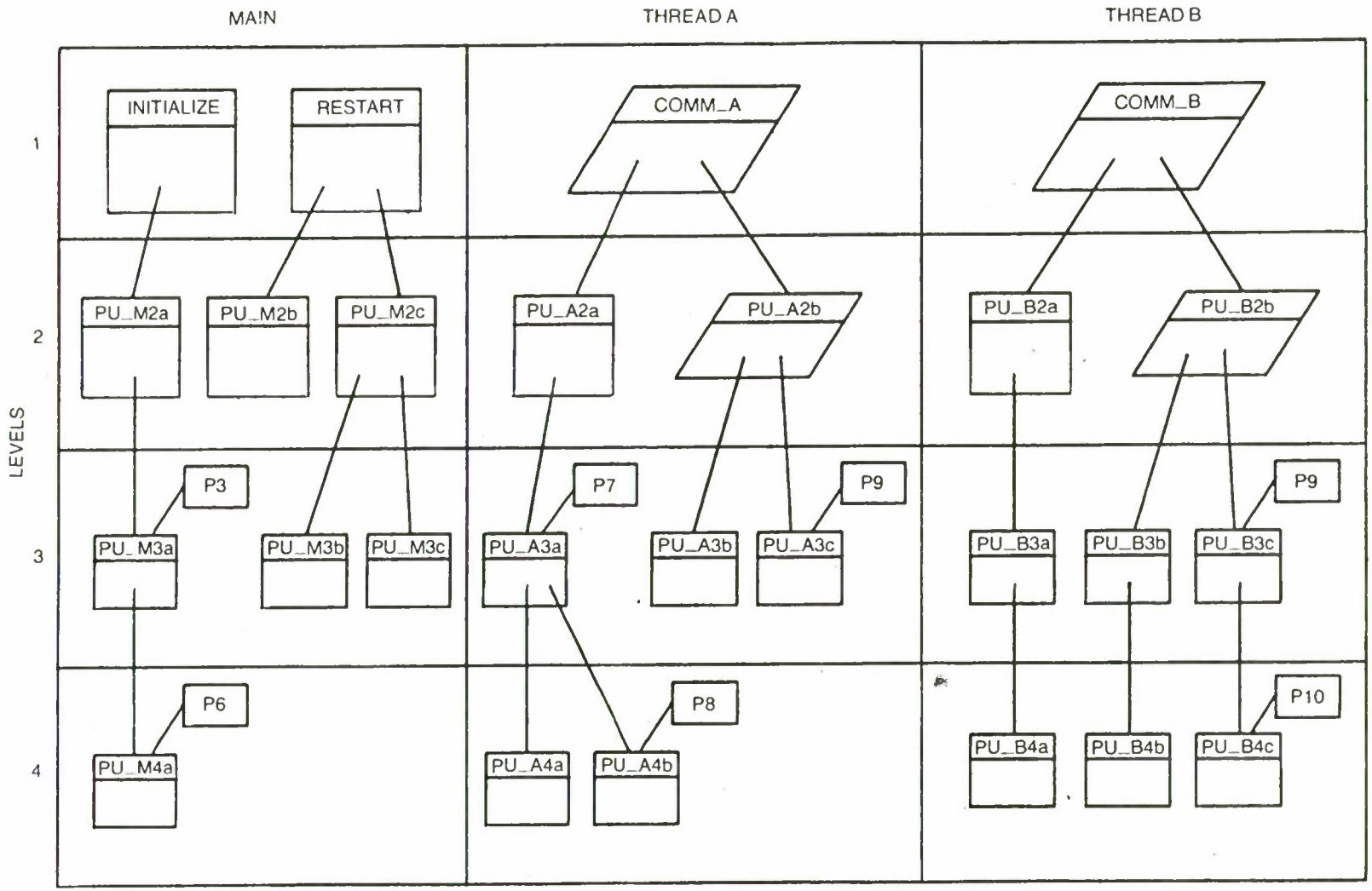


FIGURE 39. SHARP HIERARCHY DIAGRAMS
(Option A, Example 2)

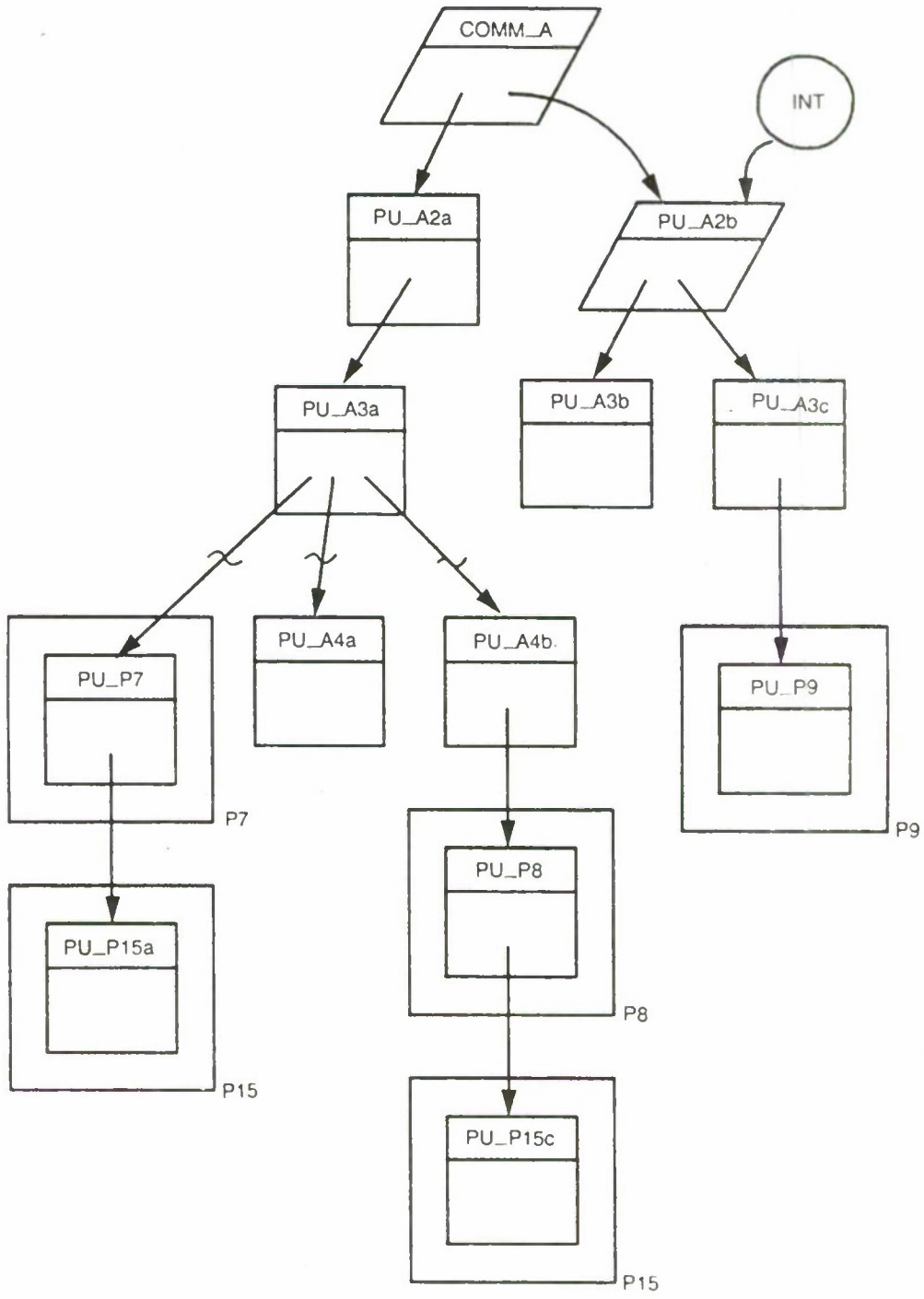


FIGURE 40. SHARP INVOCATION DIAGRAM
(Option D, Example 2)

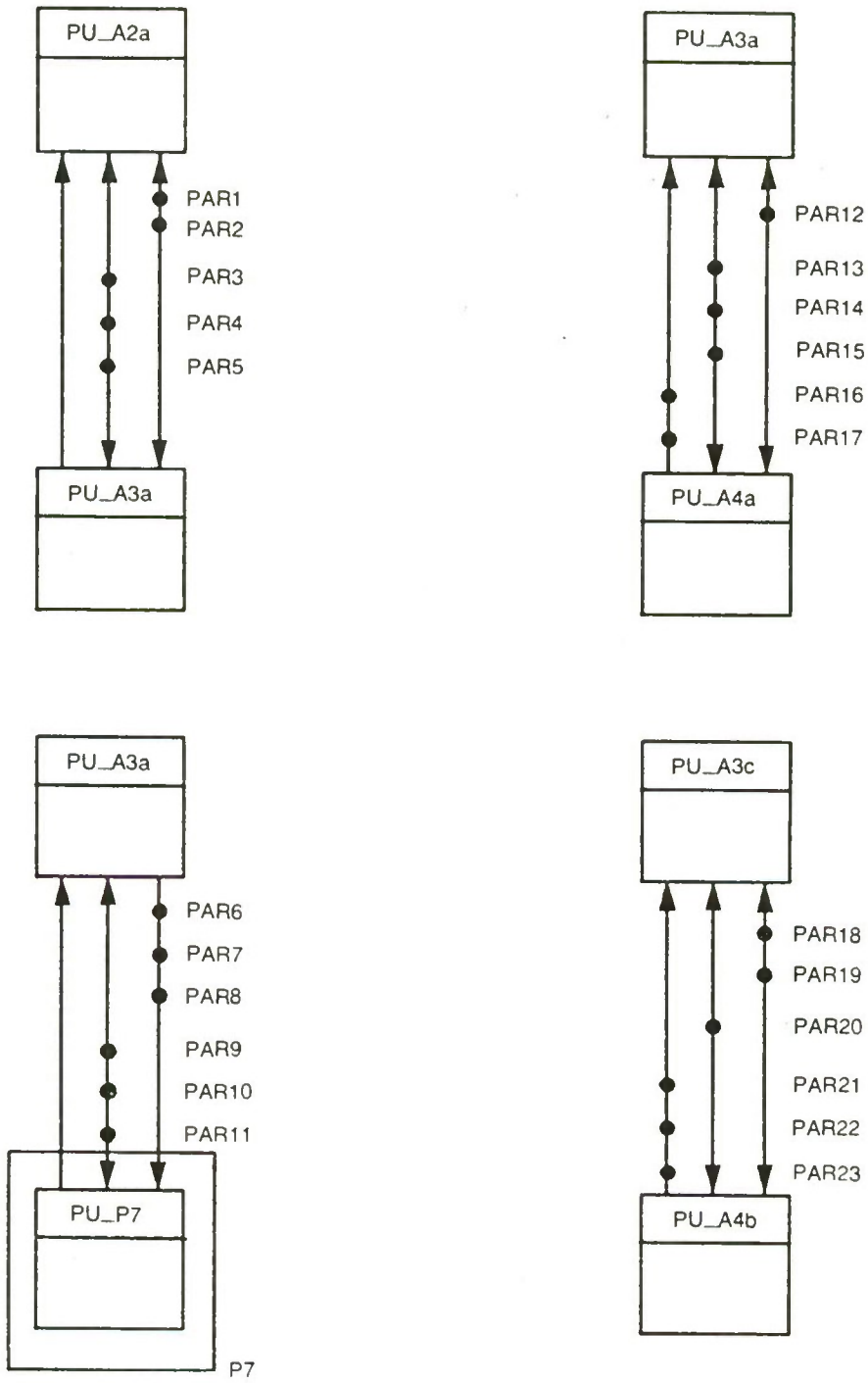


FIGURE 41. SHARP DATA FLOW DIAGRAM
(Option E , Example 2)

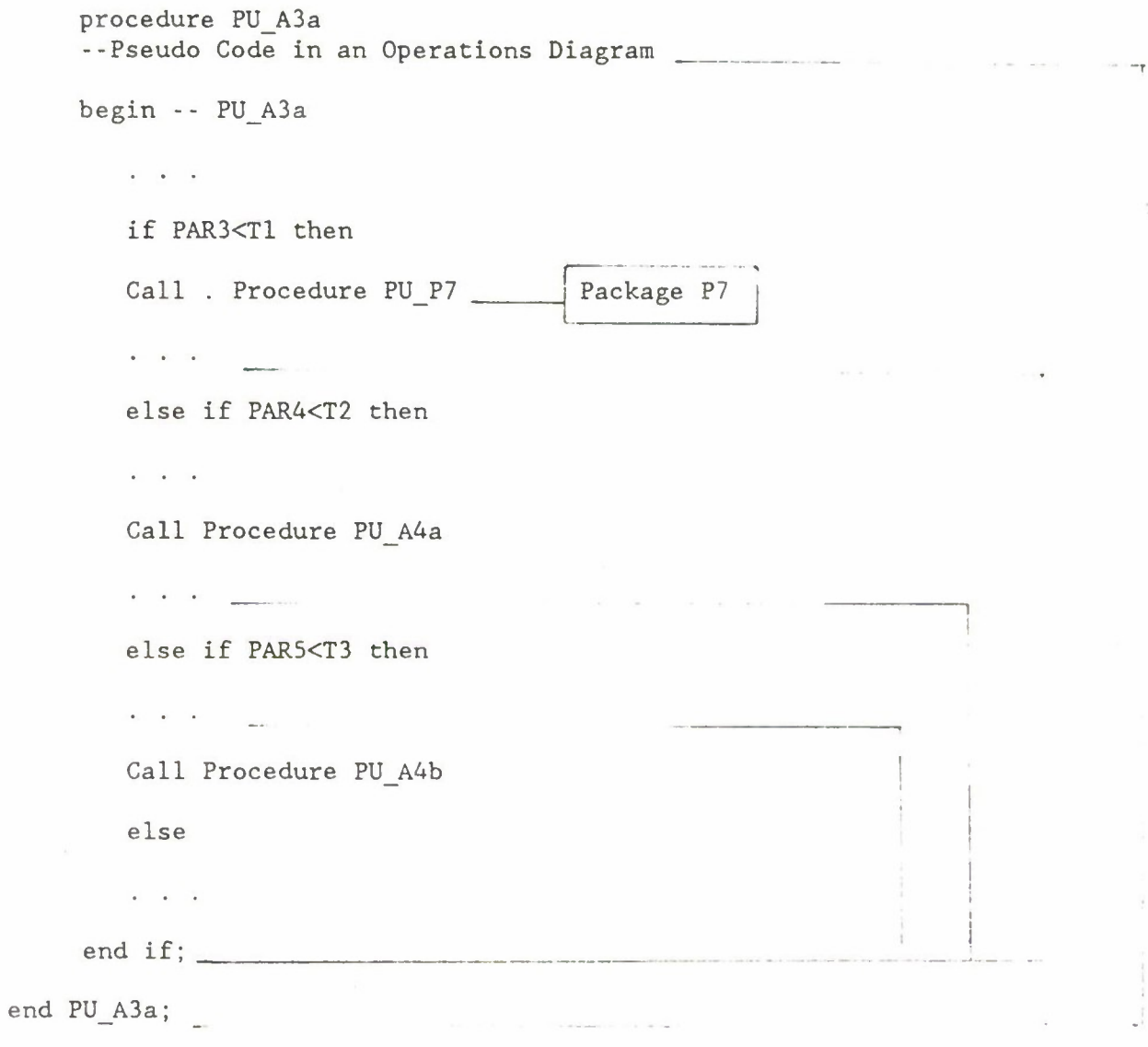


FIGURE 42. USE OF ANNOTATED PSEUDO CODE (OPTION H, EXAMPLE 2)

In Ada, packages are important to the implementation of object-oriented designs. Information hidden within each package limits dependency relationships between objects. Only parameters declared in program units, contained within the specification of a package, can be passed from one package to another.

A designer can use a set of Ada Package Content Diagrams (Option B) to show multiple packages, each encapsulating an object implementation. The multiple packages can be thought of as layers of packages, as illustrated in Figure 43. In this example, the designer has introduced a procedure in each package as the interfacing program unit. Each interfacing program unit communicates with other corresponding interfacing program units in other packages. In practice, it is desirable to keep the interfaces between the packages as simple as possible. Parameters passed between object implementations should not couple the implementations. The data flow between interfacing procedures can be specified using Subprogram Data Flow Diagrams (Option E).

Figure 44 illustrates how a designer can use an Invocation Diagram (Option D) to specify the sequence of calls between the package layers. In this example, the designer requires that the procedure in P1 calls the procedure in P2, which in turn calls the procedure in P3. The procedure in P3 calls procedures in P4 and P5. This procedure in P5 recursively calls the procedure in P2.

Each package may be complex within itself, although its interface with other packages may be kept relatively straightforward. Program units contained within the body of a package are hidden from the other packages. As illustrated in Figure 45, the complex inner structure of the package can be specified using one or more of the SHARP options, in the same manner as the detailed design was shown for a process thread as discussed in Section 3.2 of this chapter.

3.4 EXAMPLE 4 - SOFTWARE ABSTRACTION WITH ADA TASKS

As also indicated in Section 3.2, abstraction can be accomplished using tasks in the main program to implement processes. At lower levels, tasks can also be used (e.g., to monitor multiple sensors). As described in Buhr's book System Design with Ada [1], special purpose tasks termed "slaves, starters, schedulers, buffers, secretaries, agents, transporters and pools" can be used in various combinations to facilitate processing needs.

At the lower levels, abstraction with tasks takes place so that processing requirements and data structures can be spread among different tasks, all executing concurrently in an independent manner. In this way, tasks can be used, to a certain extent, in object-oriented designs in that the local state of an object implementation can be is defined by variables local to the task; and operations within the body of the task are unique to the object implementation.

If a designer decides to extensively use Ada tasks, he can use the Invocation Diagram (Option D) to show rendezvous or conditions of rendezvous, as illustrated in Figure 46. The designer can use a Task Rendezvous Diagram

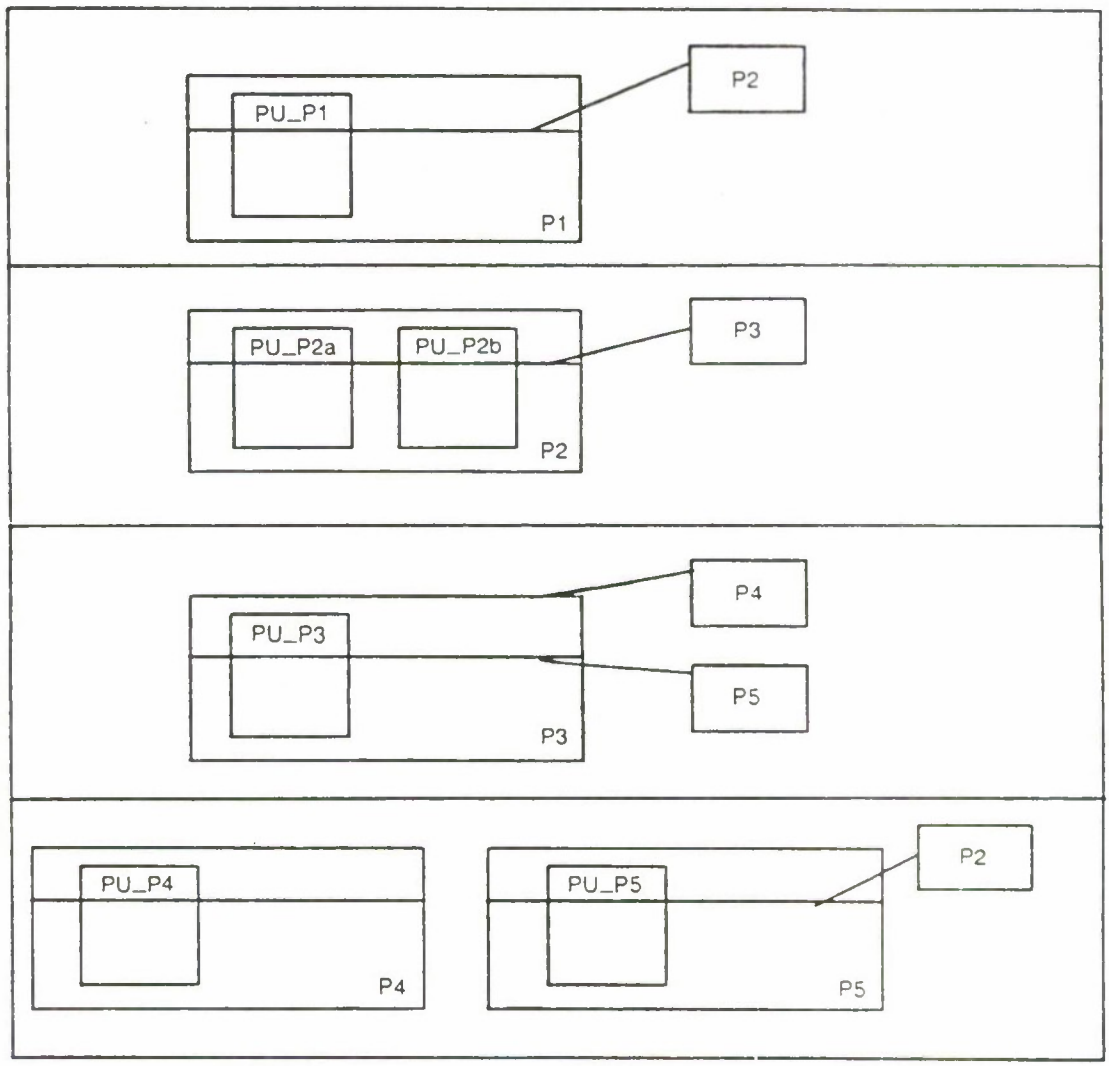


FIGURE 43 LAYERS OF ADA PACKAGES
(Option B, Example 3)

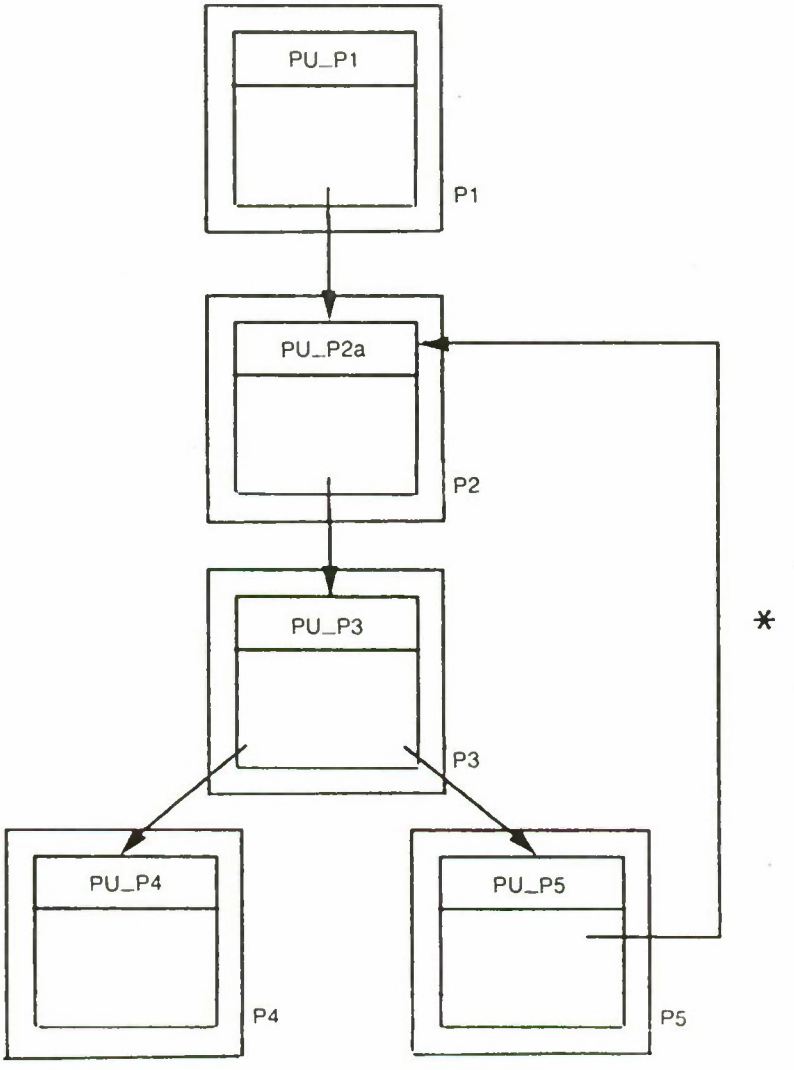


FIGURE 44. INVOCATION DIAGRAM
(Option D, Example 3)

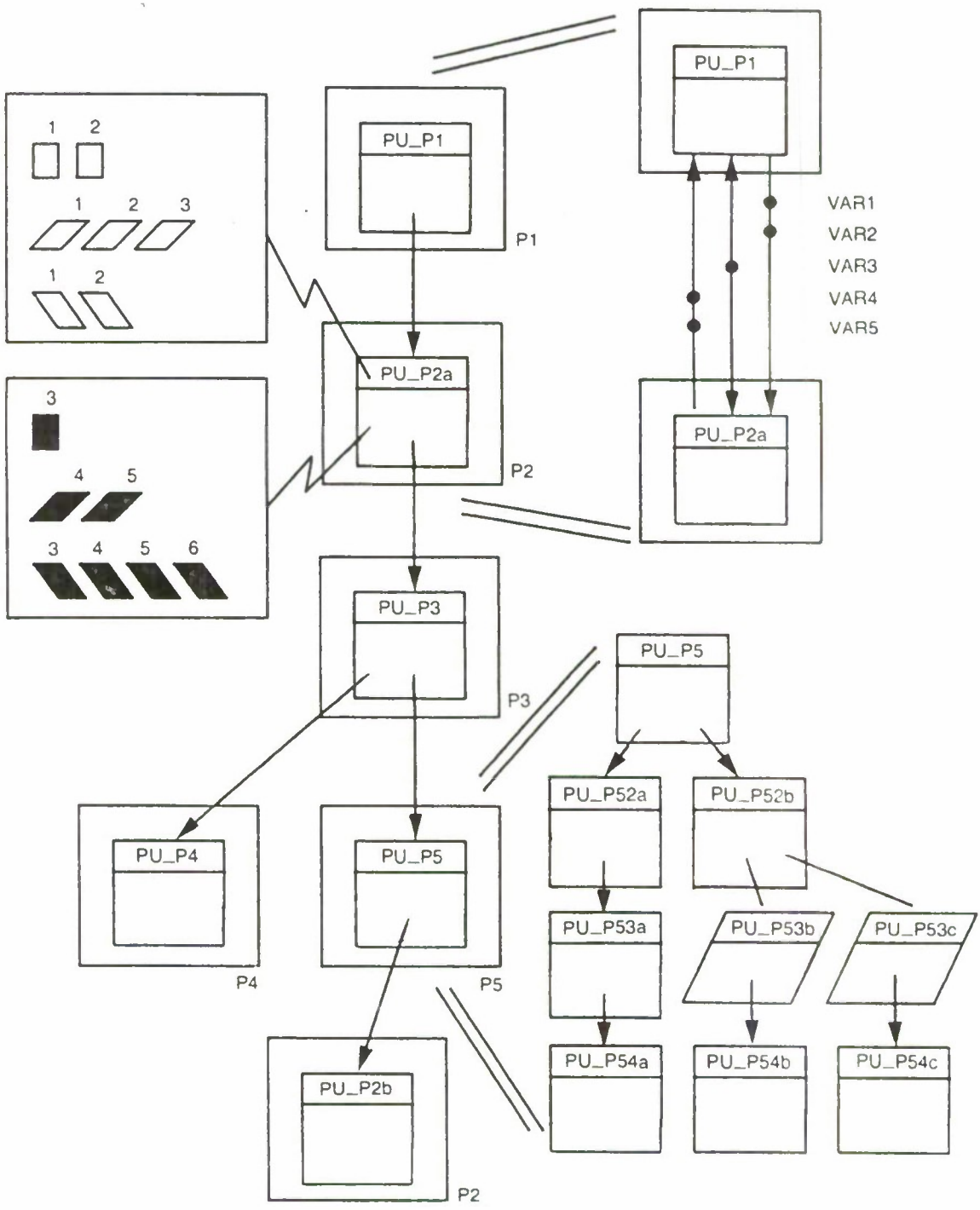


FIGURE 45. EXPANDING THE DETAIL SHOWN IN AN INVOCATION DIAGRAM FOR LAYERED PACKAGES

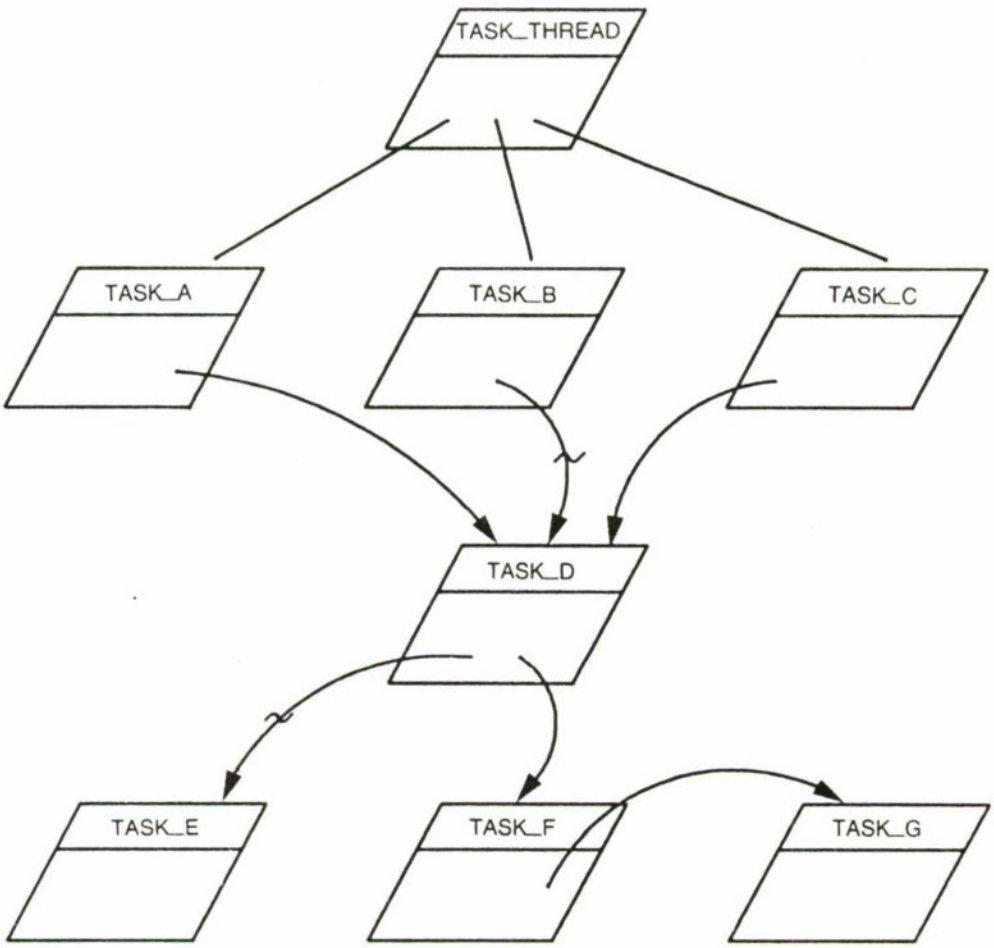


FIGURE 46. INVOCATION DIAGRAM
(Option D, Example 4)

(Option F) to specify task entry points, conditions of task acceptance and parameters passed during rendezvous, as illustrated in Figure 47. As mentioned in the case of conditional procedure calls, the designer can show the logic calls and conditional acceptances using the SHARP Annotated Pseudo Code (Option H).

As we have indicated, some feel that such use of multiple tasks should be constrained due to slow rendezvous times and potential complicated temporal properties. The complex temporal properties may be difficult to test during computer program development and maintenance.

4 CHAPTER SUMMARY

SHARP establishes a set of pictorial options that can be used to represent the design for an Ada computer program, whether designed by object-oriented or traditional methods. As such, SHARP abstracts are effective at design reviews, and to help specify requirements for a large and complex computer program in design documentation. SHARP abstracts apply equally well to all extremes of Ada designs, including design abstraction with layers of Ada packages, extensive use of Ada tasks, and mixtures of packages, tasks and subprograms.

The options of SHARP produce various abstracts of an Ada computer program. The Ada package content diagram presents at a high level an overview of an Ada package. The hierarchy and invocation diagrams are intermediate level diagrams that present the overall structure and invocation sequence of program units nested in a subject program unit (e.g., a procedure declared in the specification of a package).

Lower level design detail can be specified in rendezvous, subprogram data flow and data structure diagrams. However, each of these diagrams is at a substantially higher level than PDL.

At the lowest level, annotated pseudo code can be used to represent the data structure detail and operational part of the body of a program unit, and a glossary can be used to define certain data structure detail. At this lowest level of abstraction, SHARP junctions with traditional design presentation techniques in a manner similar to PDL.

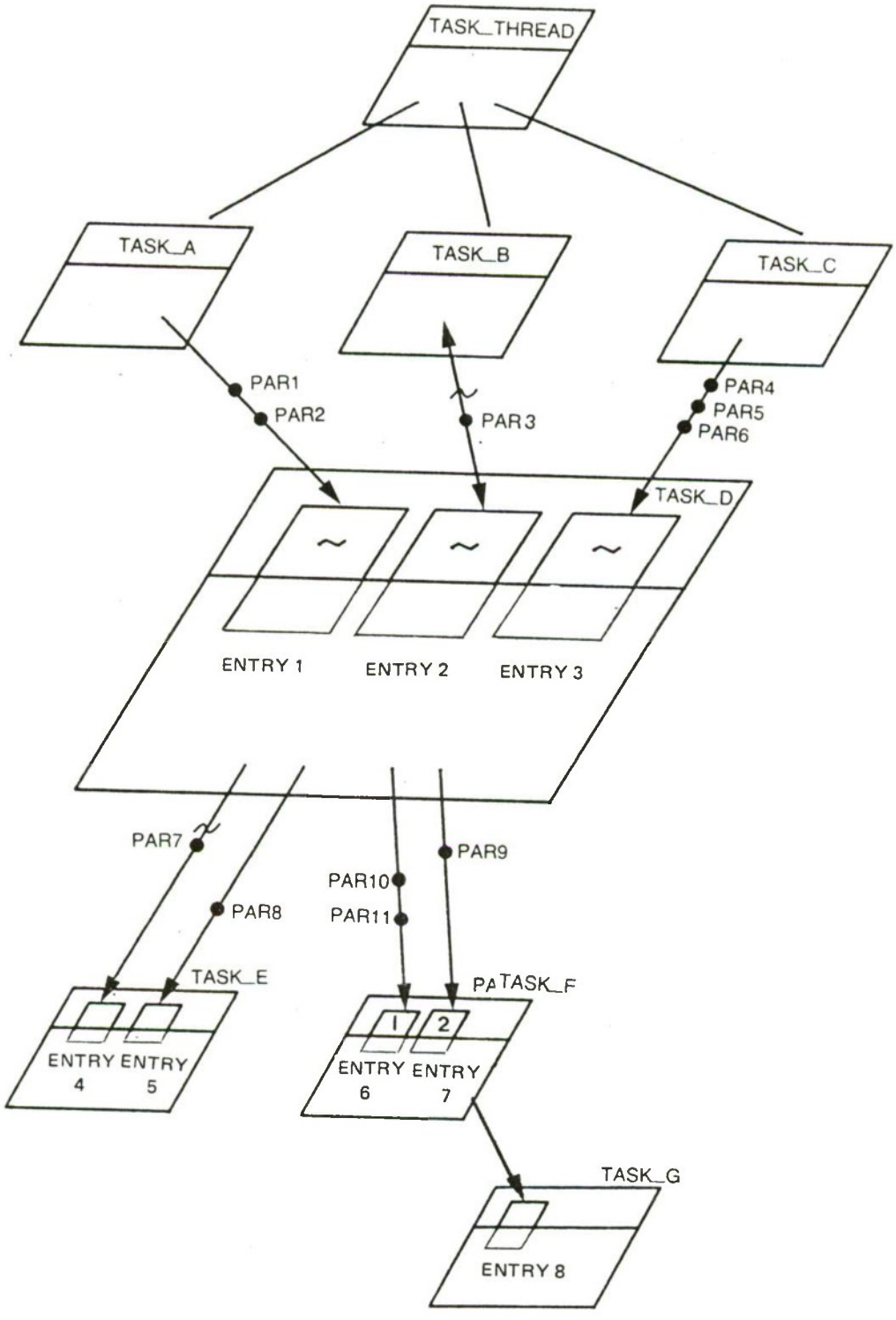


FIGURE 47. TASK RENDEZVOUS DIAGRAM
(Option F, Example 4)

PART TWO: APPLICATION OF SHARP

CHAPTER III

Basic Issues in Object-Oriented Design With Ada

This chapter addresses basic issues in designing an Ada computer program using principles of decoupling inherent in an object-oriented design. The issues are raised through example.

With the example, we illustrate the object-oriented design approach and the inherent effective software maintainability associated with an Ada computer program that has been designed in an object-oriented manner. Then we address increasing the execution speed and constraining the use of memory within the Ada-unique object-oriented design suggested for the example. Variations on the example demonstrate that whatever the design goals, the resulting object-oriented design can be effectively represented by abstracts of SHARP.

This chapter is very basic in nature and is meant for those unfamiliar with steps needed to develop an Ada-unique object-oriented design.

1 INTRODUCTION

1.1 BACKGROUND

In the past, the cost of software development has been relatively high and has increased exponentially as a function of the size of a computer program. These high costs can be attributed to several factors, including the effect of complex dependency relationships between types, variables and program units introduced using traditional software design and development techniques.

As described in Chapter II, complex dependency relationships can be controlled using Ada-unique object-oriented techniques, and in this way software development costs can be significantly reduced. With Ada, the object-oriented approach is also critical because of the speed of Ada compilers. Object implementations within a large and complex Ada computer program can be encapsulated in loosely coupled Ada packages and tasks, which the development team can code and test independently. By constraining their size and stubbing program units interacting with them, the Ada packages and tasks can be compiled individually and recompiled in a timely manner during during their development.

Because of the need to control complex dependency relationships and to adapt to slow Ada compilers, it is anticipated that large Ada computer programs by necessity will be developed using object-oriented techniques. In practice, a designer must make several decisions with respect to subdivision of processing requirements into objects and the implementation of each object. The extent of decoupling of the implemented objects and the object selection approach taken are driven by design choice and constraints associated with such things as memory and execution speed limitations.

1.2 CHAPTER SCOPE

In this chapter, variations on an object-oriented design are discussed for a sample problem. The variations are introduced due to different design goals. Through example, it is shown that typically we cannot speak of an unique optimal design. Rather the design is driven to a significant extent by the designers objectives, be they maintainability, execution speed, constrained memory or some other goal; the extent and manner in which he decides to implement the object-oriented design in Ada; and the extent to which he constrains the interaction between objects to limit dependency relationships.

Section 2 establishes requirements for a sample problem and describes an object-oriented design for the sample problem. It provides a second version of this design, a modification of the original design to increase execution speed; and also a third version of the design, a modification of the original design to save on memory used.

2 EXAMPLES OF AN ADA-UNIQUE OBJECT-ORIENTED DESIGN

Let us consider restart/recovery capabilities in a large and complex Ada computer program. In this program, several application module transactions execute concurrently.

The computer program is to have the capability to abort a transaction, restart it from its beginning, or recover the transaction by restarting from a dynamically established point in the program (e.g., a breakpoint or checkpoint); all as commanded by an operator from a work station.

A transaction tracking table is to be established to record pertinent information about the transactions (e.g., transaction type, responsible operator, console ID, status and breakpoint information). When a transaction is recovered, restarted or aborted, the transaction table must be updated to reflect this event. Also, a counter must be incremented after each transaction to keep a record of how many times a transaction has been restarted or recovered.

In Section 2.1, we consider an Ada-unique design for the restart/recovery software with efficient future maintainability as a goal. In Section 2,.2, we address variations on the original design introduced due to speed and memory constraints. Although the sample problem is relatively small, its design is developed as if it were a large and complex situation, so as to demonstrate the Ada-unique object-oriented design approach and its representation with SHARP abstracts.

2.1 ADA-UNIQUE DESIGN CONSIDERATIONS

Developing Ada-unique software in an object-oriented manner is expected to lower software development costs, as quantified in Chapter VII. More importantly, this approach is expected to significantly lower software maintenance costs.

The localization of design complexity inherent in an object-oriented design simplifies software maintenance. In the past, global parameters and routines were shared among many program units. If during maintenance, any one of the global parameters or routines had to be changed in conjunction with a program update or improvement, the change often adversely affected several parts of the computer program. Seemingly innocent changes typically caused serious problems. However, by localizing design complexity in an object-oriented manner, the effect of maintenance changes are trapped within the implementation of an object. As Booch observes, "The benefit of this facility (minimizing dependencies) should be clear: not only does this enforce one's abstraction and, hence, help manage the complexity of the problem space, but by localizing the design decisions made about as object, we reduce the scope of change upon the system.[1]

This phenomena is very important. As history has shown, software maintenance costs associated with traditional designs have been very expensive, typically costing more than the original development costs. In some cases, maintenance over ten years or more has cost as much as 50 times more than the original software development costs.

2.1.1 Establishing an Ada-Unique Design with SHARP

To establish a SHARP representation of an Ada-unique design based upon principles of object-oriented design, a designer can consider the following factors:

a. SHARP pictographs are introduced to represent Ada tasks declared in the main program to establish concurrently executing processes.

b. High level SHARP abstracts are introduced to represent Ada packages and tasks that encapsulate the implementation of objects. The objects account for all software requirements assigned to each of the process tasks established in Step a.

c. Intermediate level SHARP abstracts are introduced to represent the structure of program units used to implement the internal complexities of an object implementation (i.e., the bodies of object tasks and program units visible in object packages). This structure is shown by SHARP Hierarchy and Invocation Diagrams.

d. Low level SHARP abstracts are introduced to represent selective detail. These abstracts can be thought of as "blow ups" of entities identified in SHARP Invocation Diagrams (e.g., data flow between program units, data structures and task rendezvous).

e. At the lowest level of design representation, annotated pseudo code is used to represent certain detail in the bodies of program units. This detail accounts for algorithms, logic, Ada exceptions, and Ada generic instantiation.

Item a. is discussed in Section 3.2 of Chapter 1. Item b. is discussed in Section 2 of Chapter IV. Items c. through e. are discussed in Section 2 of Chapter II. The remainder of this section provides an example of an Ada-unique design for the hypothetical restart/recovery process. Chapter IV describes the use of SHARP in conjunction with object-oriented design in more detail and presents a more complex example.

2.1.2 Example of an Ada Design for High Maintainability

In order to produce a highly maintainable object-oriented design, a designer decides to distribute the requirements for the restart/recovery process among three highly independent objects. He chooses to implement the objects using an Ada task and two Ada packages.

The first object is to provide an interface with an operator. An Ada task is chosen to implement this object, since it can rendezvous with the operator's work station in order to receive operator commands and notify the operator of software status.

The second object is to facilitate an abort of a transaction and establish a record of this event. An Ada package is chosen to encapsulate program units that implement this object.

The third object is to facilitate restart of a transaction and establish a record of this event. An Ada package is chosen to encapsulate program units that implement this object.

2.1.2a High Level Design Representation

The Ada task and packages chosen by the designer are shown in Figure 48. Task AP_NOTIFY interacts with an operator's work station. An operator can (a) abort execution of an application program, which is implemented by Procedure ABORT_TRANS in Package P2; (b) restart an application program from a breakpoint, which is implemented by Procedure START_BKPT in Package P3; or (c) restart an application program from its beginning, which is implemented by Procedure START_BEG in Package P3.

As shown in Figure 48, the object implementations can be considered as two layers of abstraction. Layer 1 consists of Task AP_NOTIFY. Layer 2 consists of Package P2, named ABORT_HANDLER, and Package P3, named RECOVERY_HANDLER. The interaction of these object implementations is shown by the SHARP Invocation Diagram is Figure 49.

Figures 48 and 49 are high level SHARP abstracts representing the designer's Ada-unique object-oriented design. Hidden complexities of object implementations are shown in intermediate and low level SHARP abstracts.

2.1.2b Intermediate Level Design Representation

Figure 50 uses SHARP Invocation Diagrams to represent the structure of program units called in the bodies of Procedure ABORT_TRANS (the visible procedure in Package 2), Procedure START_BEG (the first visible procedure in Package 3), and Procedure START_BKPT (the second visible procedure in Package 3).

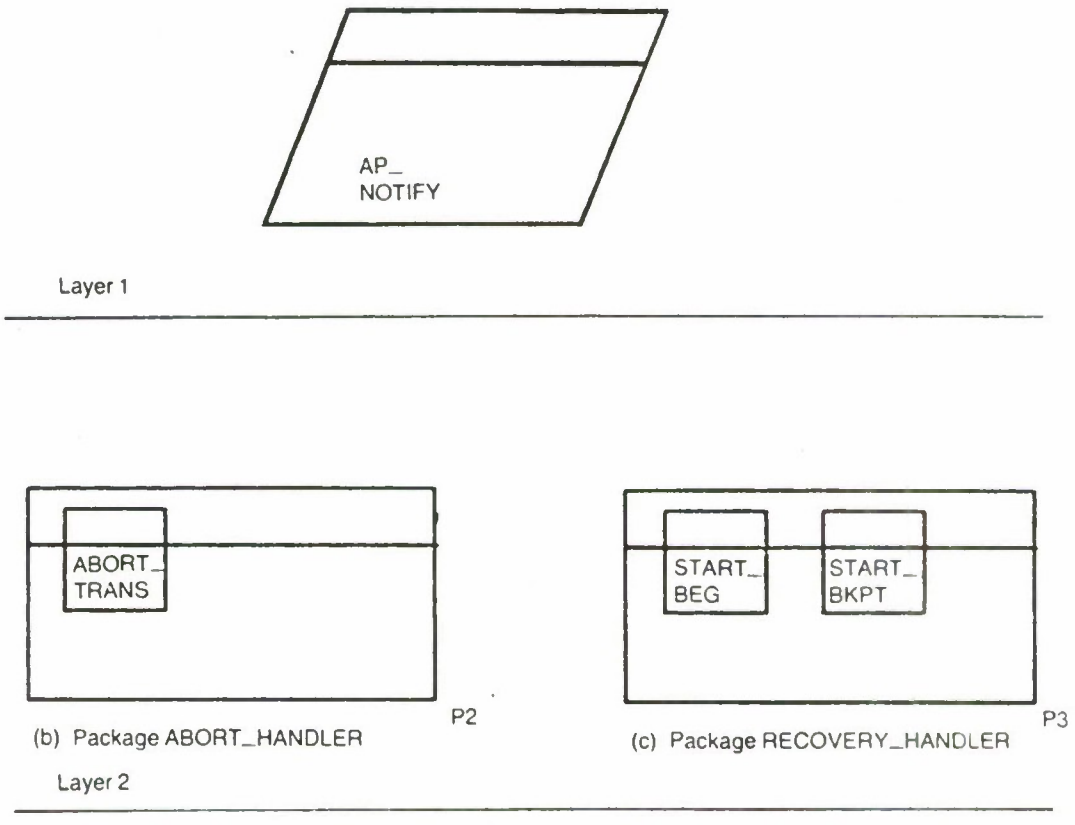


FIGURE 48. LAYERS (DESIGN FOR MAINTAINABILITY)

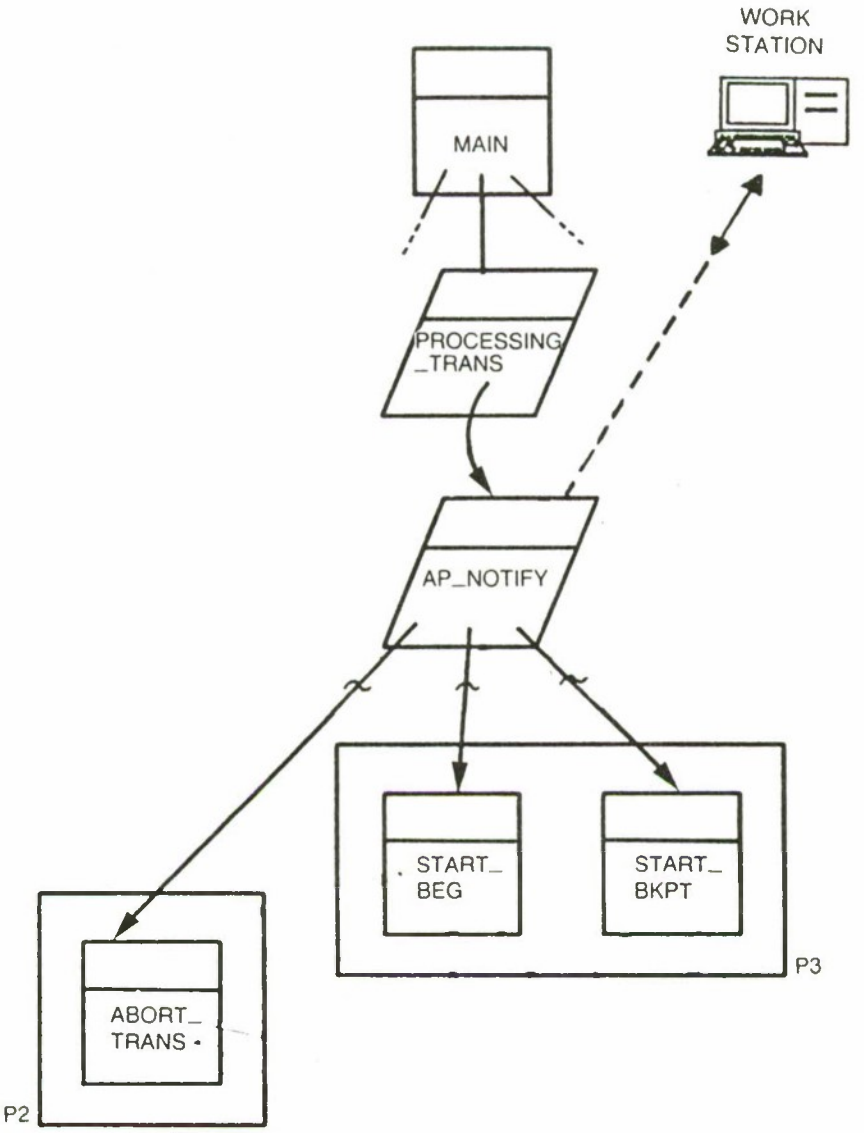
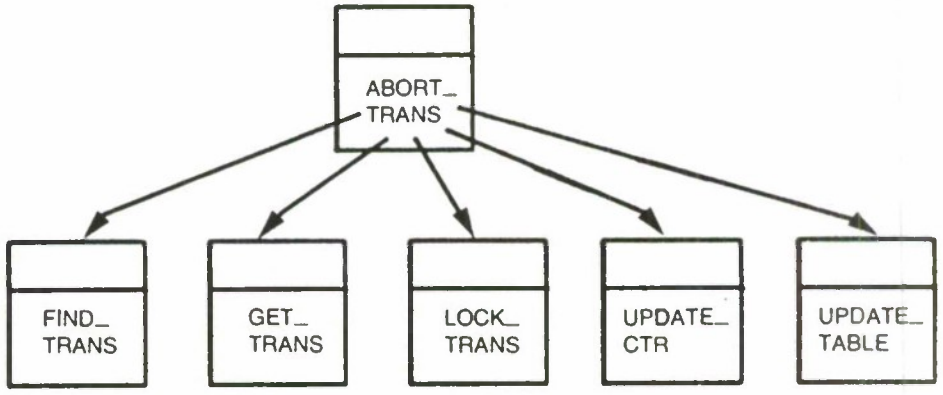
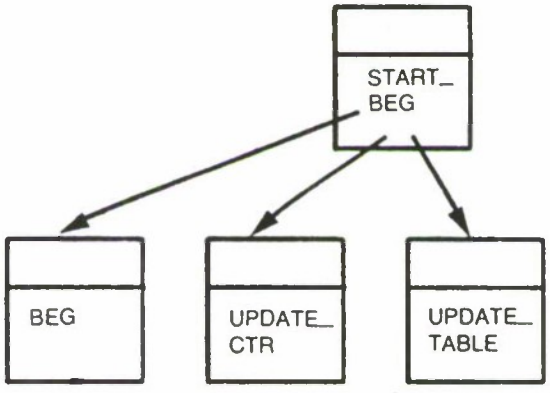


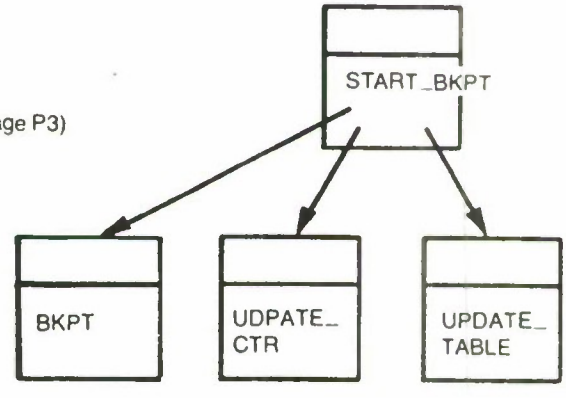
FIGURE 49. INTERACTION (DESIGN FOR MAINTAINABILITY)



(a) Procedure ABORT_TRANS (Package P2)



(b) Procedure START_BEG (Package P3)



(c) Procedure START_BKPT (Package P3)

FIGURE 50. INVOCATION DIAGRAMS FOR SELECTED ADA PROCEDURES

These diagrams are intermediate level SHARP abstracts. In large and complex computer programs, the bodies of program units visible in object tasks and packages may be complex. Accordingly, their design should typically be abstracted into understandable levels with constrained detail. Excluded detail is passed to the bodies of called program units. These bodies are, in turn, constrained to an easily understood amount of detail, with lower detail moved again to called program units.

2.1.2c Intermediate Level Design Representation

At a yet lower level of design abstraction, pictorial options of SHARP can be used to represent selective detail. Figure 51 provides SHARP Data Flow Diagrams to establish parameters passed between Task AP_NOTIFY (Layer 1) visible program units declared in the specifications of Packages P1 and P2 (Layer 2).

Figure 52 is a SHARP Task rendezvous Diagram representing a rendezvous between Task PROCESSING_TRANS AND TASK AP_NOTIFY, and between the workstation and Task AP_NOTIFY.

Figure 53 provides a set of Data Structure Diagrams showing the data structures established within Packages 2 and 3.

2.1.2d Lowest Level Design Representation

At the lowest level of design representation, SHARP junctions with traditional design representation techniques. Specifically, it utilizes annotated pseudo code to represent operations on variables and a glossary to define variables and other data structure detail.

Figure 54 shows the pseudo code used to represent logic in the body of Task AP_NOTIFY and in the bodies of Procedures ABORT_TRANS, START_BEG and START_BKPT.

2.2 DESIGN VARIATIONS

2.2.1 Design for Execution Speed

Various design options can be taken to increase execution speed. For example, since the most time consuming operation in the Ada language is task rendezvous, by limiting the use of tasks we gain speed of execution (and, as a by-product, simplify testing of Ada code).

The designer decides to increase execution speed by implementing Object 1 using procedures encapsulated in an Ada package, rather than Task AP_NOTIFY. Figure 55 shows his approach, where a procedure NOTIFY_OPERATOR has been established in a Package P1, named TRANS_HANDLER.

As shown in Figure 56, NOTIFY_OPERATOR is called by Task PROCESSING_TRANS. The procedure call is faster than the original task rendezvous between Task PROCESSING_TRANS and the Layer 1 object implementation in Task AP_NOTIFY.

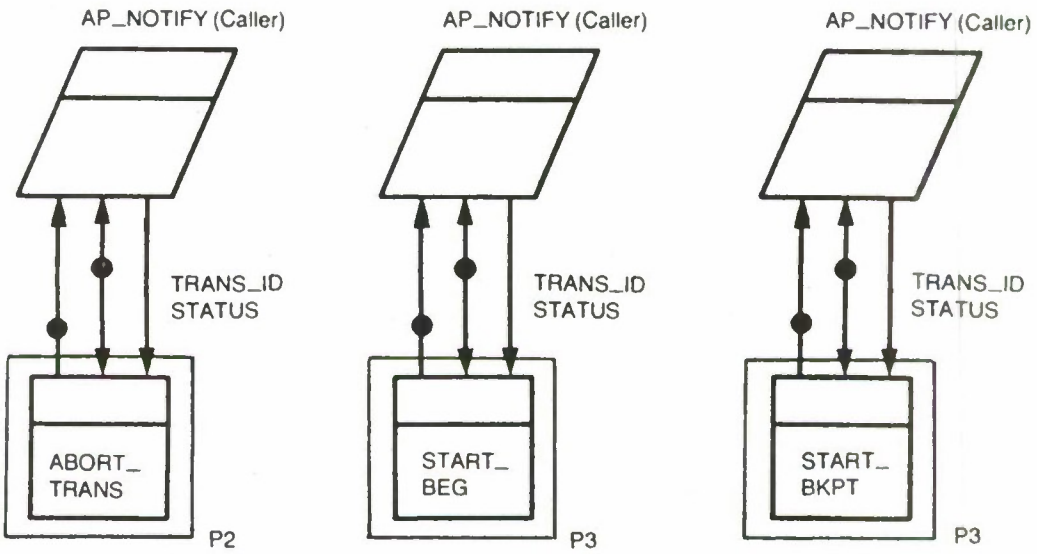


FIGURE 51. DATA FLOW DIAGRAMS

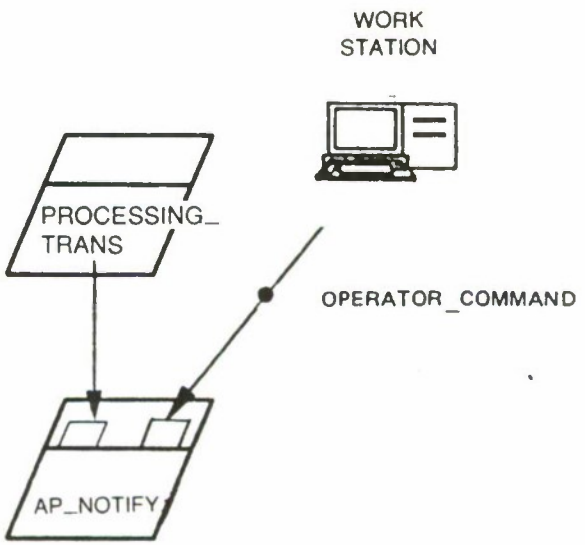
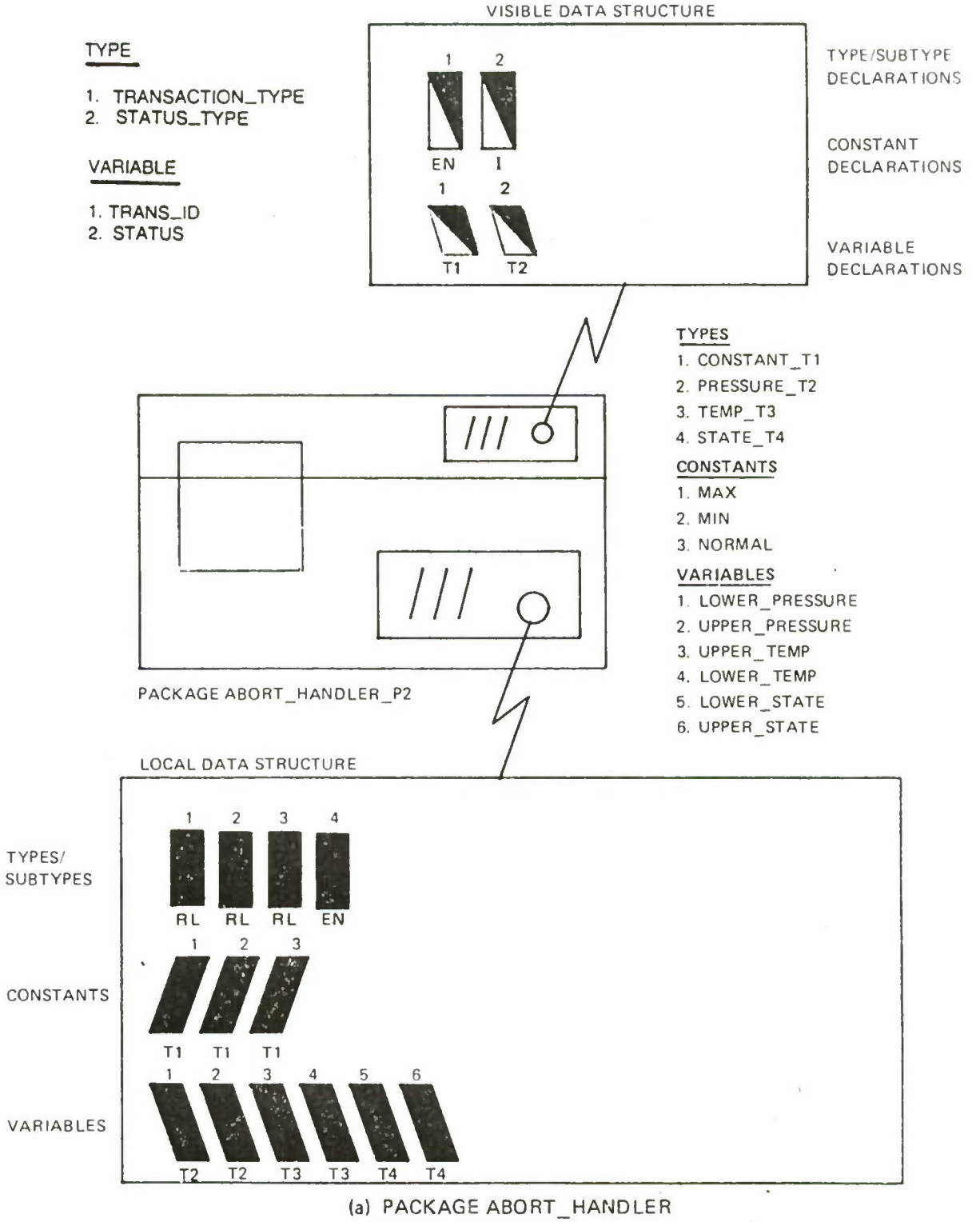
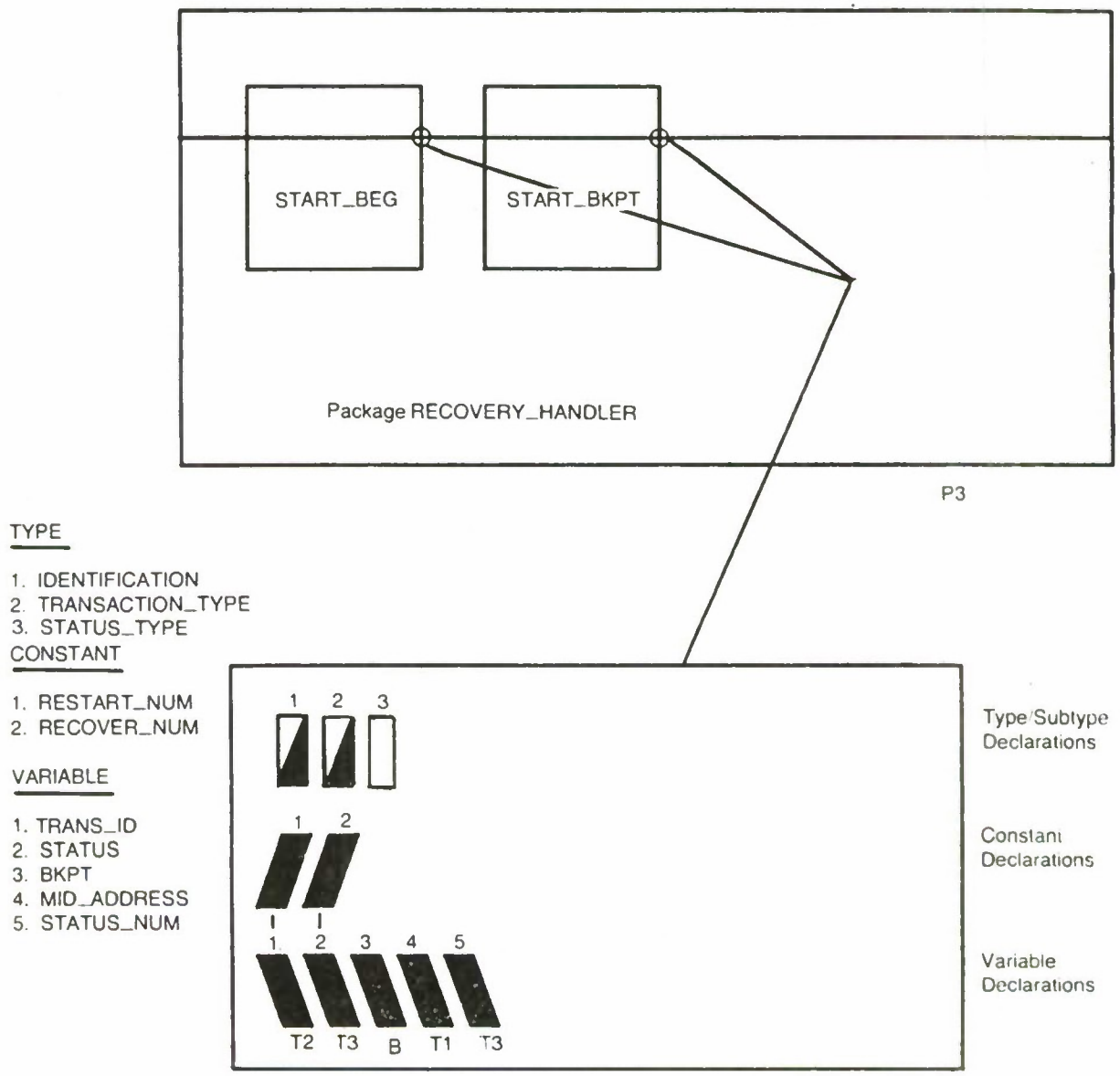


FIGURE 52. TASK RENDEZVOUS DIAGRAM



(a) PACKAGE ABORT_HANDLER

FIGURE 53. DATA STRUCTURE DIAGRAMS



(b) Package RECOVERY_HANDLER

FIGURE 53. (CONCLUDED)

```
Begin pseudo code for Task AP_NOTIFY

Receive operator command
Establish TRANS_ID
If TRANS_ID=ABORT then
     call Procedure ABORT_TRANS
     Receive and check STATUS to confirm 'abort'
     Inform operator of result

     Else if TRANS_ID = RESTART then
       call Procedure START_BEG
       Receive and check STATUS to confirm 'restart'
       Inform operator of result

       Else if TRANS_ID = BREAKPOINT then
          call Procedure START_BKPT
          Receive and check STATUS to confirm
               'restart from breakpoint'
          Inform operator of result

          Else notify operator of incorrect TRANS_ID code

end if

End pseudo code for Task AP_NOTIFY
```

(a) PSEUDO CODE FOR TASK AP_NOTIFY (T1)

```
Begin pseudo code for Procedure ABORT_TRANS

Receive TRANS_ID
Assess TRANS_ID to establish transaction to be aborted

To abort transaction
Call Procedure FIND_TRANS
Call Procedure GET_TRANS
Call Procedure LOCK_TRANS

To record abort
Call Procedure UPDATE_CTR
Call Procedure UPDATE_TABLE

End pseudo code for Procedure ABORT_TRANS
```

(b) PSEUDO CODE FOR PROCEDURE ABORT_TRANS (P2)

FIGURE 54. SHARP LOWEST LEVEL ABSTRACTION

```
Begin pseudo code for Procedure START_BEG

Receive TRANS_ID
Assess TRANS_ID to establish transaction to be restarted

To restart transaction
Call Procedure BEG

To record restart
Call Procedure UPDATE_CTR
Call Procedure UPDATE_TABLE

End pseudo code for Procedure START_BEG
```

(c) PSEUDO CODE FOR PROCEDURE START_BEG (P3)

```
Begin pseudo code for Procedure START_BKPT

Receive TRANS_ID

Assess TRANS_ID to establish transaction to be
     restarted from a breakpoint
Establish location of breakpoint

To restart transaction from breakpoint
Call Procedure BKPT

To record restart from breakpoint
Call Procedure UPDATE_CTR
Call Procedure UPDATE_TABLE

End pseudo code for Procedure START_BKPT
```

(d) PSEUDO CODE FOR PROCEDURE START_BKPT (P3)

FIGURE 54. (CONCLUDED)

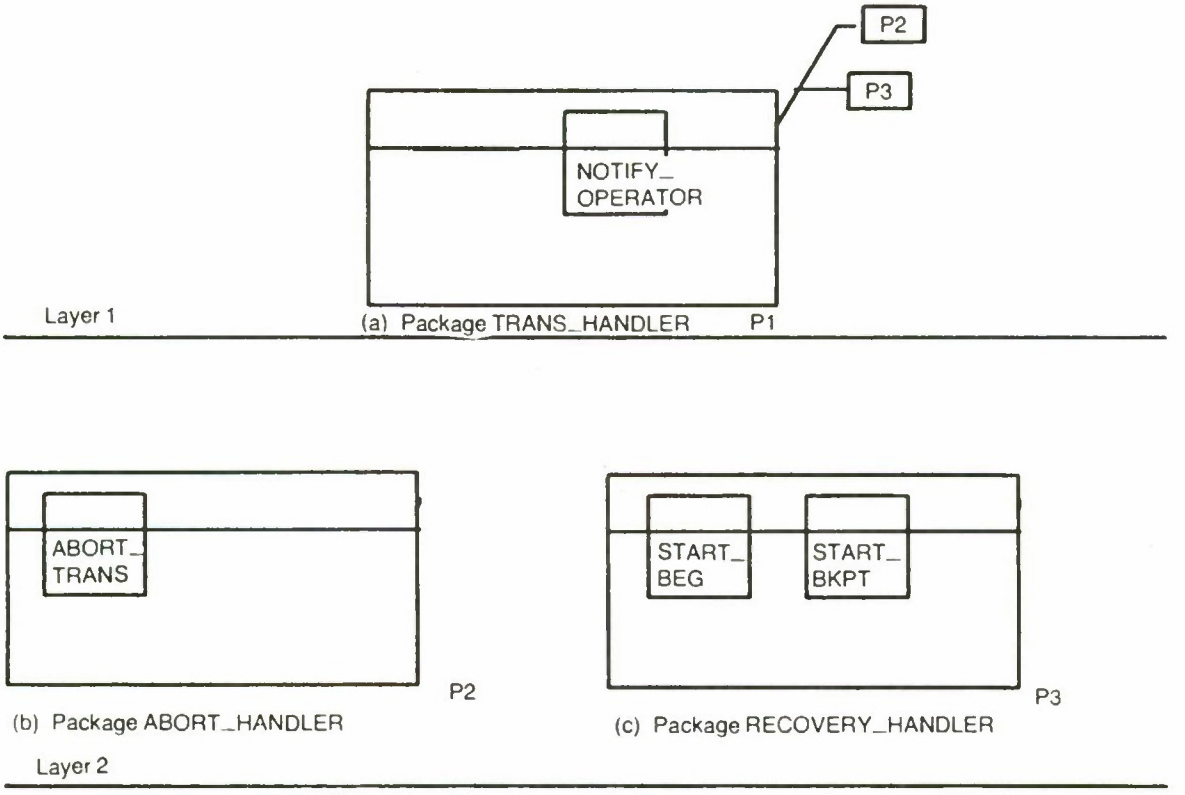


FIGURE 55. LAYERS OF PACKAGES (DESIGN FOR SPEED)

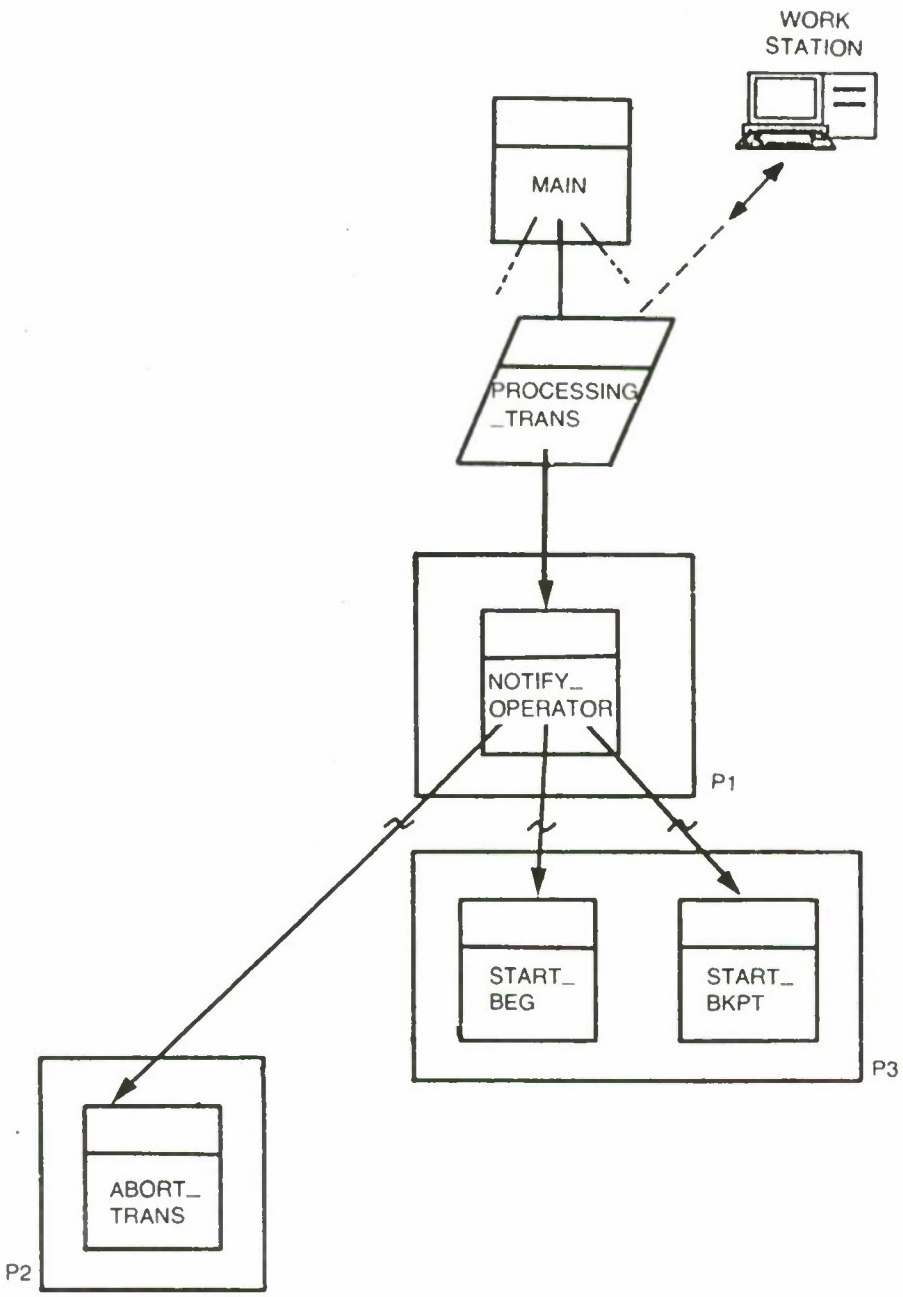


FIGURE 56. PACKAGE INTERACTION (DESIGN FOR SPEED)

With this design, Task PROCESSING_TRANS has been assigned the responsibility of interacting with the operator's work station through on Ada task rendezvous. Otherwise, the implementation of Object 1 internal detail is essentially the same using a package as it was using a task.

2.2.2 Design Subject to Memory Constraint

There may be times when a military system must be designed with some limitation on the availability of memory. Although the latest advancements in hardware have made commercial memory inexpensive and extensively available, this is not necessarily the case with a defense system. The cost of "ruggedized" memory chips used in defense systems is much greater than the commercial counterpart. Also, since space in a military system is often limited, the physical space constraint can limit the amount of memory available. This is especially true in avionics systems.

In order to save memory in the implementation of the restart/recovery process, the designer decides to establish common Ada program units. Figure 57 shows his approach, where Procedures UPDATE_CTR and GEN_TABLE have been removed from Packages P2 and P3 and inserted into a Package 4, named R_R_SERVICES.

These procedures establish a record of the transaction abort or restart. Specifically, Procedure UPDATE_CTR updates variables containing transaction status (i.e., aborted or restarted from the beginning or a breakpoint). Procedure UPDATE_TABLE tracks all transactions, keeping a record of active, halted and inactive transactions.

This design variation saves memory by establishing the package of common program units. However, the implementation of Object 2 in Package 2 and Objects 3 in Package 3 is no longer completely independent of each other, since they are now coupled through the use of common program units in Package P4, as shown in Figure 58.

3 CHAPTER SUMMARY

SHARP establishes a set of pictorial abstracts that can be used to represent an object-oriented design for an Ada computer program. With such a design, we can localize design complexity, thus reducing interdependence relationships and thereby facilitating cost effective software maintenance.

In the past, large global sets of parameters and routines were extensively shared. If any one of the global variables or routines were modified, a "domino effect" was introduced in that the change affected several different parts of the computer program. Seemingly innocent changes caused traumatic problems. By segmenting a program into objects, and constraining the interface between the objects, software maintenance becomes a simpler and less expensive problem. This is important to the Air Force since historically software maintenance costs have been very high.

In practice, a designer of a large and complex Ada computer program may have to face certain design constraints as he develops his object-oriented design. For example, certain objects may have execution speed constraints

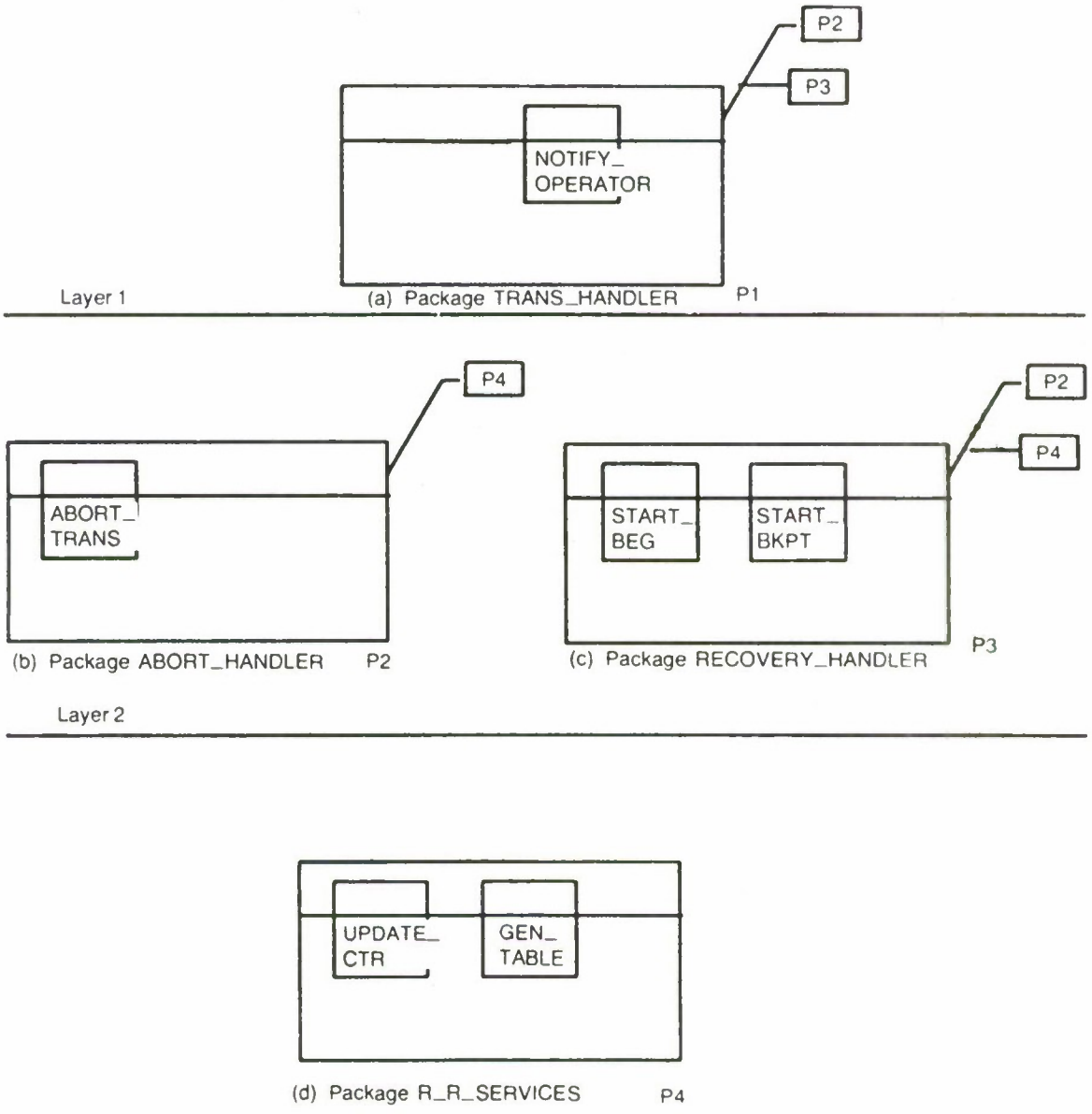


FIGURE 57. LAYERS OF PACKAGES (DESIGN FOR LIMITED MEMORY)

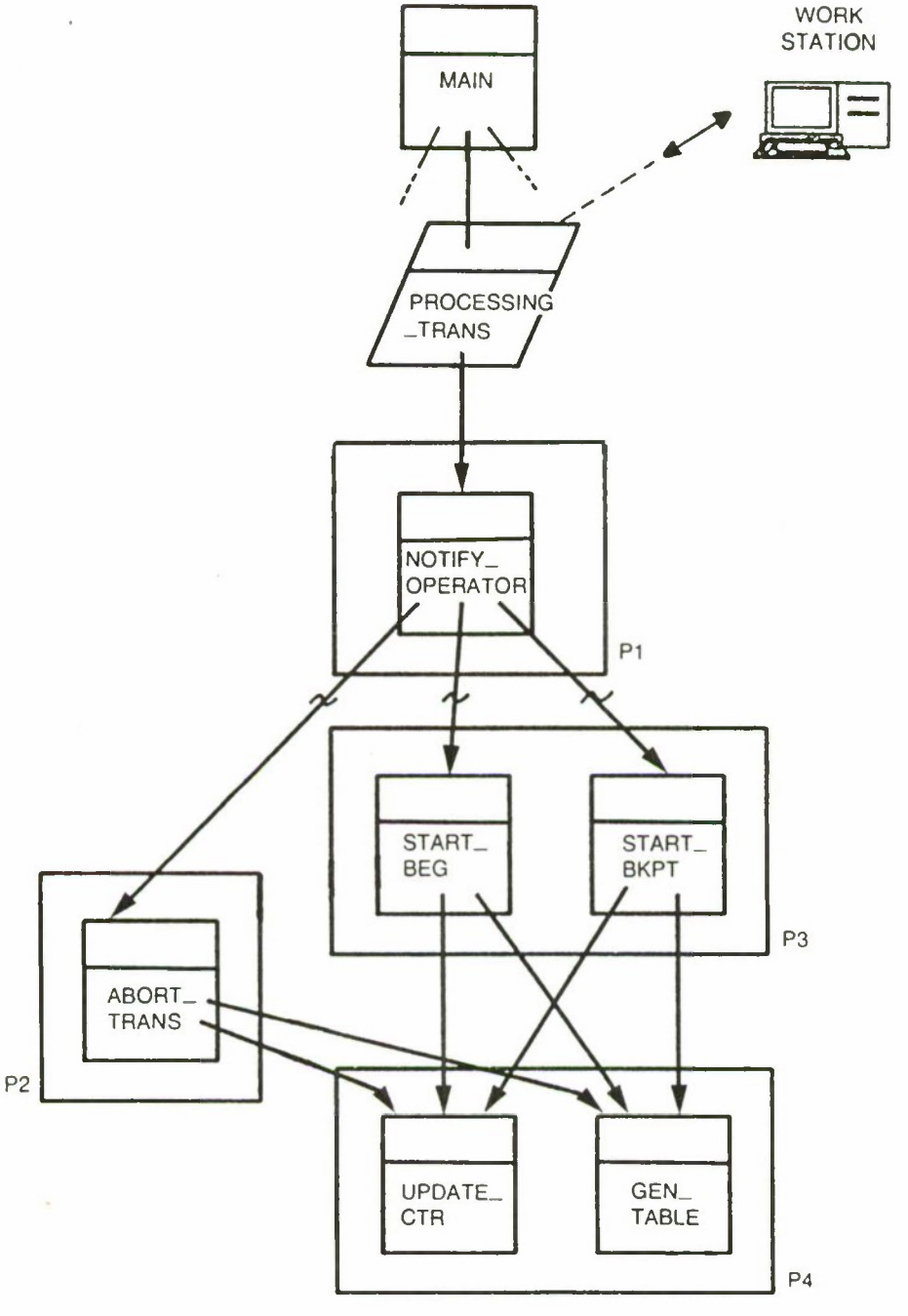


FIGURE 58. PACKAGE INTERACTION (DESIGN FOR LIMITED MEMORY)

while others may have memory constraints. These constraints may have to be introduced at the expense of some of the power of the object-oriented design (e.g., a compromise to the extent of object independence may have to be introduced). Therefore, the final form of the object-oriented design may be affected by design goals conflicting to some extent with the goals of an object-oriented design with high maintainability.

Whatever the design goals are, the ultimate design can be effectively represented by SHARP abstracts. At the highest level of the design, the abstracts represent Ada packages introduced to encapsulate objects or sets of objects. The interaction of the objects is represented by SHARP invocation diagrams, and the bodies of program units responsible for communication between objects, visible within each task or package encapsulating an object, can be represented by the SHARP Hierarchy and Invocation Diagrams. Abstracts envisioned as "blow ups" of entities identified in an invocation diagram can be used to represent details of task rendezvous, data flow between program units and data structures. The later diagram shows information hiding within data structures local to objects. At the lowest level of SHARP, annotated pseudo code is used to represent operations and logic within the bodies of individual units.

CHAPTER IV

Steps for an Object-Oriented Ada-Unique Design

This chapter presents steps that can be used to design an Ada computer program using principles of object-oriented design. It provides an example of applying the steps to develop and represent the design using SHARP.

1 INTRODUCTION

1.1 BACKGROUND

Projections for the amount of software to be implemented using Ada are very large and will incur a high acquisition cost if developed using traditional methods. The high cost, and typical schedule slippage of software development efforts in the past, can be attributed to several factors, including the effect of complex dependency relationships between types, variables and program units. However, as we have indicated, dependency relationships can be controlled using Ada-unique object-oriented techniques, since loosely coupled Ada packages and tasks can be developed and tested largely independently.

Given the size and complexity of projected software systems, the government software review process, and the large development teams, there is a need for representing a software design in an understandable and abstracted manner.

1.2 CHAPTER SCOPE

In this chapter, selective detail of an Ada-unique object-oriented design is presented for a command and control computer program of a hypothetical space station. The use of each level of SHARP abstracts is demonstrated in representing the design of the command and control computer program.

Section 2 presents a set of steps that can be used to establish the design of a large and complex computer program to be implemented using Ada. Section 3 presents SHARP abstracts that document the results of the steps when applied to represent the design of the command and control computer program.

2 OBJECT-ORIENTED ADA DESIGN

2.1 INTRODUCTION

When using object-oriented design techniques during software development, the requirements for a large computer program are distributed among objects. The implementation of each object has a unique set of operations and a local state defined in a data structure. The unique operations are known only to the object implementation. Access to an object implementation can only be made via calls to program units in packages or entry points to tasks introduced to implement the object.

As described in Chapter II, the implementations of objects hide information about their internal representation and in a sense, mimic real world objects, such as telephones. Such real world objects present a small number of basic operations that can be performed on them (e.g., dial number, answer, hang up) and hide the relatively complex details of their implementation. Since the objects are essentially independent of each other, one object implementation can be modified without affecting a second. For example, one telephone receiver can be modified or replaced without affecting others.

In Ada, packages and tasks are important to the implementation of object-oriented designs. For example, information hidden within a package limits dependency relationships between objects. Only parameters declared in program units, contained within the specification of a package, can be passed from one package to another.

2.2 STEPS FOR ESTABLISHING AN OBJECT-ORIENTED DESIGN

A software engineer must take a series of steps to establish the design of a large and complex computer program. A basic set of steps that take into account principles of object-oriented and structured top-down design, and the SHARP graphics used in conjunction with each step, are summarized in the following paragraphs.

2.2.1 Step 1 - Establish Processes

For a given set of software requirements, in Step 1 we identify processes needed to establish concurrent processing threads referred to as processes, and we assign requirements to each process as appropriate. With Ada, each process can be established by a task declared in the main program, as illustrated by the SHARP pictographs shown in Figure 59. As discussed in Chapter I, processes consist of abstracted threads of program units needed to service user requests from work stations and other hardware interfaces (e.g., communication links); and to perform processing automatically initiated on a periodic or some other basis.

2.2.2 Step 2 - Establish Objects for Each Process

In Step 2, we identify objects specified in the requirements assigned to each process. With respect to computer programs, an object is a system component implemented in software using a set of operations unique to it and a local data structure not accessible by entities external to it. With Ada, objects can be implemented by tasks, or by packages as illustrated in Figure 60.

For each process, objects will account for data reception, storage, manipulation and output. Data reception objects might be protocol managers, command interpreters, message handlers, sensor monitors, and external device data receivers. Data storage objects might be file or data base managers. Data manipulation objects might be controllers, planners, operations managers, trackers, detectors, and testers. Data output objects might be display, tape or hard copy data generators, as well as data exporters to communication links, distributed processors or other hardware devices.

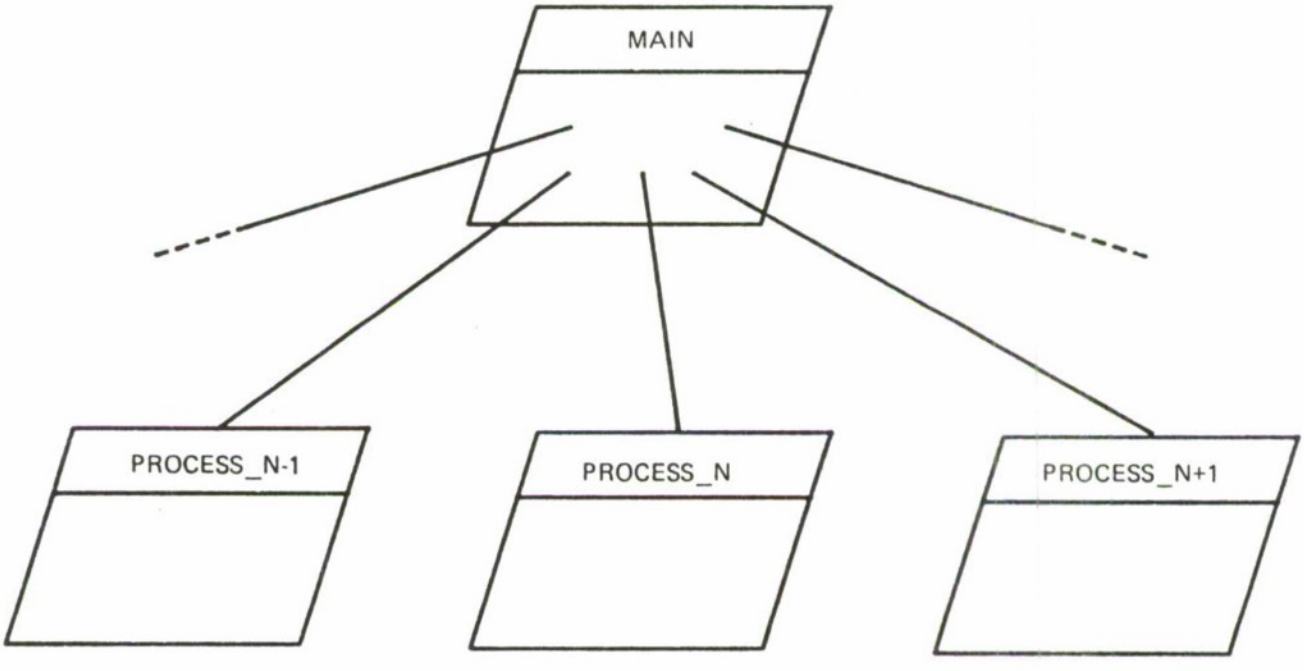


FIGURE 59. STEP 1 – ESTABLISH PROCESSES

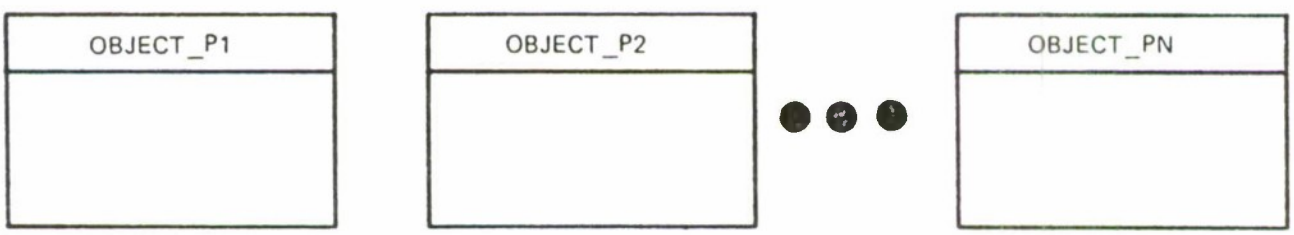


FIGURE 60. STEP 2 – ESTABLISH OBJECTS FOR EACH PROCESS

An experienced designer will group together requirements for each of the identified objects. As the designer makes these groupings, he will keep in mind that each object should be implemented with unique operations, and a local hidden data structure of variables and constants needed to facilitate the unique operations. He will also keep in mind that (a) the object's local hidden data structure should not overlap with a visible data structure that facilitates interobject communication; (b) the characteristics of the requirements should be consistent (e.g., requirements for data base management should not be mixed with requirements for such things as statistical processing and display); and (c) the requirements assigned to an object should be constrained, to the extent possible, so that the implementation of the object is relatively easy to understand, implement, compile and test.

Less experienced designers might use a more mechanical approach suggested by Booch, where nouns in the software requirements specification are candidate objects and verbs identify operations on the objects.[8] This approach to object identification is described in detail in Section 3.3 of an Object Oriented Design Handbook for Ada Software [7].

2.2.3 Step 3 - Establish Interfaces Between Objects

In Step 3, we establish parameters to be passed between the implementation of objects and the program units to be used to facilitate the parameter passing, as illustrated in the first part of Figure 61. With Ada, procedures and tasks declared in the specification of an object package can be used to facilitate such parameter passing, as illustrated by the SHARP Invocation diagram in the second part of Figure 61. As shown, the packages can be grouped into layers in an hierarchical manner, for clarity and ease of testing.

As a general rule, parameters used in the formulation of object implementations should not be passed between the implementations, which would couple the implementations. If they have to be passed, they should be made private or limited private. In this way, the receiving object package has limited use of the passed parameters. For example, if the parameter is private, the user is excluded from applying operations on the parameter other than those operations defined within the package specification. The only exception to this rule is assignment and tests for equality. If the parameter is limited private, assignments and tests for equality are no longer automatically available.

2.2.4 Step 4 - Establish Hidden Internal Design of Each Object

2.2.4a Establish Internal Structure of Each Object Implementation

In Step 4, we establish the structure for the internal complexities of an object implementation using a traditional structured/top-down approach and abstraction of detail into levels. For example, a relatively small and

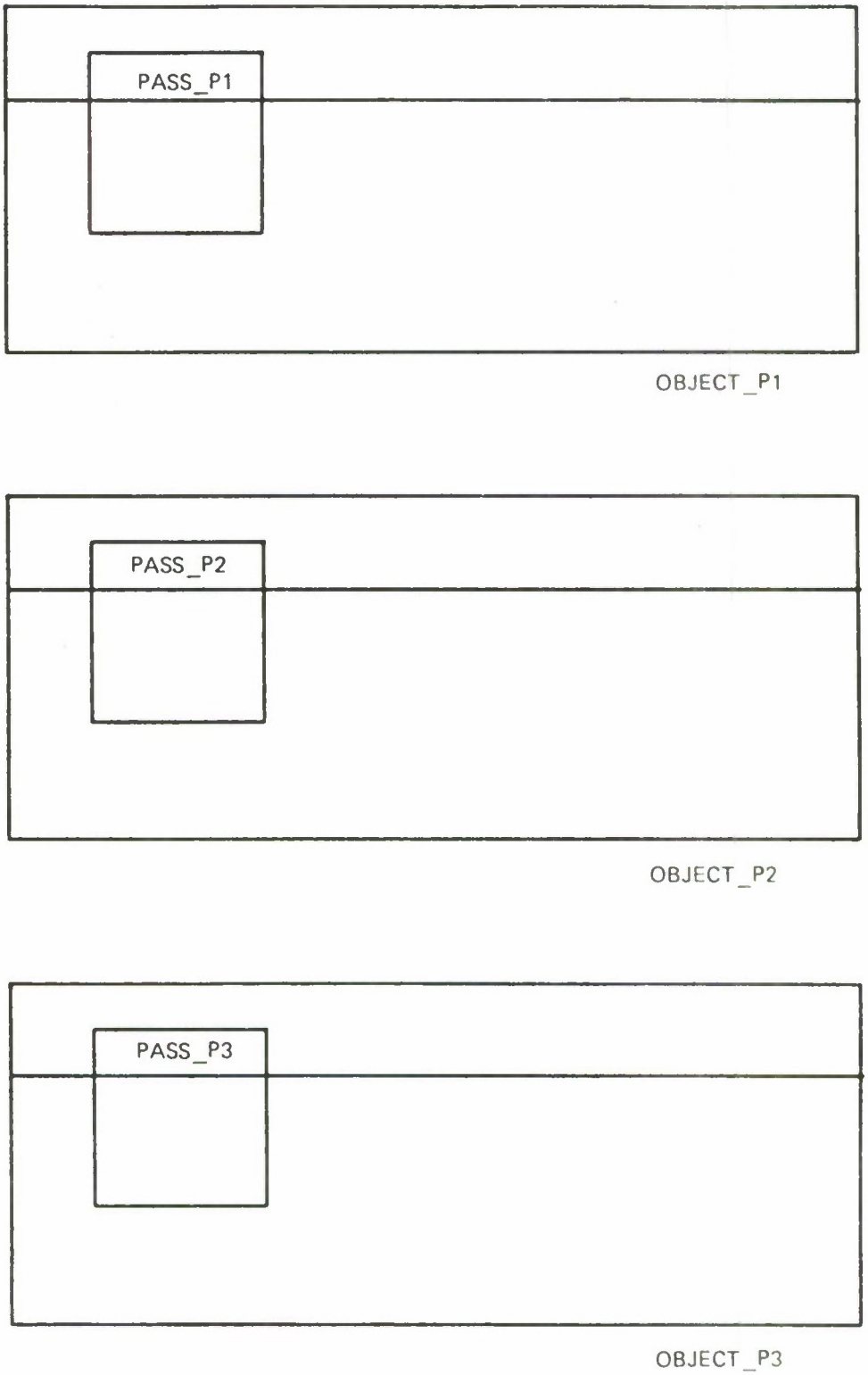


FIGURE 61. STEP 3 – ESTABLISH OBJECT INTERFACES

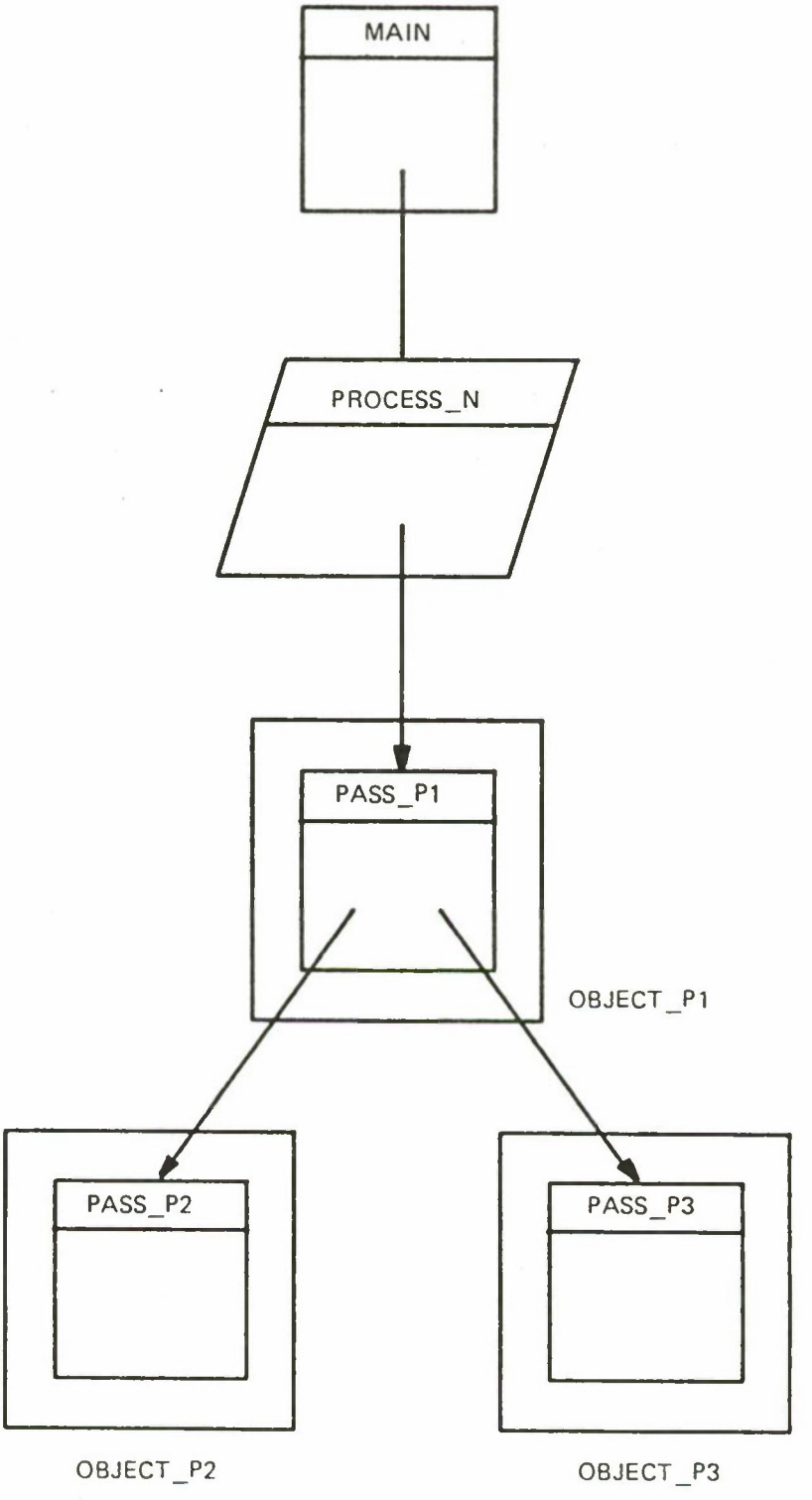


FIGURE 61. (CONCLUDED)

easily comprehended portion of detail can be implemented at one level, with the remainder of the detail implemented in called program units at lower levels. At each lower level, the abstraction process is repeated. With Ada, a controlling structure of subprograms and tasks can be assigned to the levels, and packages (e.g., containing existing or common program units) can be accessed by the subprograms and tasks using the Ada "with" clause, as illustrated by the SHARP Hierarchy Diagram in Item a of Figure 62 and the SHARP Invocation Diagram in Item b.

2.2.4b Establish Data Flow Between Program Units Internal to Each Object Implementation

In Step 4, we also establish the details of interaction between the internal program units defined in Substep 4a. With Ada, calls are made to pass parameters from one subprogram to another, as illustrated by the SHARP Subprogram Data Flow Diagram shown in Item a of Figure 63; and with Ada, task rendezvous is introduced for passing parameters from one task to another, as illustrated by the SHARP Task Rendezvous Diagram shown in Item b of Figure 63.

2.2.4c Establish Annotated Pseudo Code for Program Unit Bodies

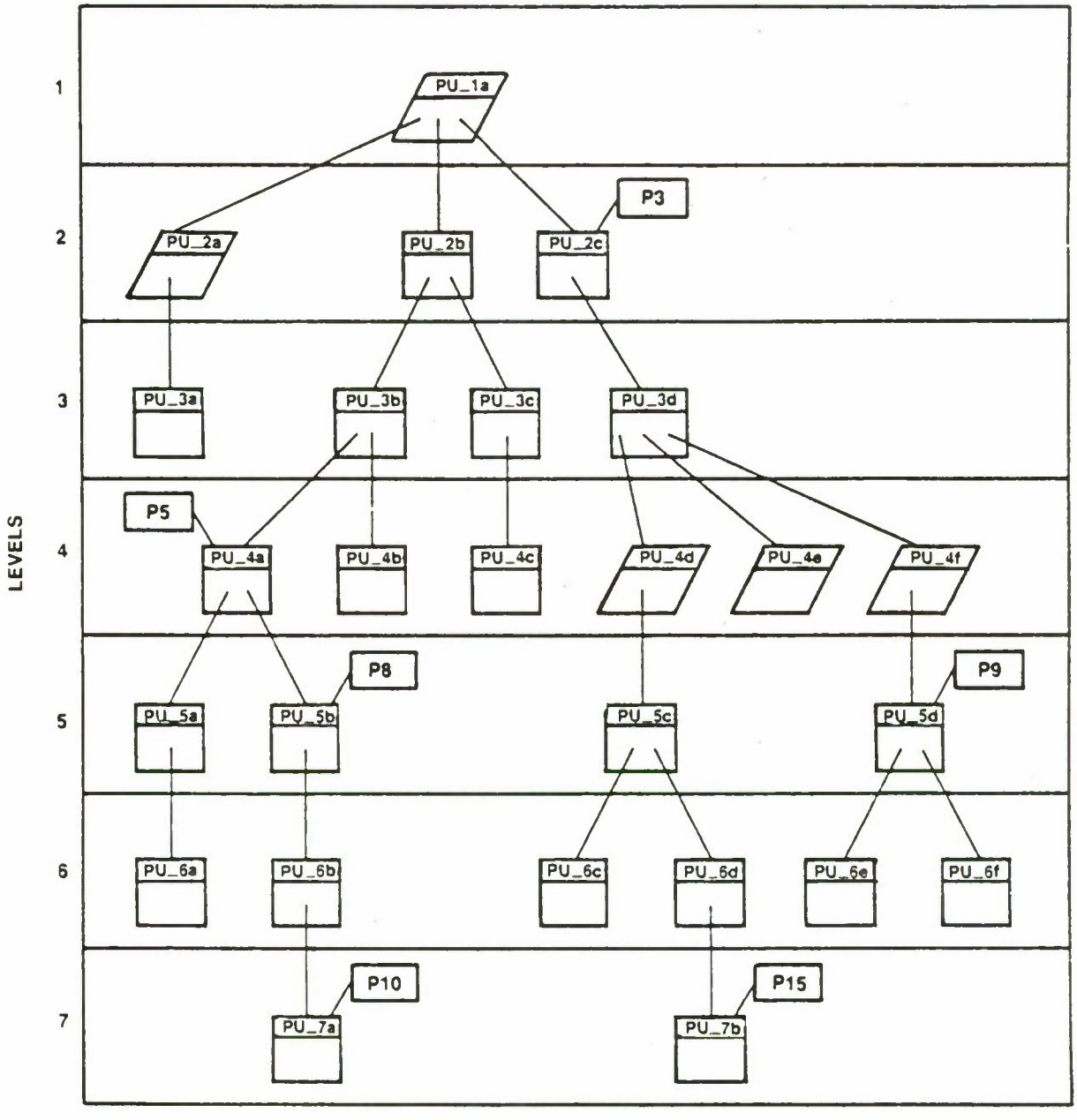
For internal program units used to implement an object, in Step 4 we furthermore establish processing to be undertaken in the bodies of these program units, including the algorithms, operations on variables, decisions, generic instantiations, exceptions, high-level and low-level I/O, decisions and other logic. We represent this information in annotated pseudo code. For example, Figure 64 shows pseudo code that could be used to represent the body of an Ada procedure.

2.2.4d Establish Data Structure of Each Object Implementation

To complete Step 4, we establish the data structures for each object implementation. Figure 65 illustrates the definition of types, variables and constants for an object implemented within an Ada package. As shown, passed parameters are declared in a visible data structure, while variables used to facilitate operations unique to the object implementation are hidden in the local data structure.

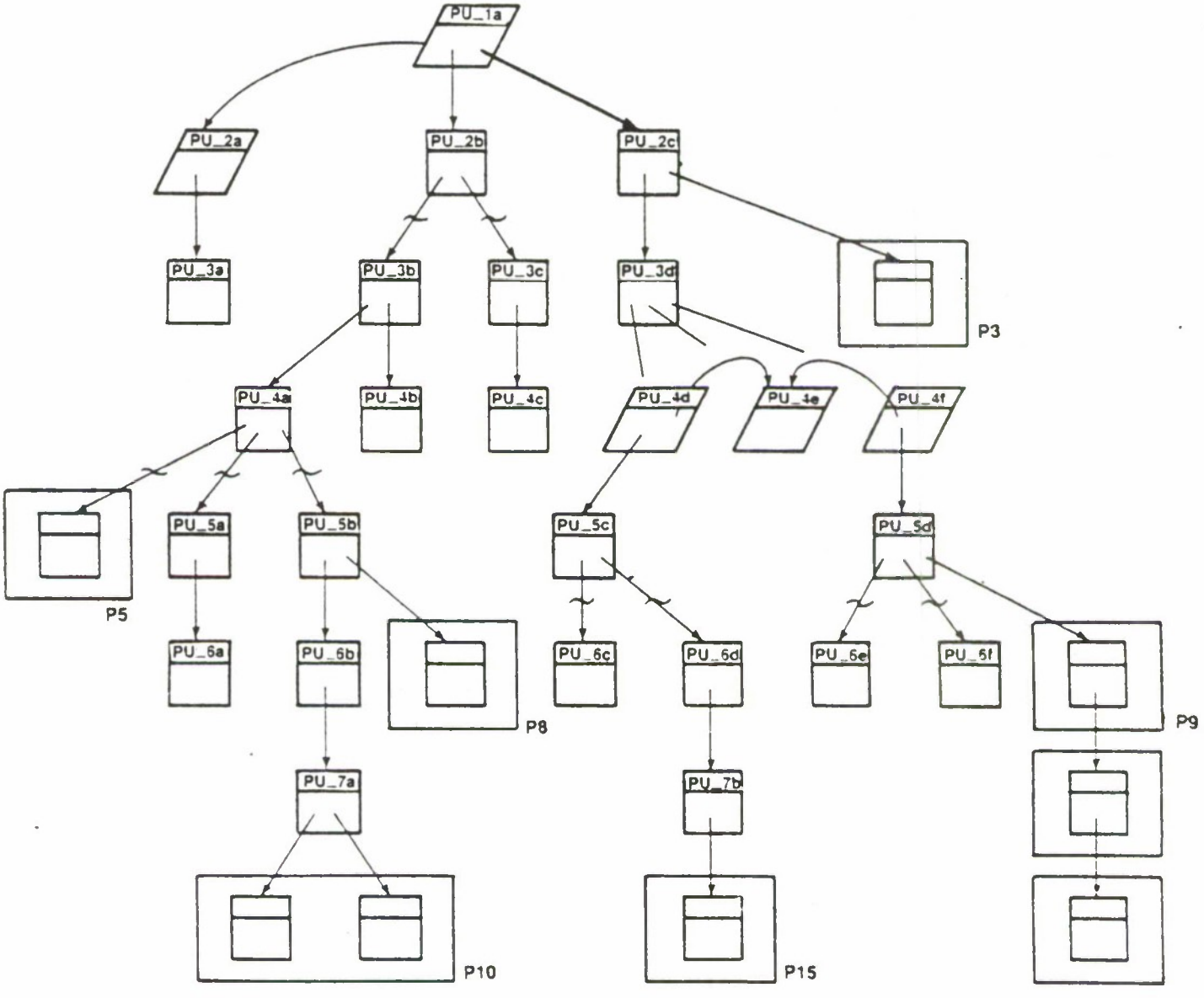
With SHARP an integer type is represented by an upright narrow rectangle with the letter "I" underneath it; a real type with the letters "RL"; and a enumeration type with the letters "EN." Furthermore, with SHARP an array type is represented by an upright narrow rectangle with the letters "AR" underneath it; a record type with the letter "R," underneath it; a discriminated type with the letter "D;" and a task type with the letters "T."

The type of a variable or constant is represented by the first letter of a predefined type (e.g., I for INTEGER); and by the letter "T" followed by the type glossary number (e.g., T2 for the 2nd type) for a defined type. See Section 2.4.4 of Chapter II for a detailed discussion of SHARP Data Structure Diagrams.



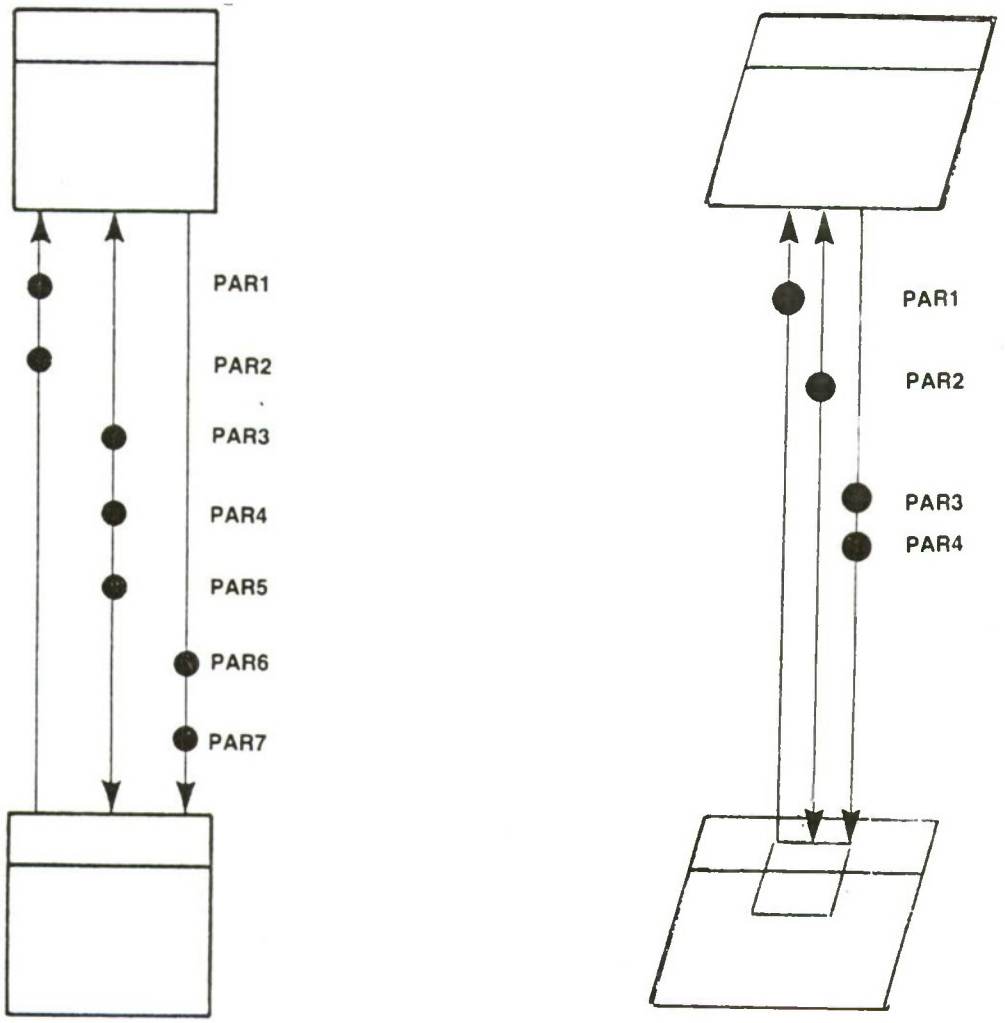
(a) SHARP HIERARCHY DIAGRAM

FIGURE 62. STEP 4 – ESTABLISH INTERNAL STRUCTURE OF EACH OBJECT



(b) SHARP INVOCATION DIAGRAM

FIGURE 62. (CONCLUDED)



(a) DATA FLOW DIAGRAM

(b) TASK RENDEZVOUS DIAGRAM

FIGURE 63. STEP 4 – ESTABLISH DATA FLOW BETWEEN PROGRAM UNITS INTERNAL TO EACH OBJECT

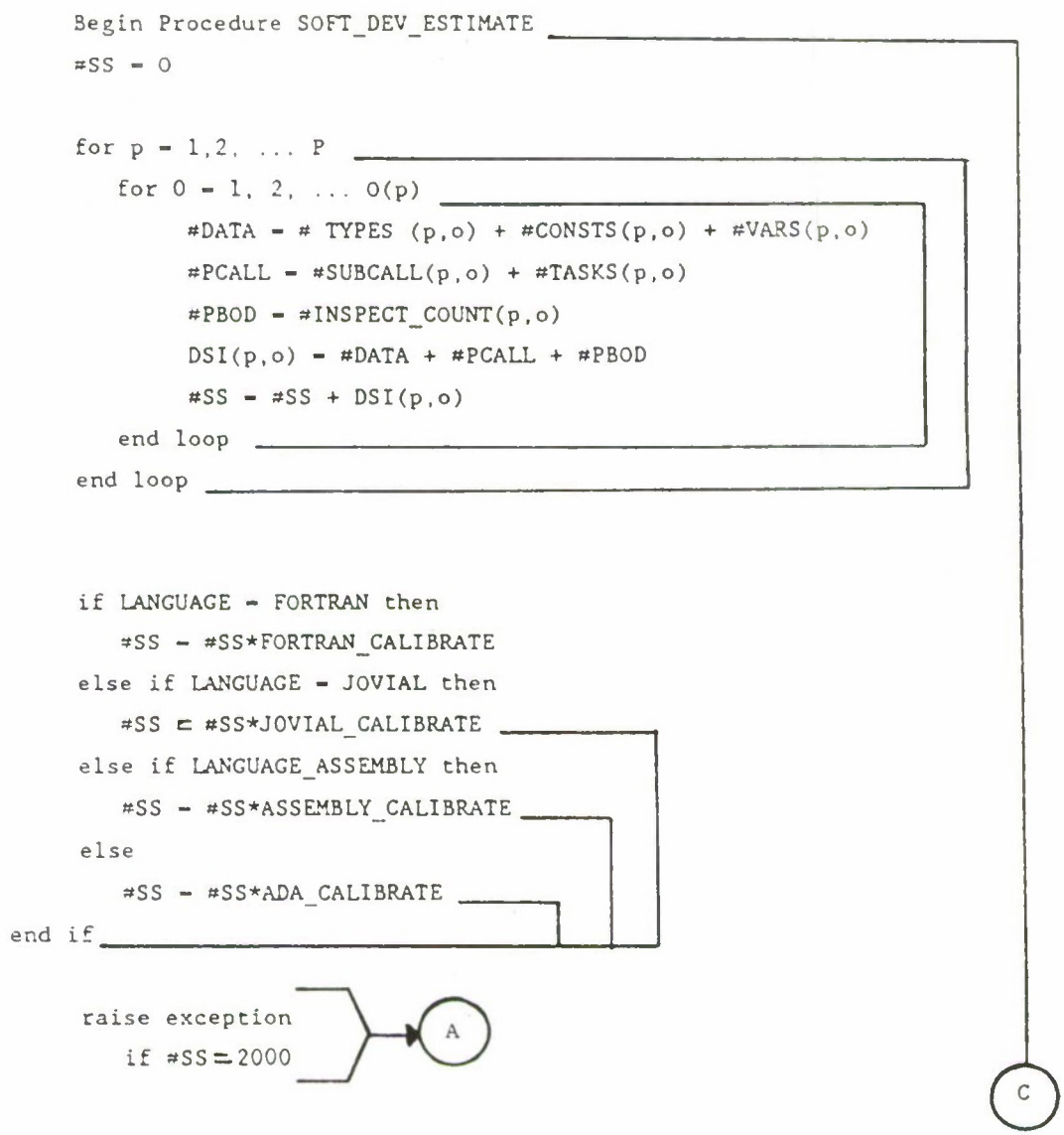


FIGURE 64. STEP 4 – ESTABLISH ANNOTATED PSEUDO CODE FOR PROGRAM UNIT BODIES

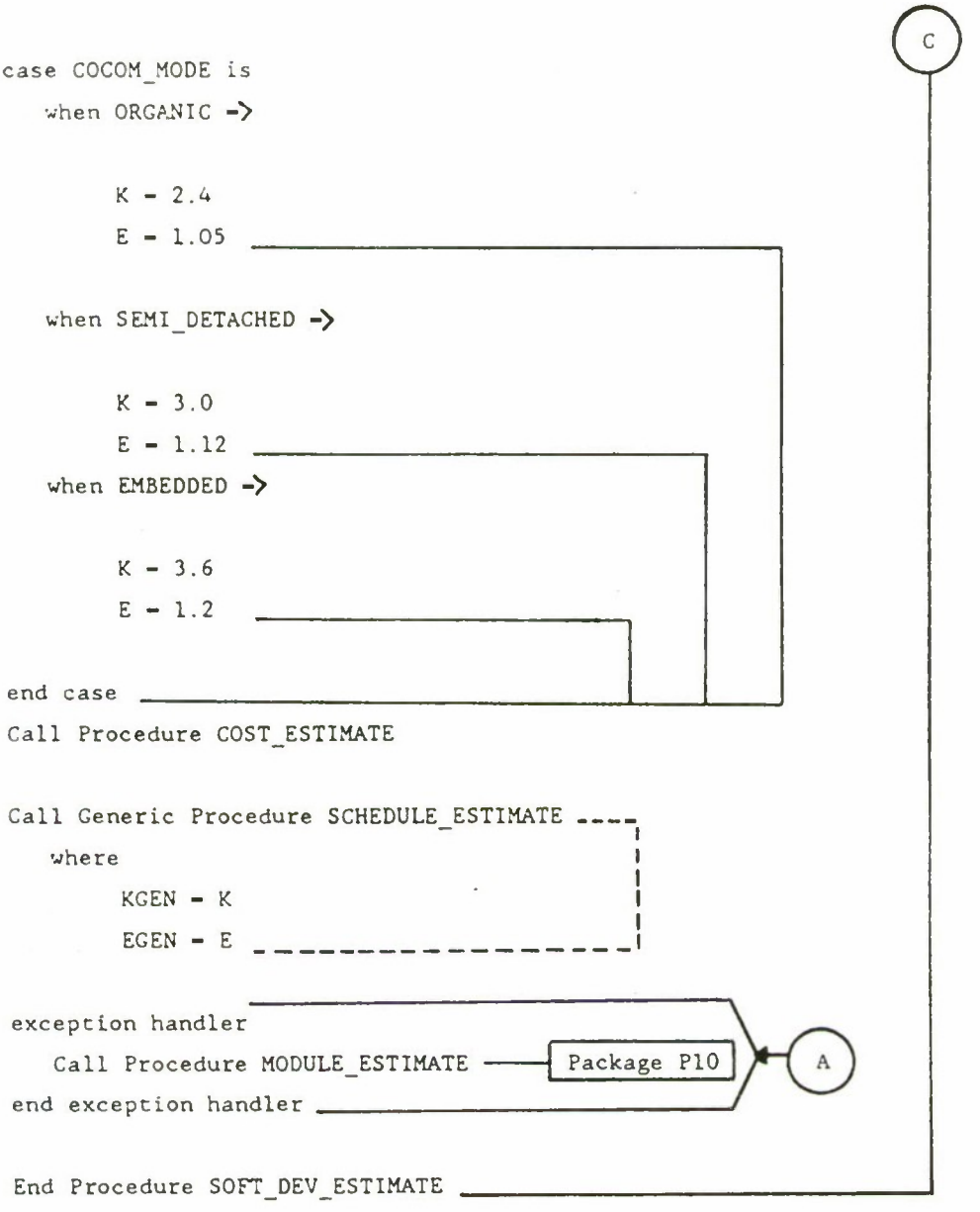


FIGURE 64. (CONCLUDED)

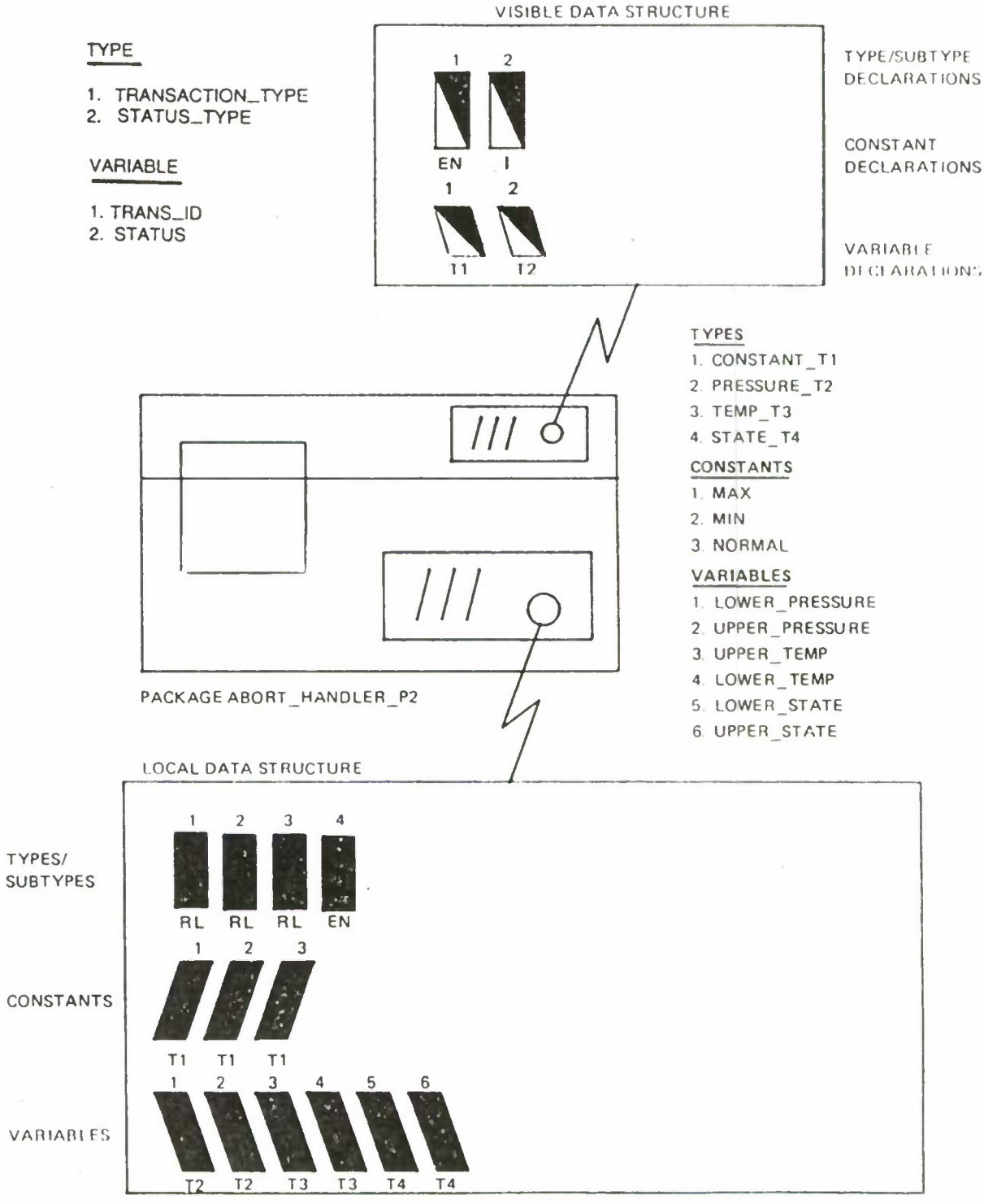


FIGURE 65. STEP 4 – ESTABLISH DATA STRUCTURE FOR EACH OBJECT

2.2.5 Step 5 - Refine the Design

In Step 5, we check the design for consistency and correctness. For example, we compare parameters passed between program units to those defined in the data structure diagram. We also check to make sure the implementation of each object is relatively easy to understand, implement, compile and test. For example, we could estimate the number of Ada source statements required to implement each object, using the algorithms defined in Chapter VI. We then could calculate the approximate time to compile each object implementation. For each object that is too large (and therefore, takes too long to compile), we would partition it into a set of sub-object implementations as illustrated in Items a and b of Figure 66. Then we would repeat Steps 3 and 4 for the group of sub-object implementations. The sub-objects would be used to construct the original object (i.e., small objects can be used to build large objects) or would be used to replace the original object, as illustrated in Item c of Figure 66.

3 EXAMPLE

3.1 INTRODUCTION

In this section, we apply the software design steps described in Section 2 to establish an Ada-unique design for the command and control software associated with the hypothetical earth orbiting space station. As specified in Appendix D, the software provides capabilities to (a) collect and process experimental data, (b) monitor sensors, (c) orient solar panels, and (d) perform built in tests of system processing hardware.

3.2 ESTABLISHING PROCESSES (STEP 1)

The first step in the design of a large and complex computer program is to identify the top level concurrent processing threads needed to implement software requirements. In the space station example, three processes can be established to implement the following software requirements:

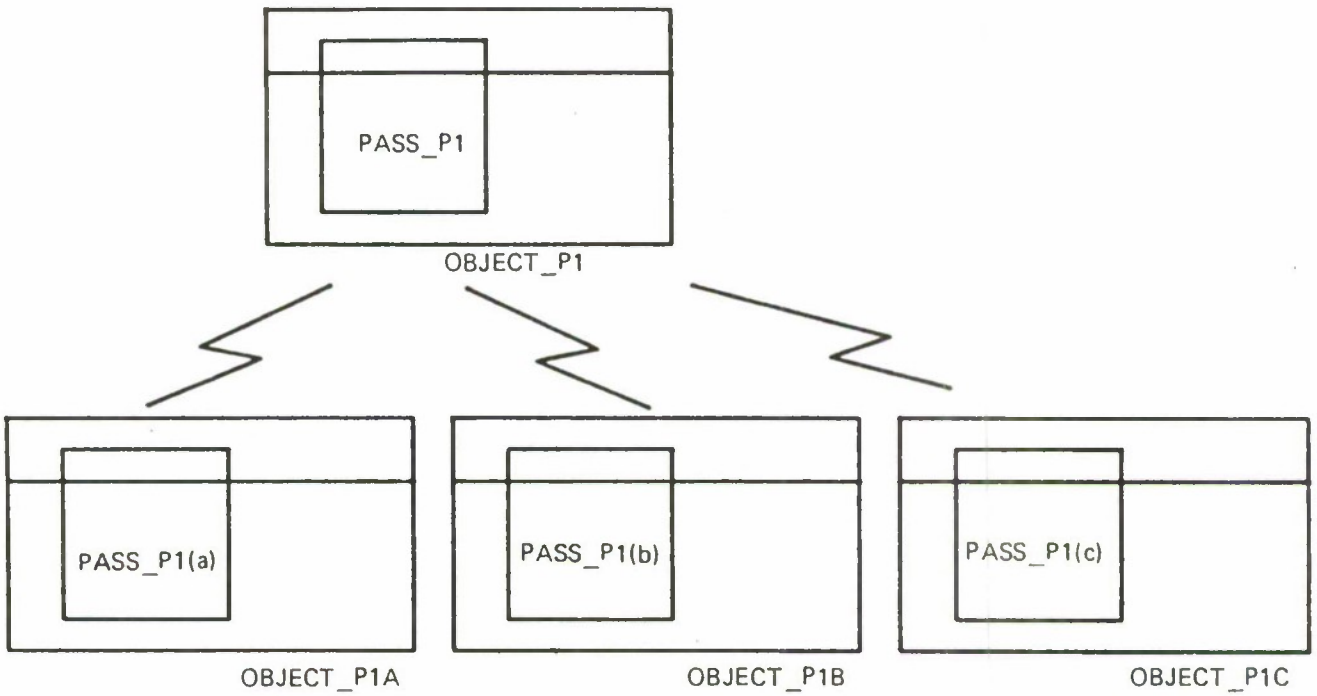
1) The collecting, processing and statistical display of experimental data, as commanded by work station called [illegible]

2) The check of environmental status and system processing hardware fitness on a periodic basic, where the results are reported to the operator console at a work station called WS

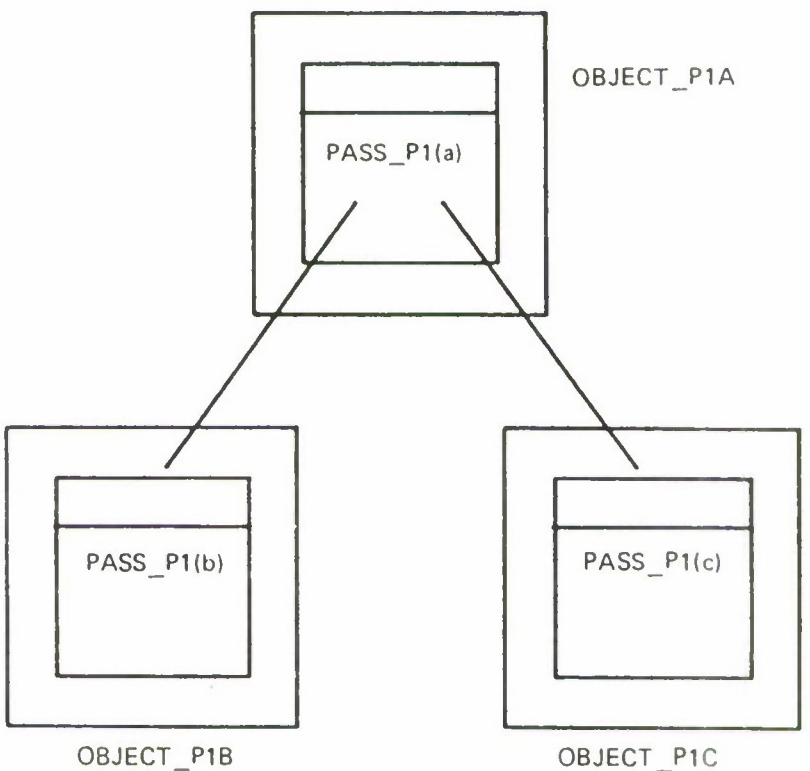
3) The orientation of solar panels as commanded by earth mission centers.

These processes are implemented by Ada tasks declared in the main subprogram. The SHARP diagram shown in Figure 67 graphically presents the results of this initial design step.

Ideally the processes are loosely coupled with little or no communication between the implementing tasks. This allows largely independent development of the system requirements assigned to the top level tasks. This goal is met in the design to follow in that the three processes do not interact.

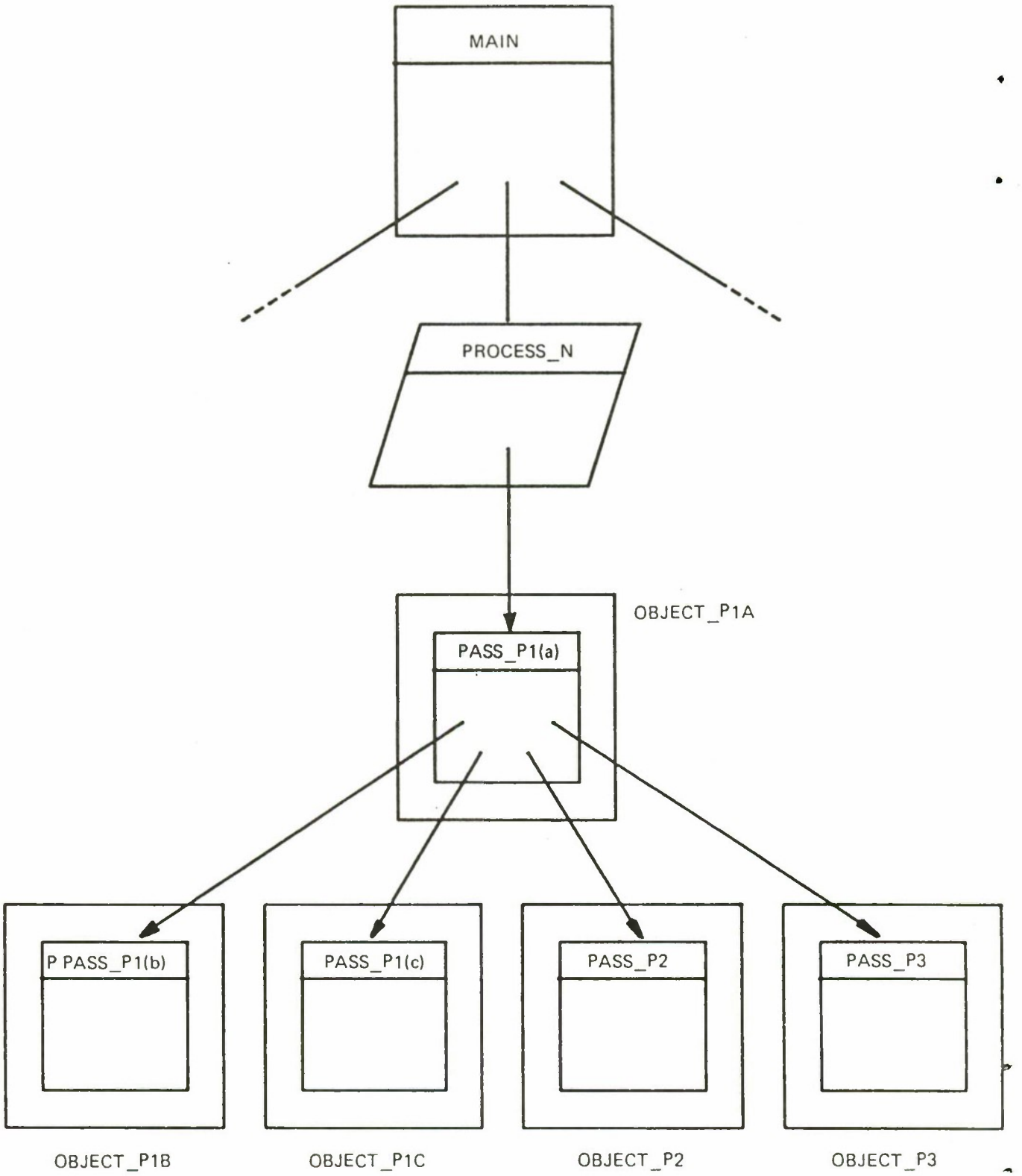


(a) PARTITIONING A LARGE OBJECT INTO A SET OF SMALLER OBJECTS



(b) INVOCATION DIAGRAM FOR THE SET OF SMALLER OBJECTS

FIGURE 66. STEP 5 – PARTITION LARGE OBJECTS INTO A SET OF SMALLER OBJECTS



(c) RESULTING NEW SET OF OBJECTS

FIGURE 66. (CONCLUDED)

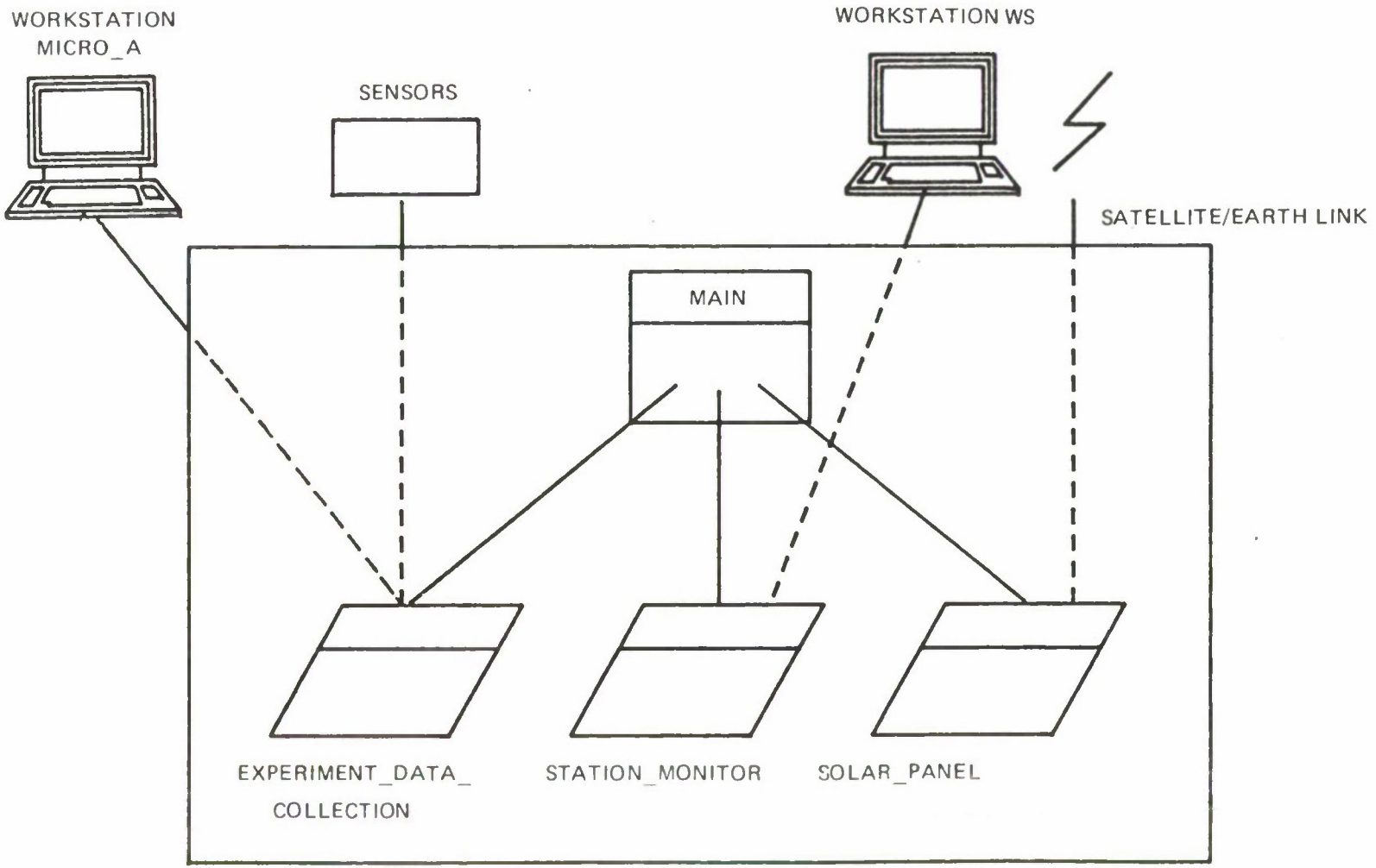


FIGURE 67. PROCESSES ESTABLISHED IN PROCEDURE MAIN (STEP 1)

3.3 ESTABLISHING OBJECTS FOR EACH PROCESS (STEP 2)

3.3.1 Objects for the Experimental Data Collection and Reduction Process

The 'Experiment Data Collection and Reduction' process collects data from three experiments, stores data samples in a data base and provides interactive access to the data base via work station MICRO_A. At the work station, the operator can view available data samples, command the display of statistical averages for a specific data sample, and command the display of a normal distribution or poisson distribution of the sample.

The objects needed to implement these requirements are presented in Figure 68. The objects are as follows:

a. **Experiment Data Collection**, which is implemented by program units contained within Package EXP_PL1_A. It contains three visible tasks that assemble data samples (one for each of the three different experiments) by receiving sensor readings during rendezvous with experiment hardware data ports. The readings are grouped in lists of designated sizes. Each list is referred to as a data sample.

b. **Data Base**, which is implemented by program units contained within Package EXP_PL2_A. It contains Task RECORD_DATA, which receives data samples from the Experiment Data Collection object and stores them by sample and sensor identification numbers. If local memory is full, the data samples for that experiment are archived in mass memory. Procedure GET_RECORD is provided to access data samples.

c. **Command Coordinator**, which is implemented by the package EXP_PL1_B. This object responds to input commands received from the work station at microprocessor MICRO_A. It accesses data via the Data Base object to retrieve a specific data sample for viewing, or subsequent statistical processing. It also can respond to user commands to list the identifiers of experiment data samples.

d. **Statistical Distribution**, which is implemented by the package EXP_PL2_B. It is used to establish a data sample's mean, standard deviation, and normal or poisson distribution. This object uses math functions contained in Package EXP_PL3_A. The results of its calculations are presented on the screen of work station MICRO_A.

3.3.2 Objects for the Station Monitor Process

The 'Station Monitor' process periodically takes power, temperature and pressure readings from space station sensors. A range of acceptable readings are either input by the operator or are automatically established (i.e., default values). If a reading is out of bounds, an alarm message is displayed at the work station. A record is made of sensor readings and alarm messages.

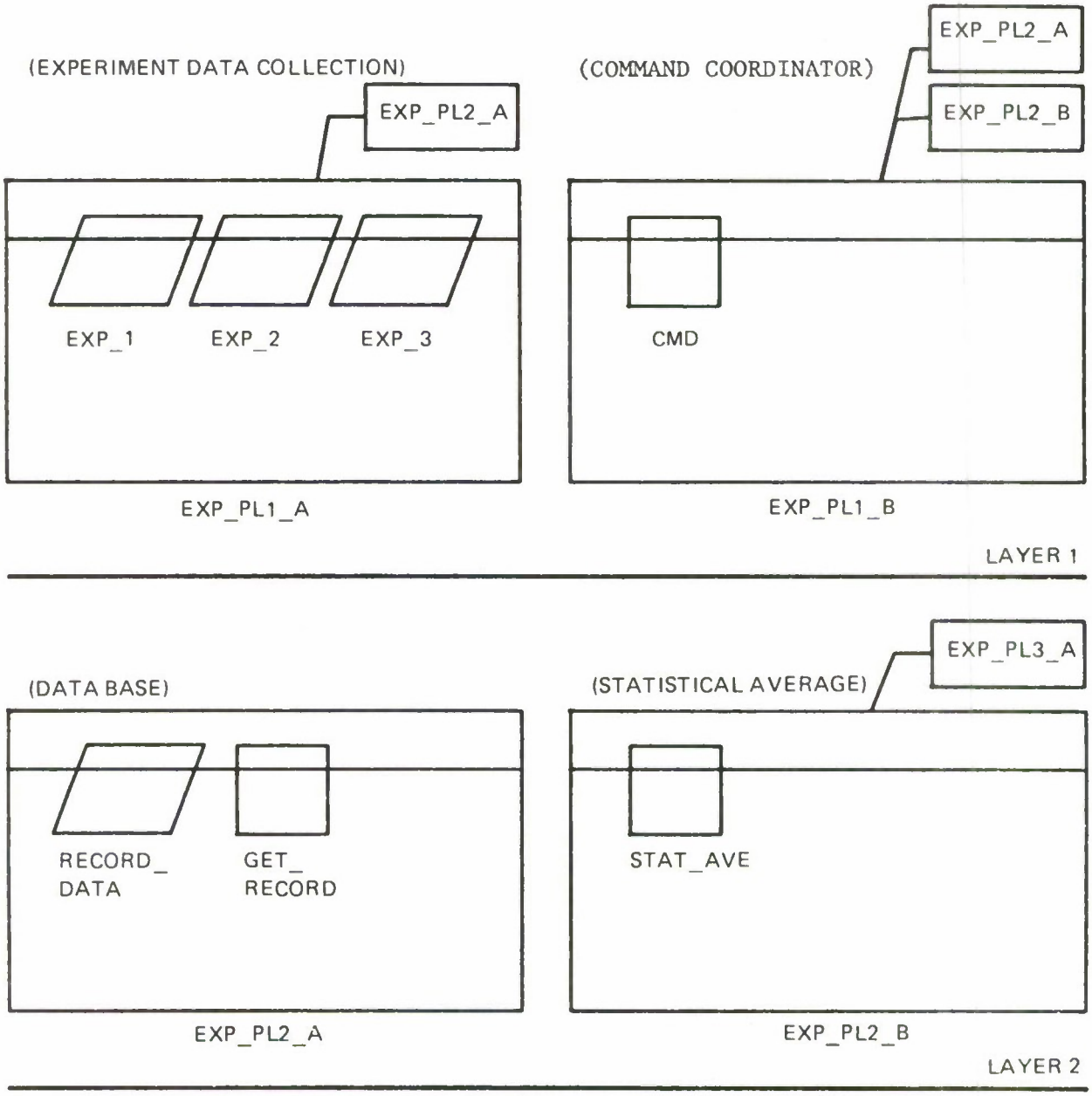


FIGURE 68. OBJECT IMPLEMENTATIONS IN PROCESS TASK "EXPERIMENT_DATA_COLLECTION" (STEP 2)

In addition, a built-in-test is periodically run to check the fitness of the selected system hardware. Messages indicating detected errors are displayed at the work station.

The objects needed to implement these requirements are represented in Figure 69. The objects are as follows:

a. **Environmental Sensors**, which is implemented by the package M_PL1_A. It contains three tasks that rendezvous with environmental sensor hardware to obtain values for power, temperature and pressure. The values are passed to the sensor processing package M_PL2_A.

b. **Sensor Processing**, which is implemented by the package M_PL2_A. It contains a task to receive and record sensor values. A check sensor procedure is invoked to determine if the sensor value is in an acceptable range. If a reading is out of bounds, program units in the alarm package are invoked, an alarm is generated at the work station WS and the alarm is recorded in mass storage.

c. **Alarm**, which is implemented by the package M_PL3_A. Program units within it are invoked when the Sensor Processing package detects an out of bound sensor condition. An alarm message is generated at the work station WS and the alarm is recorded in mass storage.

d. **Command Processor**, which is implemented by the package M_PL1_B. It is invoked by the Task STATION_MONITOR and accepts a user command made at Work Station WS to set a range of acceptable sensor values. Task CMD invokes the Procedure SET_LIMITS in the sensor processing package to update the acceptable sensor ranges.

e. **Built-In-Test**, which is implemented by the package M_PL1_C. It contains the Task BIT, which periodically checks the functioning of the central processor by executing a series of Ada instructions. Unexpected results are reported to the work station WS as processor error conditions.

3.3.3 Objects for the Solar Panel Orientation Process

THE 'Solar Panel Orientation' process controls the orientation of the solar panel. An earth/satellite communication link provides two-way data transmission between earth and the space station. The current solar panel orientation is transmitted to earth and the earth mission centers transmit directional data to the station to give the desired new orientation.

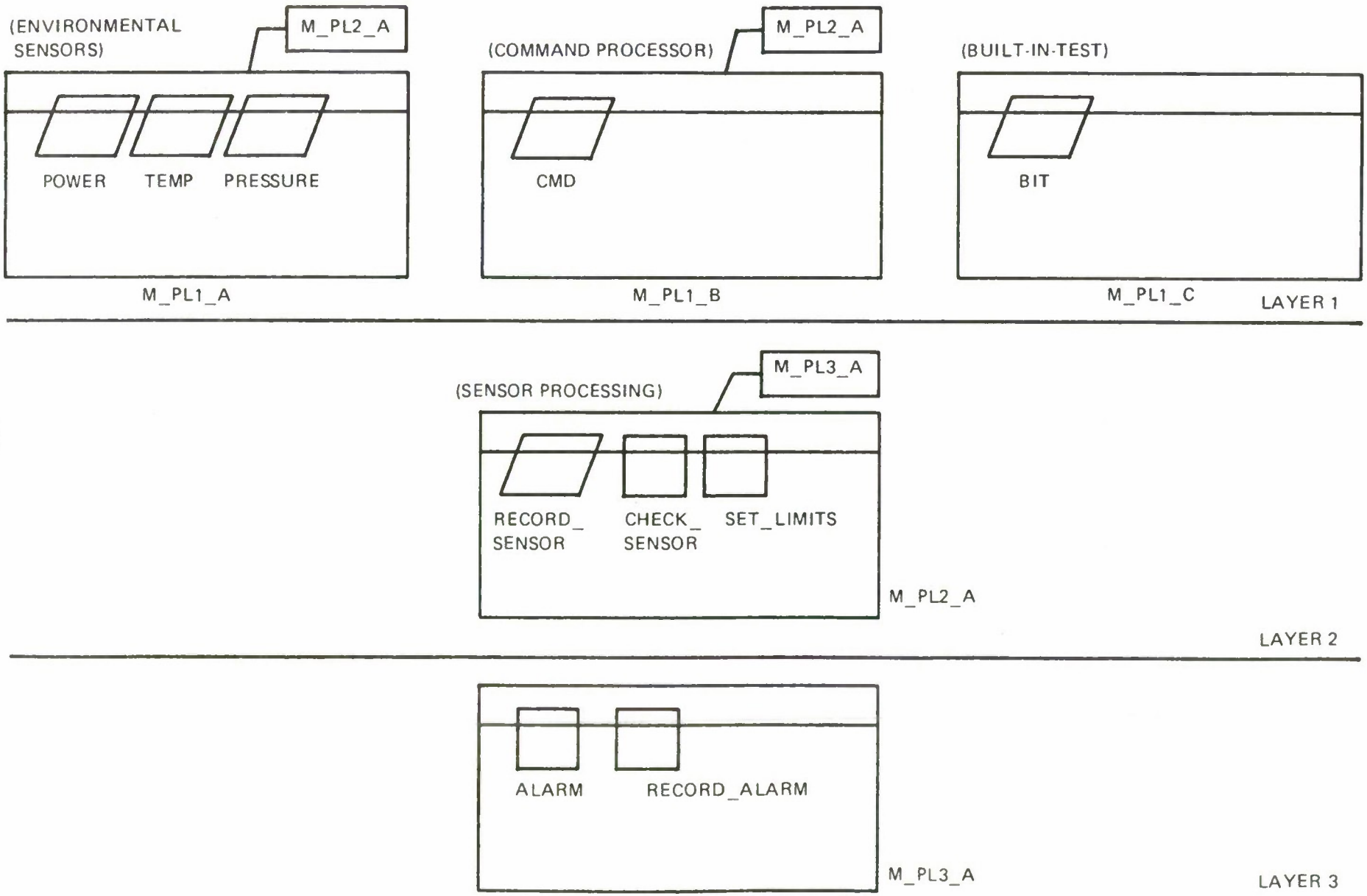


FIGURE 69. OBJECT IMPLEMENTATIONS IN PROCESS TASK "STATION_MONITOR" (STEP 2)

The objects needed to implement these requirements are presented in Figure 70. The objects are as follows:

a. **Update Orientation**, which is implemented by the package SP_PL1_A. It receives directional data via rendezvous with radio data link hardware. A new orientation is calculated using the math package SP_PL2_A and is passed to the panel arm motion package that uses the old and new orientation to calculate and implement motions to orient the solar panel. The new orientation is then sent to the solar panel task for transmission to earth.

b. **Math Package**, which is implemented by the package SP_PL2_A. It contains mathematical functions needed for calculating the new solar panel orientation.

c. **Panel Arm Motion**, which is implemented by the package SP_PL2_B. It uses the old and new orientation to calculate solar panel arm motions needed to reorient the solar panel. The panel mount motors package is invoked to generate the required motions.

d. **Panel Motors**, which is implemented by the package SP_PL3_B. Procedures MOTOR1 and MOTOR2, visible procedures within this package are invoked by Procedure PANEL_ARM_DELTAS of the panel arm motion package. In this way, the required panel motions are obtained using two controlling motors. A library package MOTOR_IO encapsulates access to low level control functionally needed for operating the hardware.

3.4 ESTABLISHING INTERFACES BETWEEN OBJECTS (STEP 3)

This section establishes the interfaces between objects of the Experiment Data Collection and Reduction Process. An invocation diagram for these object implementations process is presented in Figure 71. The calling dependencies of the visible procedures in the object packages are represented along with task interactions.

In Figure 72, a SHARP task rendezvous diagram shows data flow between hardware and the 'Experiment Data Collection' object implementation, and data flow between the later and the 'Data Base' object implementation. Figure 73 shows data flow between the 'Command Coordinator' object implementation and both the 'Data Base' and 'Statistical Distribution' object implementation. The parameter passing is used to assemble the experiment data base, access that data base, and establish statistical distributions for data selected by an operator. Parameters passed account for operator commands and the results of statistical calculations. Variables and flags used to implement the operations unique to each object implementation (e.g., the statistical calculations) are not passed. Because of this, the object implementations are decoupled.

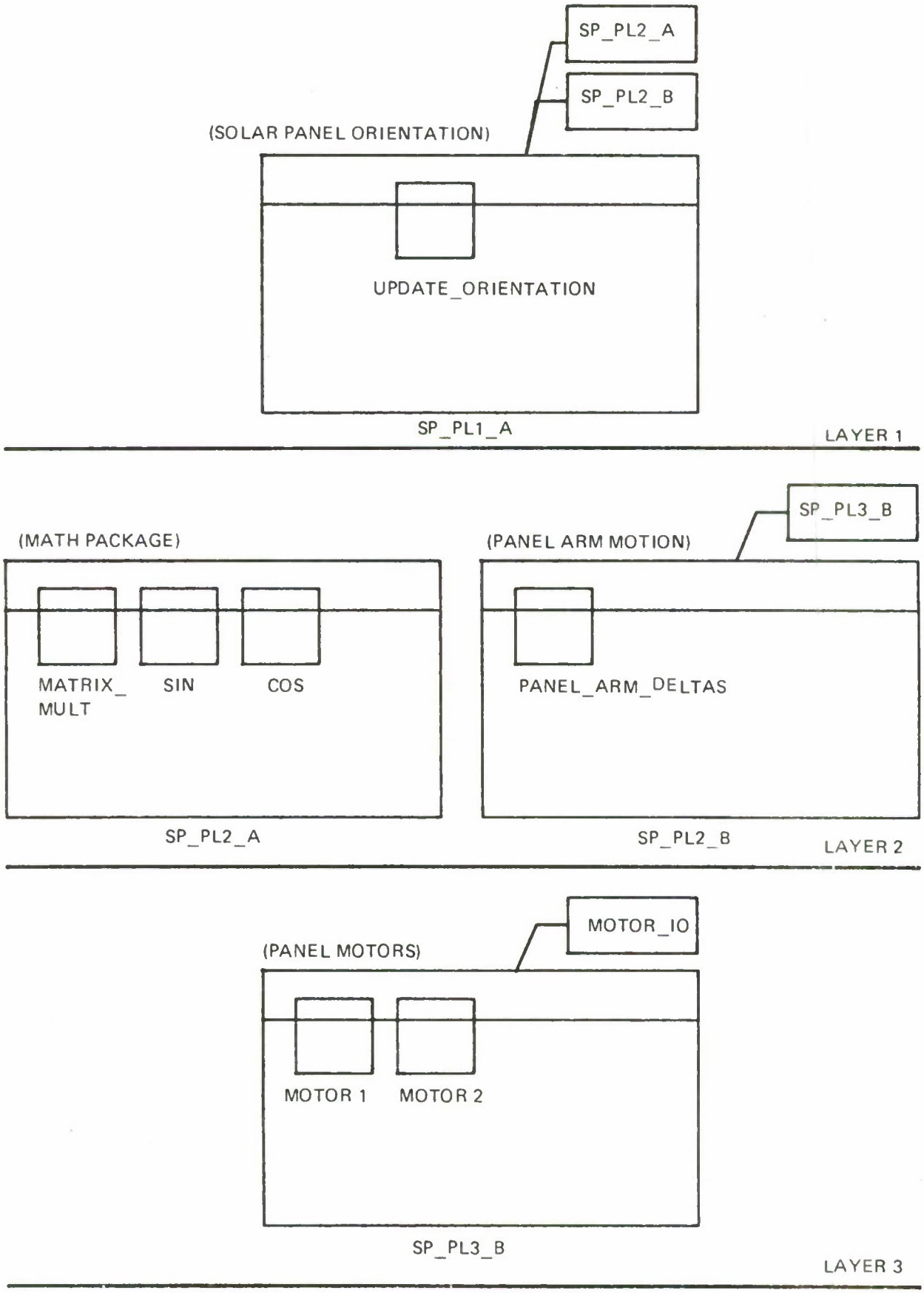
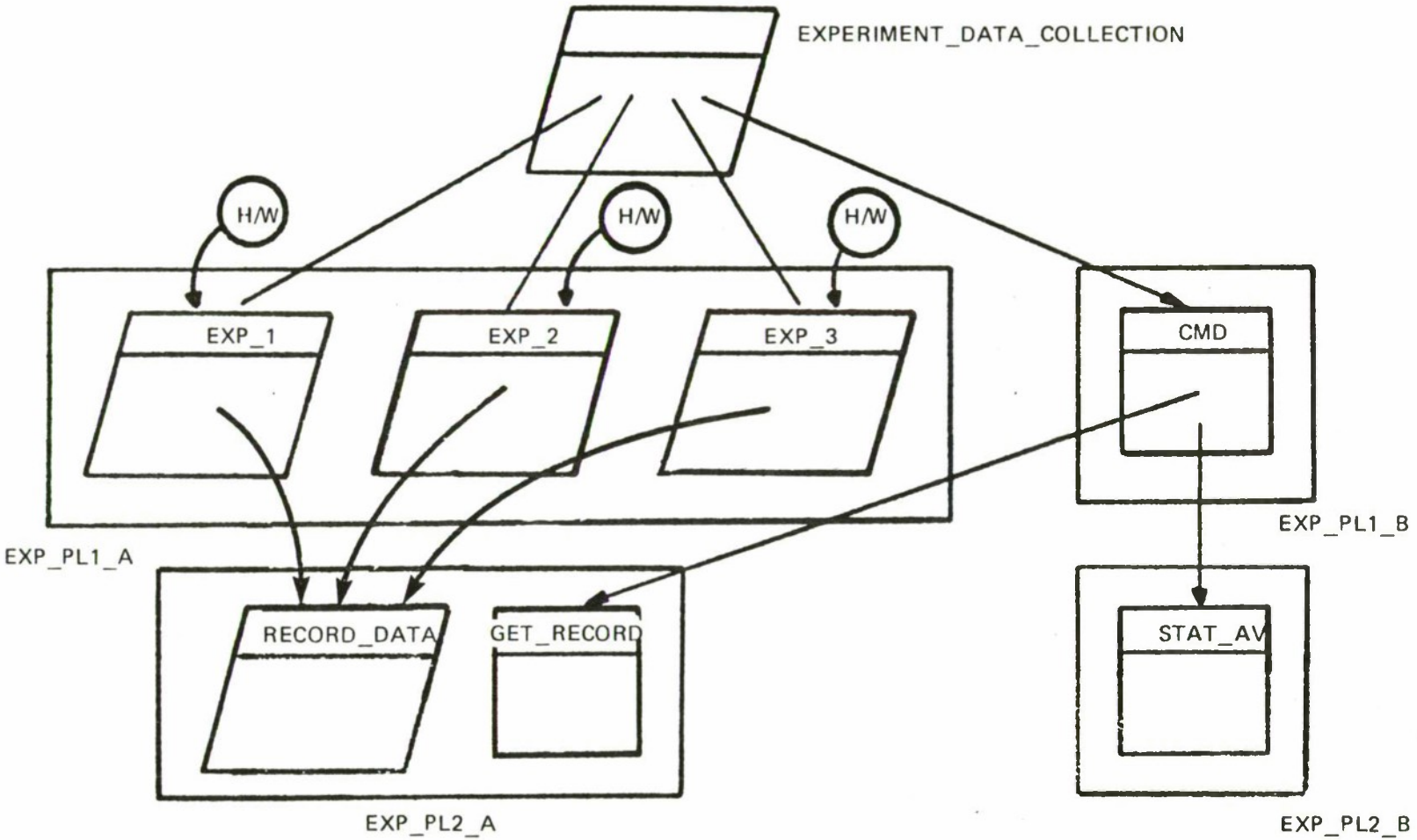


FIGURE 70. OBJECT IMPLEMENTATIONS IN PROCESS TASK "SOLAR_PANEL" (STEP 2)



(a) OBJECT INTERACTION

FIGURE 71. INTERFACE OF OBJECT IMPLEMENTATIONS WITHIN PROCESS 'EXPERIMENT_DATA_COLLECTION' (STEP 4)

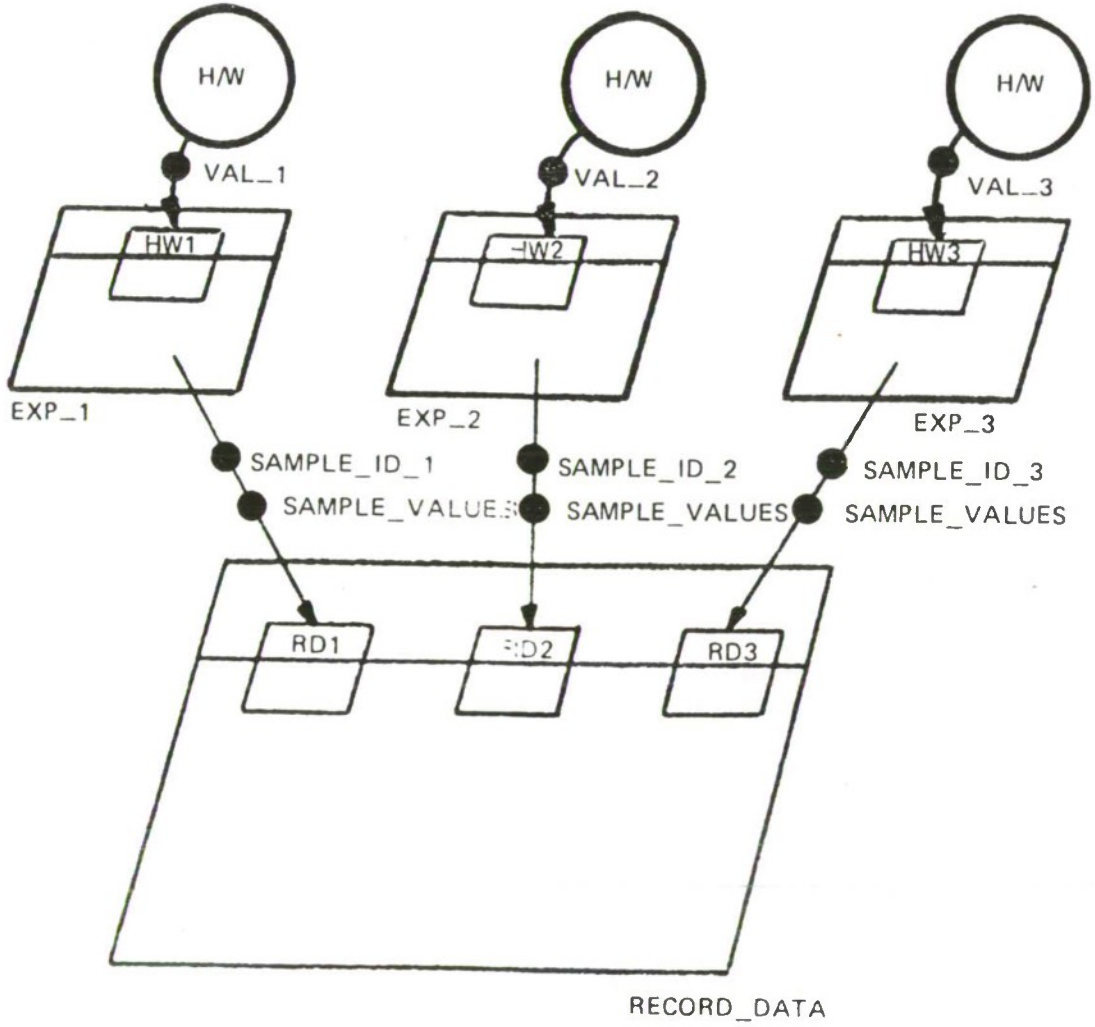


FIGURE 72. RENDEZVOUS BETWEEN OBJECT COMMUNICATING TASKS (STEP 3)

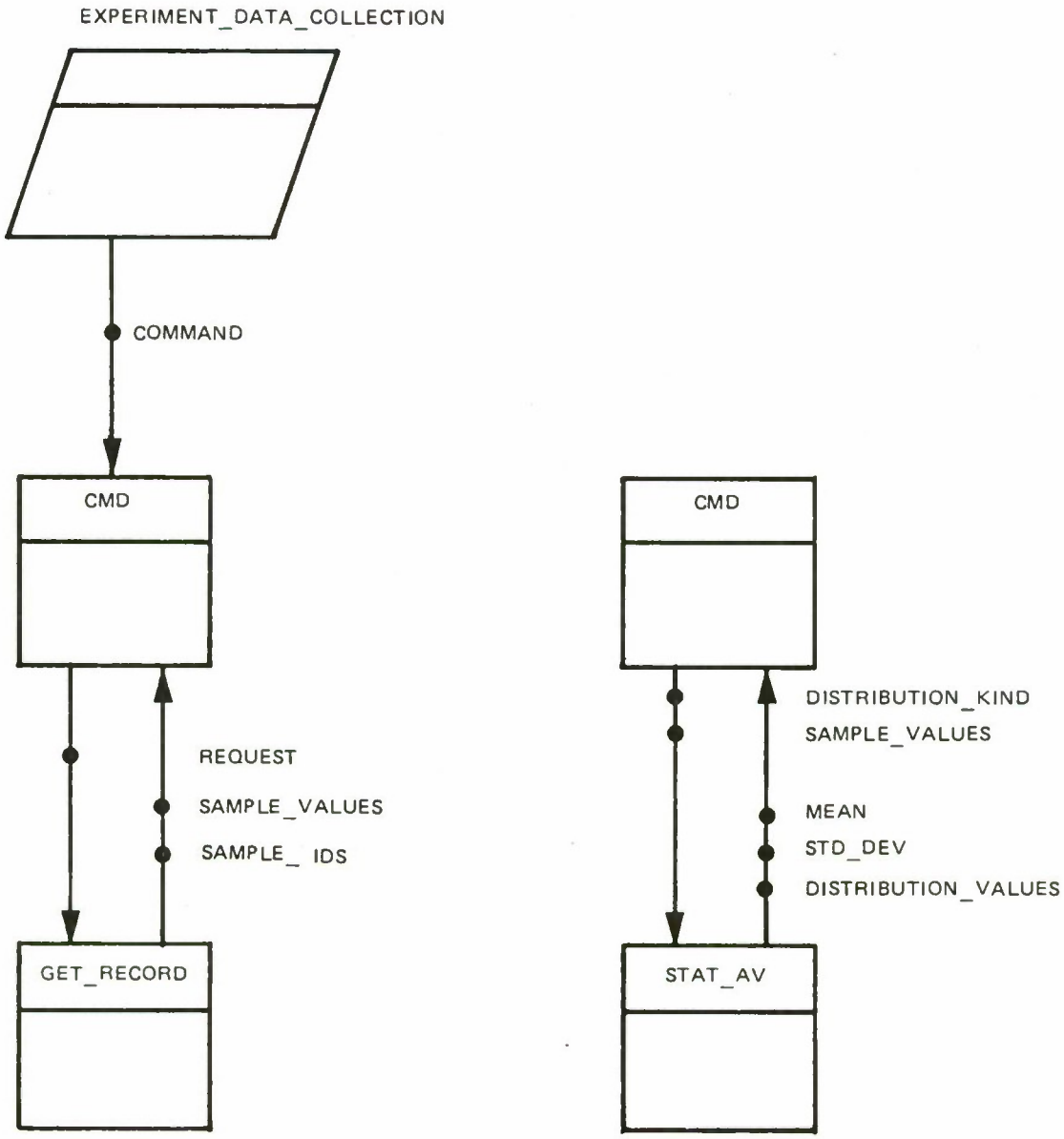


FIGURE 73. DATA FLOW BETWEEN OBJECT COMMUNICATING SUBPROGRAMS (STEP 3)

3.5 COMMENCE DESIGN OF OBJECT IMPLEMENTATIONS (STEP 4)

3.5.1 Experiment Data Collection Object Implementation (Package EXP_PL1_A)

The 'Experiment Data Collection' object, which is implemented by a Package EXP_PL1_A, contains three visible communicating tasks that rendezvous with experiment hardware and build lists of experimental values, which are referred to as samples. For a communicating task the hierarchy diagram is shown in Item a of Figure 74, a data flow diagram is presented in Item b, and annotated pseudo code is presented in Item c. These SHARP abstracts commence the internal design of the 'Experiment Data Collection' object implementation. With them, an initial version of the object's data structure can be established as shown in Item d.

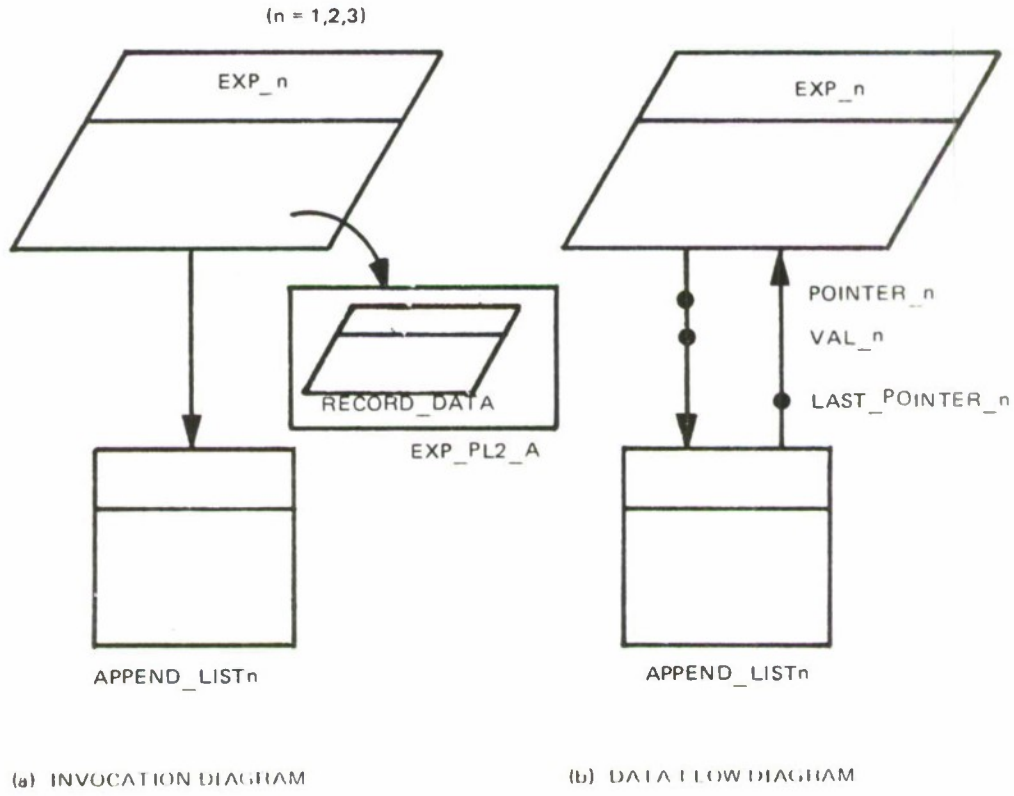
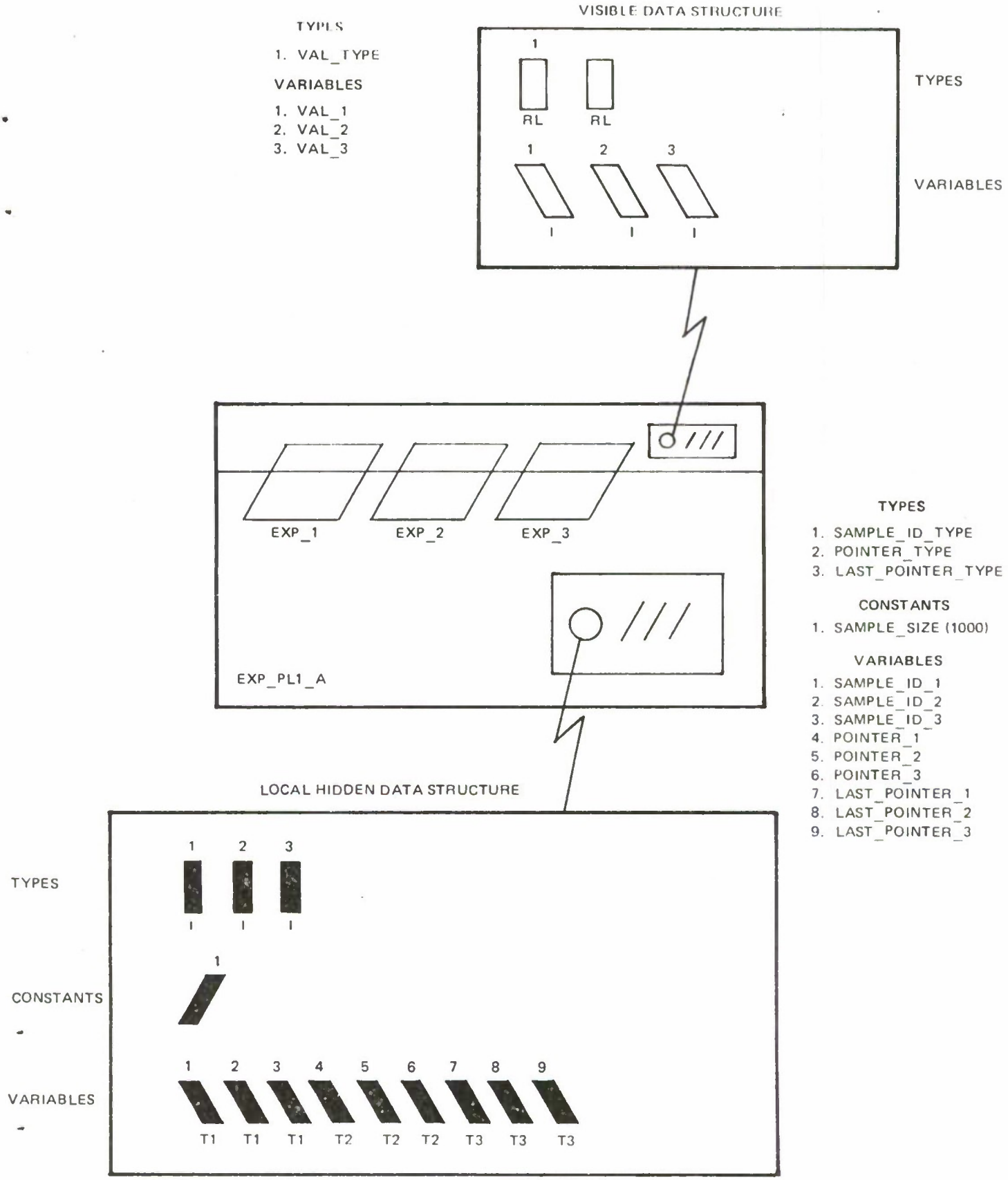


FIGURE 74. INTERNAL DESIGN OF THE 'EXPERIMENT DATA COLLECTION' OBJECT (STEP 4)

```
Begin pseudo code for Task EXP_n

initialize variables including SAMPLE_ID_n =1

loop

     for i = 1 to SAMPLE_SIZE

     rendezvous with sensor to receive
     current sensor reading VAL_n


     CALL APPEND_LISTn to enter sensor reading into a linked list

     end for
     We now have a list consisting of 'SAMPLE_SIZE'
     sensor readings, which shall be referred to as a sample.

Call Entry Point RDI of Task RECORD_DATA -- Package EXP_PL2-A
     to pass the sample to the data base object
     implementation along with its identifi-
     cation number

SAMPLE_ID_n = SAMPLE_ID_n+1

end loop

End pseudo code for Task EXP_n
```

(c) PSEUDO CODE FOR THE BODY OF TASK 'EXP_1'

FIGURE 74. (CONTINUED)



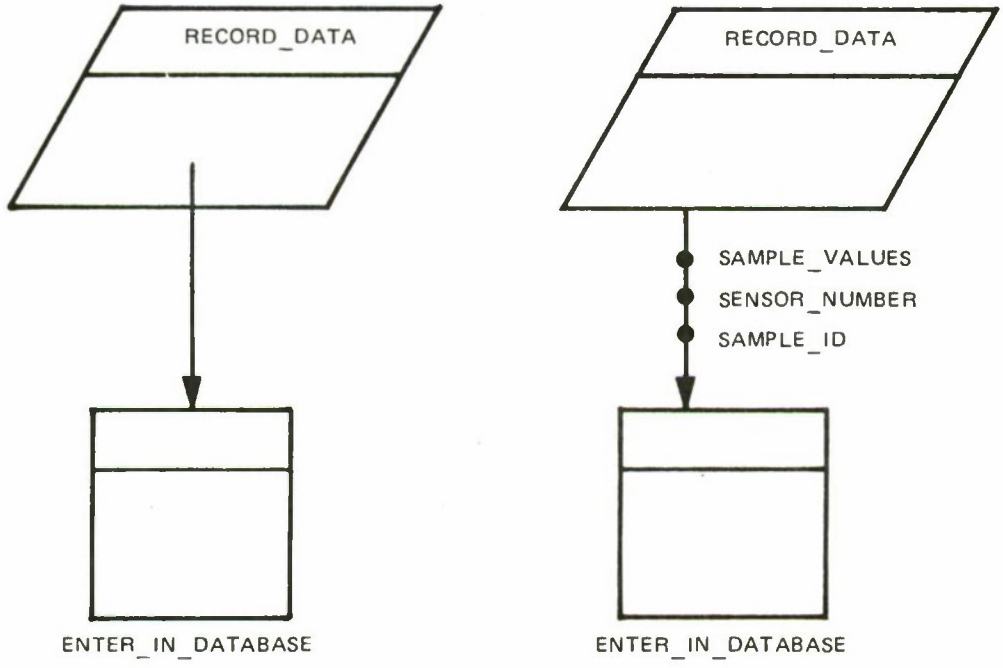
(d) DATA STRUCTURE DIAGRAM

FIGURE 74. (CONCLUDED)

3.5.2 Data Base Object Implementation (Package EXP_PL2_A)

The 'Data Base' object, which is implemented by Package EXP_PL2_A, contains two visible program units. The first, Task RECORD_DATA, establishes a data base containing the experimental data. The data is stored by an experiment identifier and by a data sample identifier in local memory. The second visible procedure in Package EXP_PL2_A, Procedure GET_RECORD, retrieves specific data samples for a specific experiment from the data base. It searches the data base to find a given experiment and specific samples for that experiment.

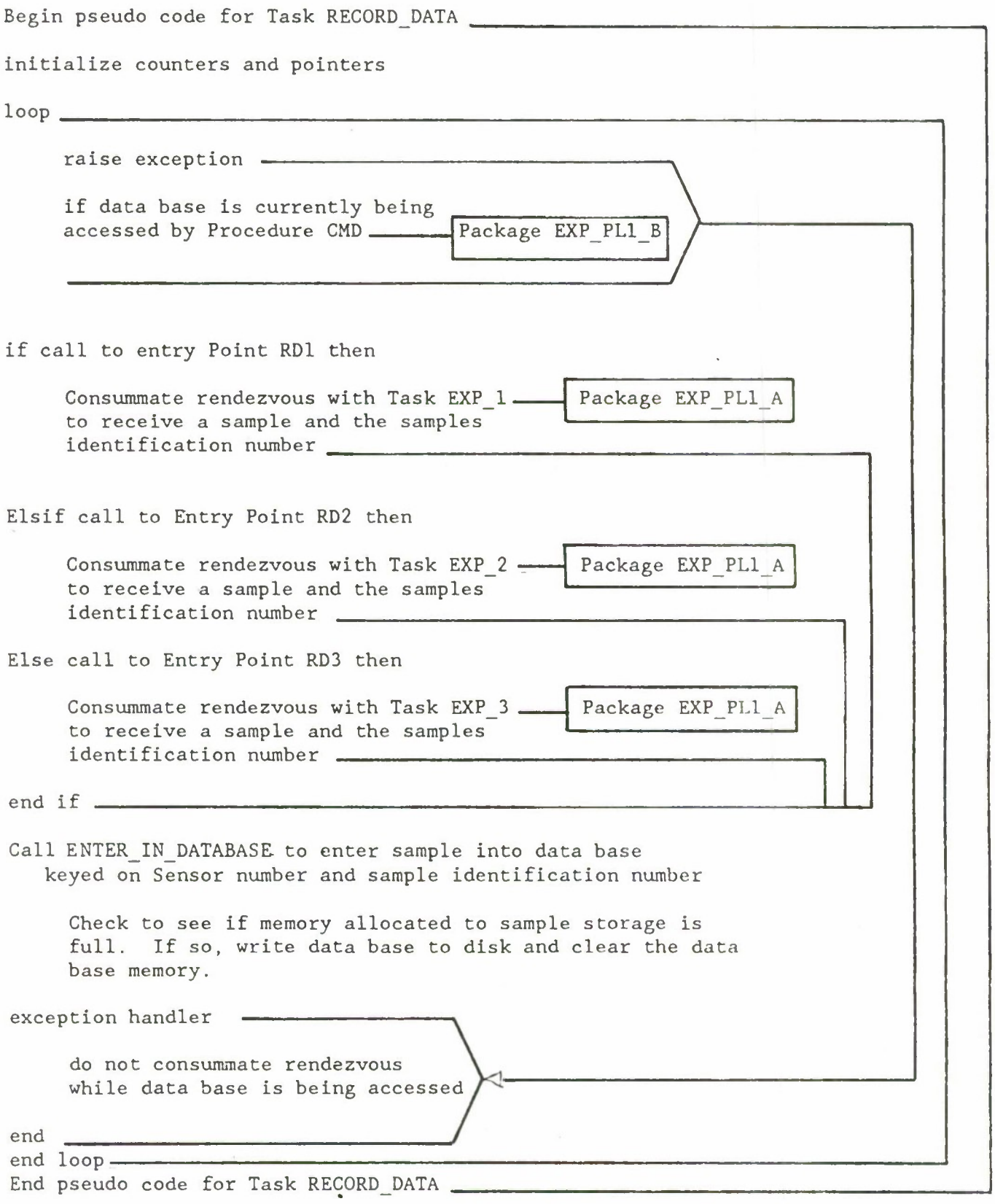
For the communicating program unit 'Task RECORD_DATA', a hierarchy diagrams is shown in Item a of Figure 75, a data flow diagram is presented in Item c. Annotated pseudo code for communicating task RECORD_DATA is shown in Item c, and Procedure GET_RECORD' is shown in Item d. These abstracts commence the internal design of the 'Data Base' object implementation. With them, an initial version of the object's data structure can be established as shown in Item e.



(a) INVOCATION DIAGRAM

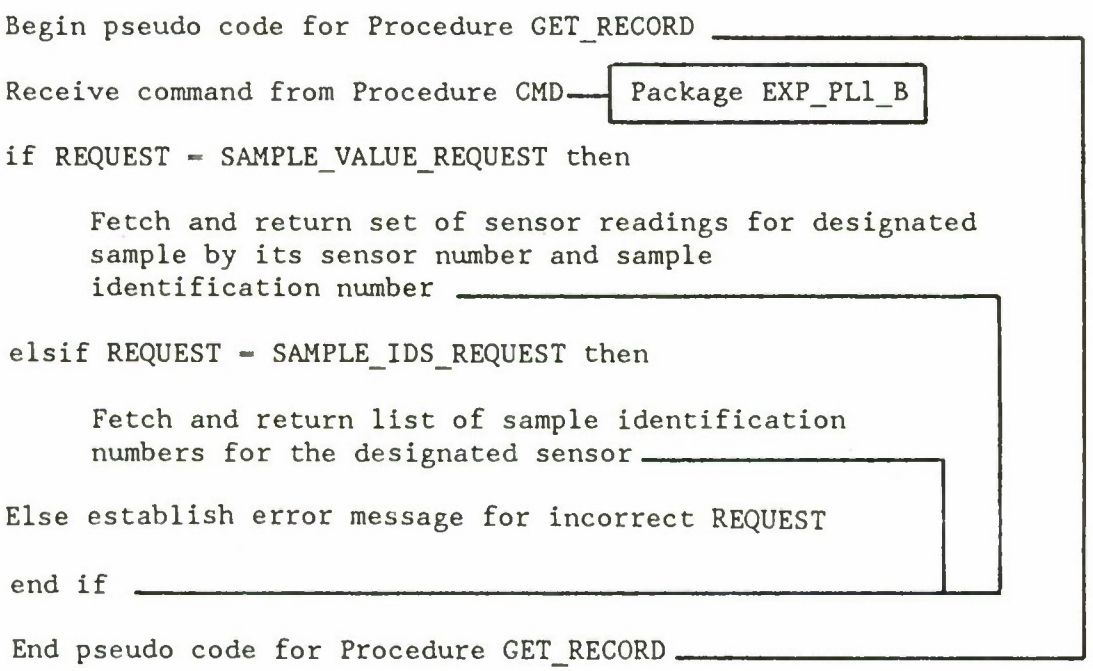
(b) DATA FLOW DIAGRAM

FIGURE 75. INTERNAL DESIGN OF THE 'DATA BASE' OBJECT (STEP 4)



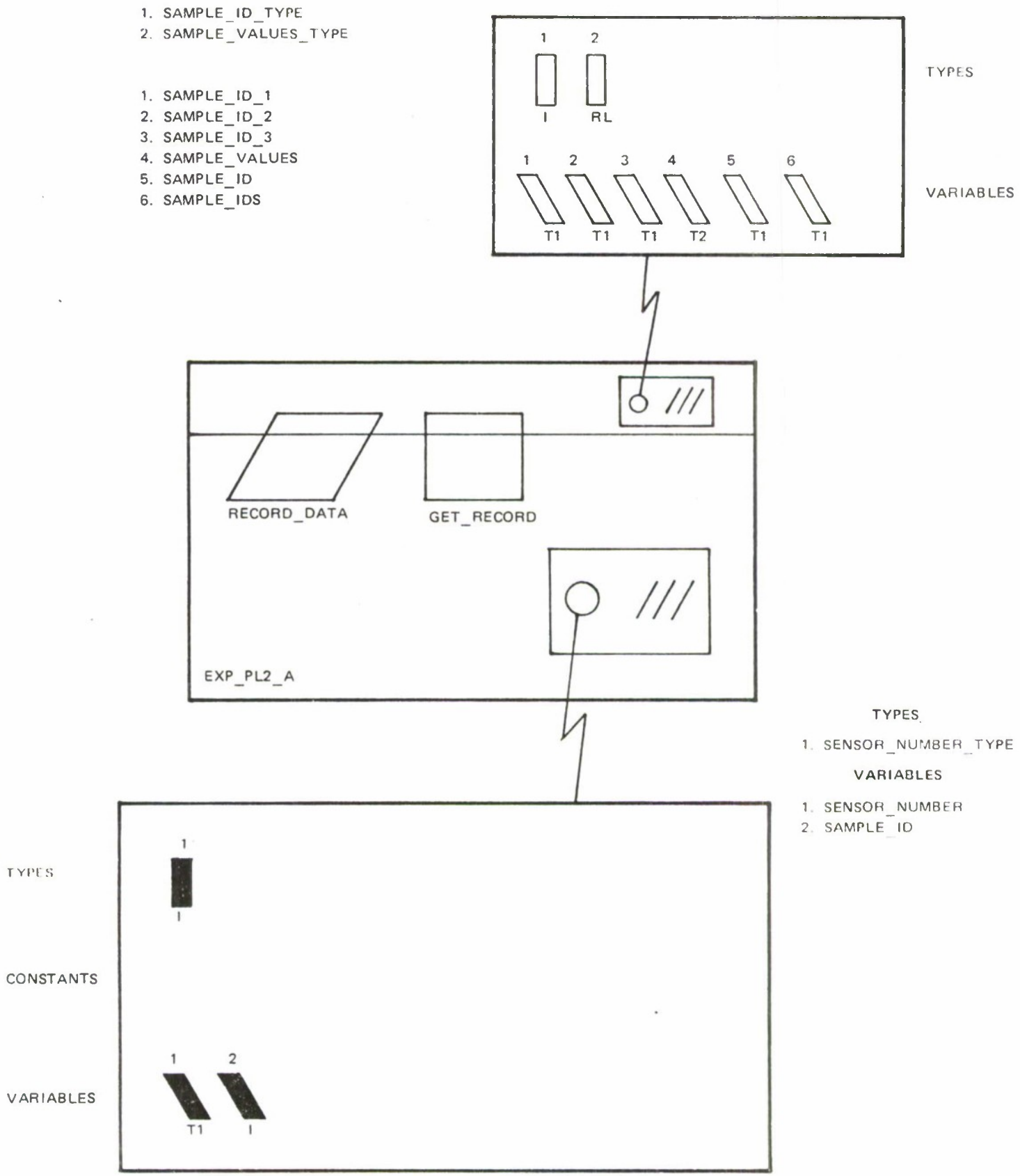
(c) PSEUDO CODE FOR THE BODY OF TASK 'RECORD_DATA'

FIGURE 75. (CONTINUED)



(d) PSUEDO CODE FOR THE BODY OF PROCEDURE 'GET_RECORD'

FIGURE 75. (CONTINUED)



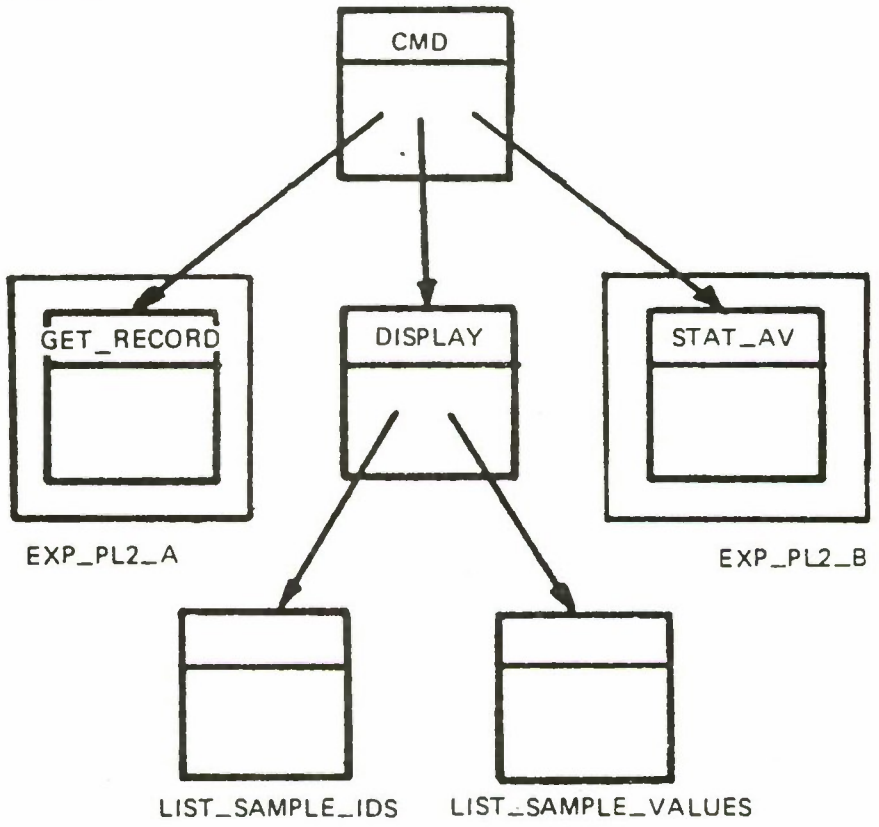
(e) DATA STRUCTURE DIAGRAM

FIGURE 75. (CONCLUDED)

3.5.3 Command Coordinator Object Implementation (Package EXP_PL2_A)

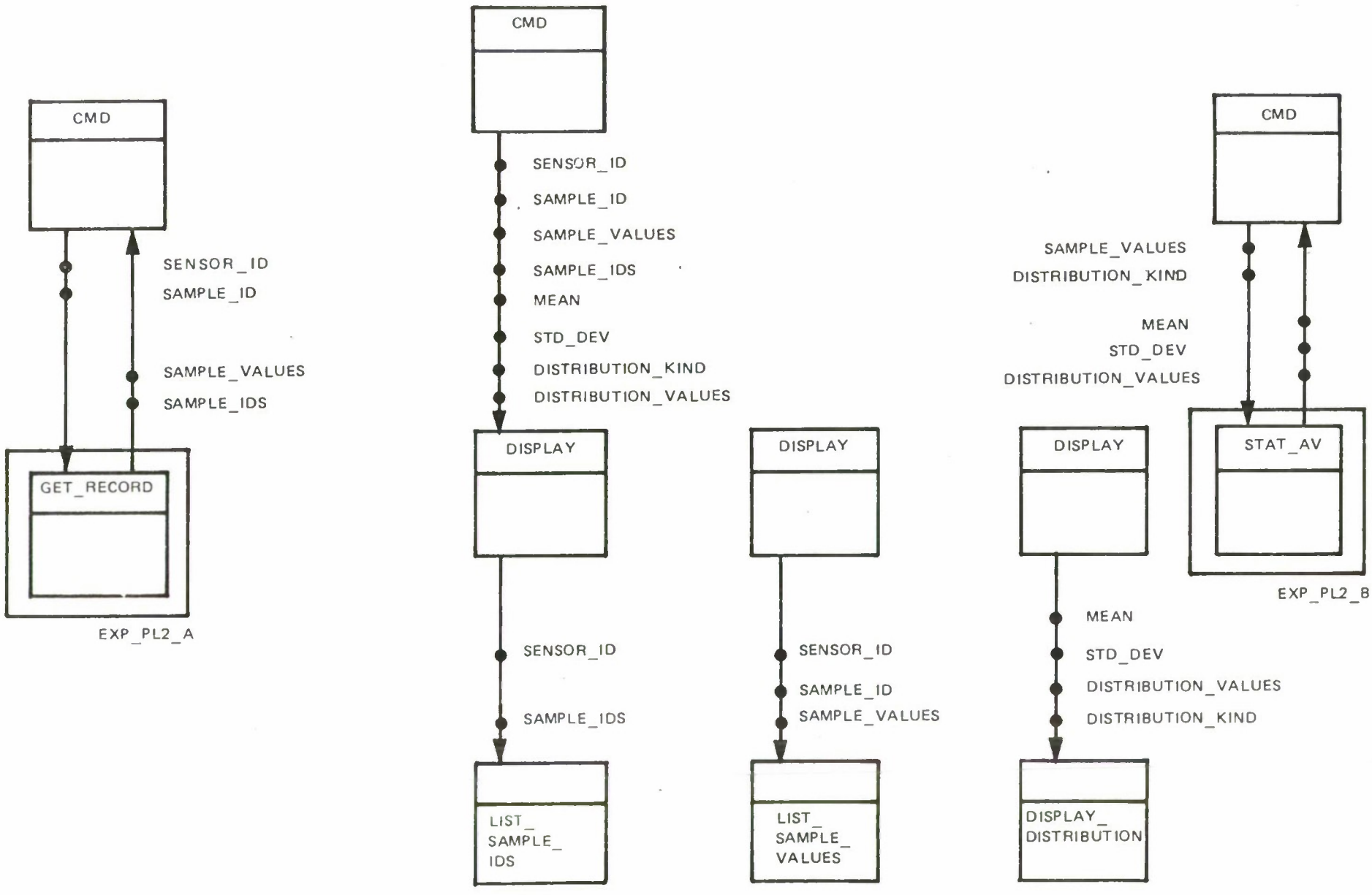
The 'Command Coordinator' object, which is implemented by Package EXP_PL2_A, contains the visible Procedure CMD. It receives and interprets instructions from the system operator via the top level process task (i.e., Task EXPERIMENT_DATA_COLLECTION). It accesses Procedure GET_RECORD of the Data Base object to obtain an experimental sample from the data base. It calls procedures of the 'Statistical Distribution' object to establish a requested distribution for a data sample retrieved from the data base. Procedure CMD can also call Procedure DISPLAY to establish for display at the work station MICRO_A: (a) a list of sample identifiers for a particular experiment or (b) the data values for a particular sample of a particular experiment.

For the communicating program unit 'Procedure CMD,' a hierarchy diagram is shown in Item a of Figure 76, a data flow diagram is presented in Item b, and annotated pseudo code is presented in Item c. These abstracts commence the internal design of the Command Coordinator Object implementation. With them, an initial version if the object's data structure can be established as shown in Item d.



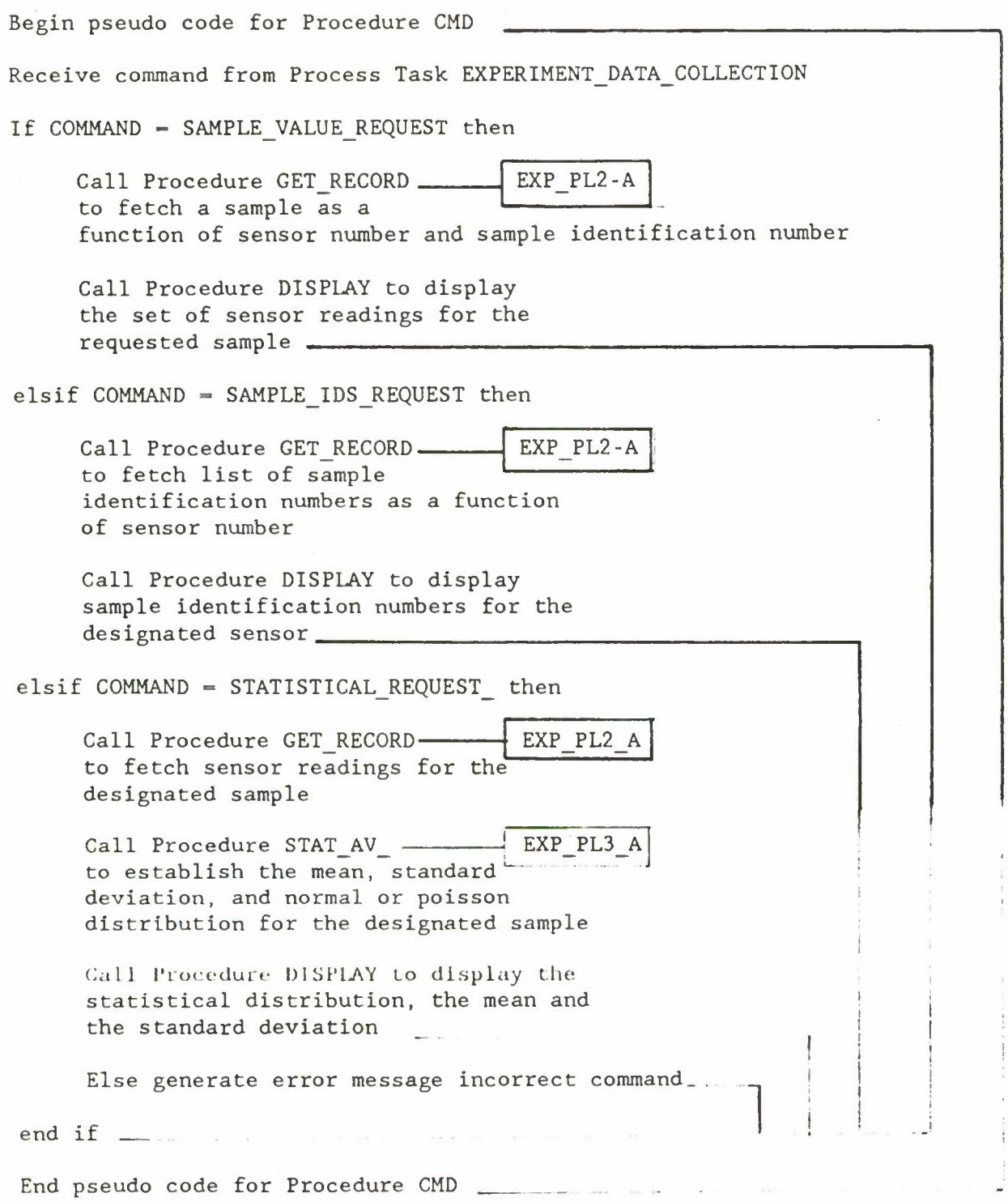
(a) INVOCATION DIAGRAM

FIGURE 76. INTERNAL DESIGN OF THE 'COMMAND COORDINATOR' OBJECT (STEP 4)



(b) DATA FLOW DIAGRAMS

FIGURE 76. (CONTINUED)



(c) PSEUDO CODE FOR THE BODY OF PROCEDURE 'CMD'

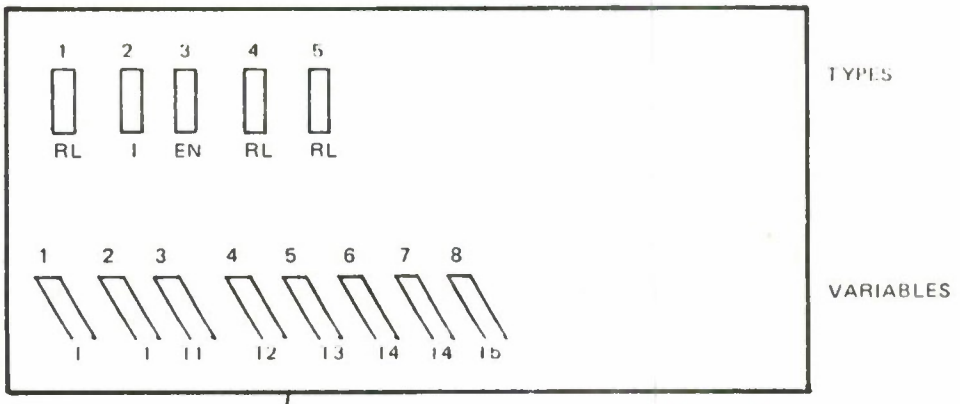
FIGURE 76. (CONTINUED)

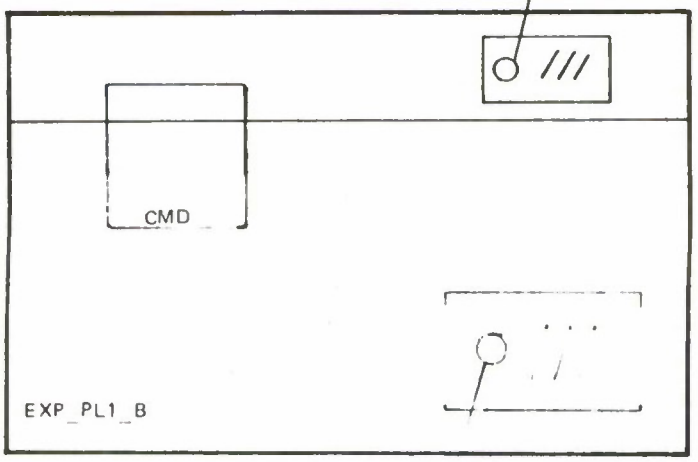
TYPES

1. SAMPLE_VALUES_TYPE
2. SAMPLE_ID_TYPE
3. DISTRIBUTION_KIND_TYPE
4. STAT_FACTOR_TYPE
5. DISTRIBUTION_VALUES_TYPE

VARIABLES

1. SENSOR_ID
2. SAMPLE_ID
3. SAMPLE_VALUES
4. SAMPLE_IDS
5. DISTRIBUTION_KIND
6. MEAN
7. STD_DEV
8. DISTRIBUTION_VALUES





TYPES

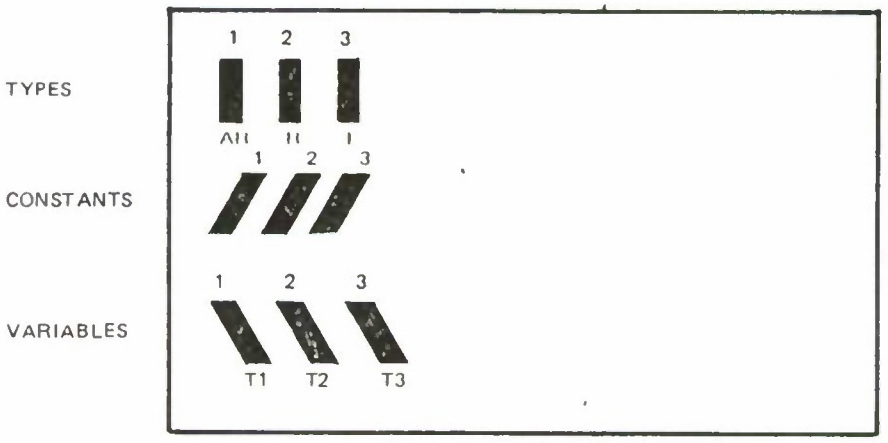
1. DISPLAY_FACTORS_TYPE
2. DISPLAY_PARAMETERS_TYPE
3. DISPLAY_RESOLUTION_TYPE

CONSTANTS

1. DISPLAY_CONST1
2. DISPLAY_CONST2
3. DISPLAY_CONST3

VARIABLES

1. DISPLAY_FACTORS
2. DISPLAY_PARAMETERS
3. DISPLAY_RESOLUTION



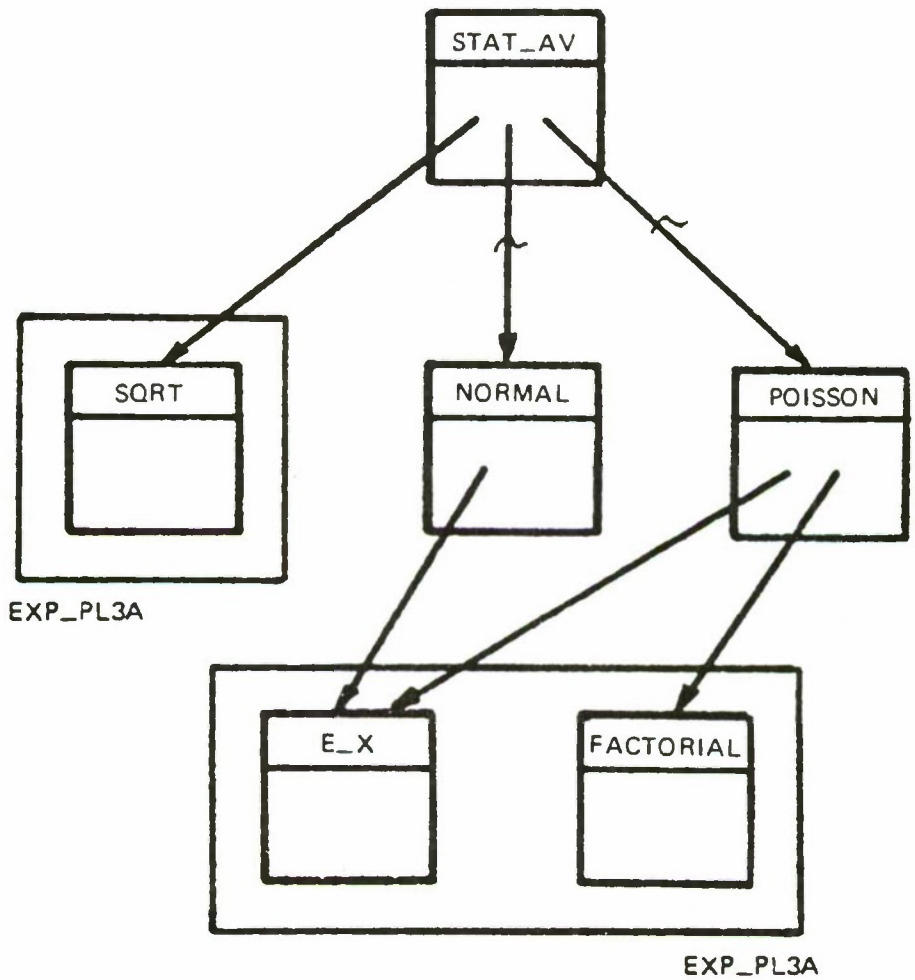
(d) DATA STRUCTURE DIAGRAM

FIGURE 76. (CONCLUDED)

3.5.4 Statistical Distributions Object Implementation (Package EXP_PL2_B)

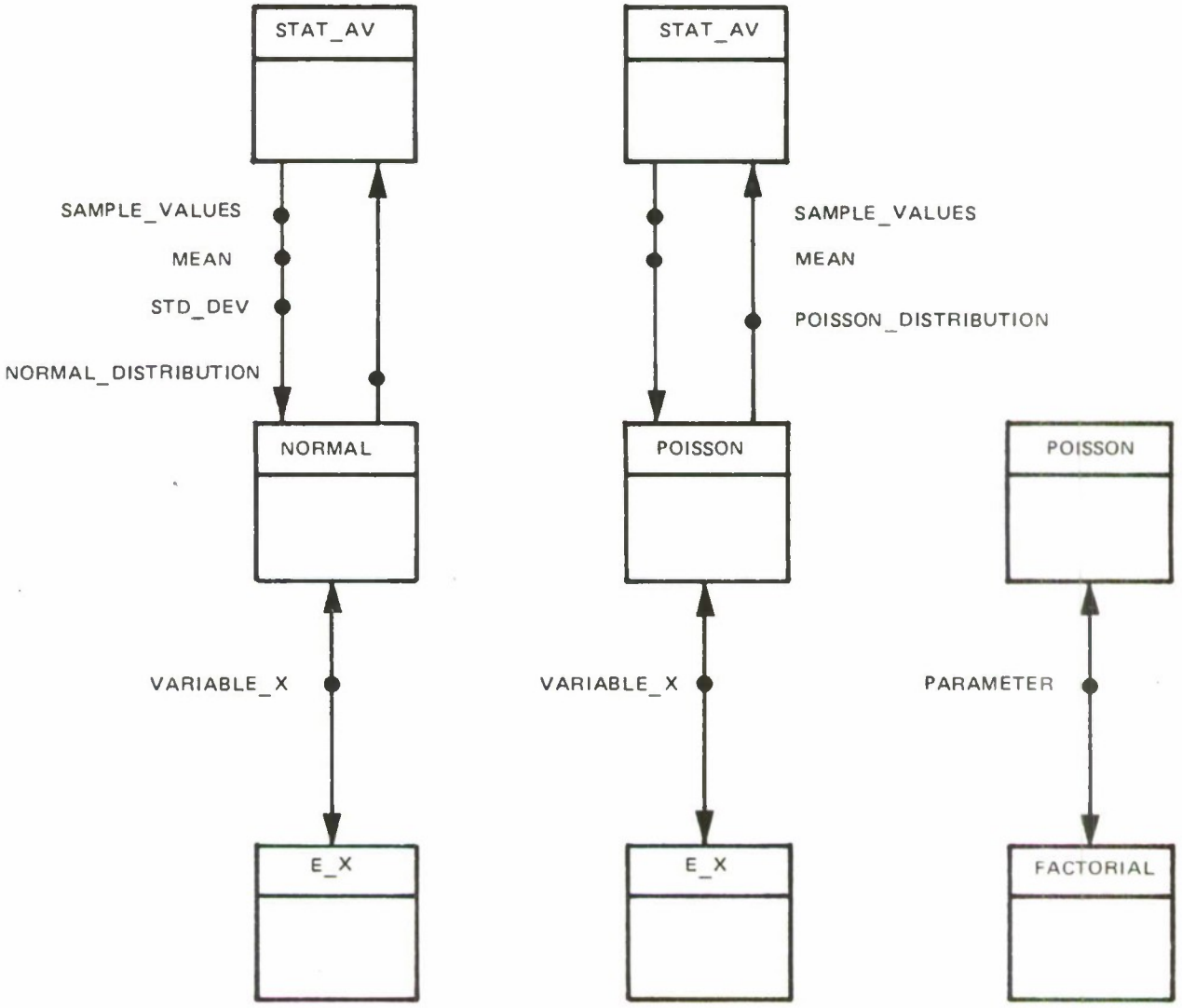
The 'Statistical Distributions' object, which is implemented by Package EXP_PL2_B, contains a visible procedure STAT_AV. It calculates the average and standard deviation of a received data sample. These results are then used to calculate a sample's normal or poisson distribution. This package accesses mathematical functions of a math package.

For the communicating program unit 'Procedure STAT_AV,' a hierarchy diagram is shown in Item a of Figure 77, a data flow diagram is presented in Item b, and annotated pseudo code is presented in Item c. These abstracts commence the internal design of the Statistical Distributions Object implementation. With them, an initial version of the object's data structure can be established as shown in Item d.



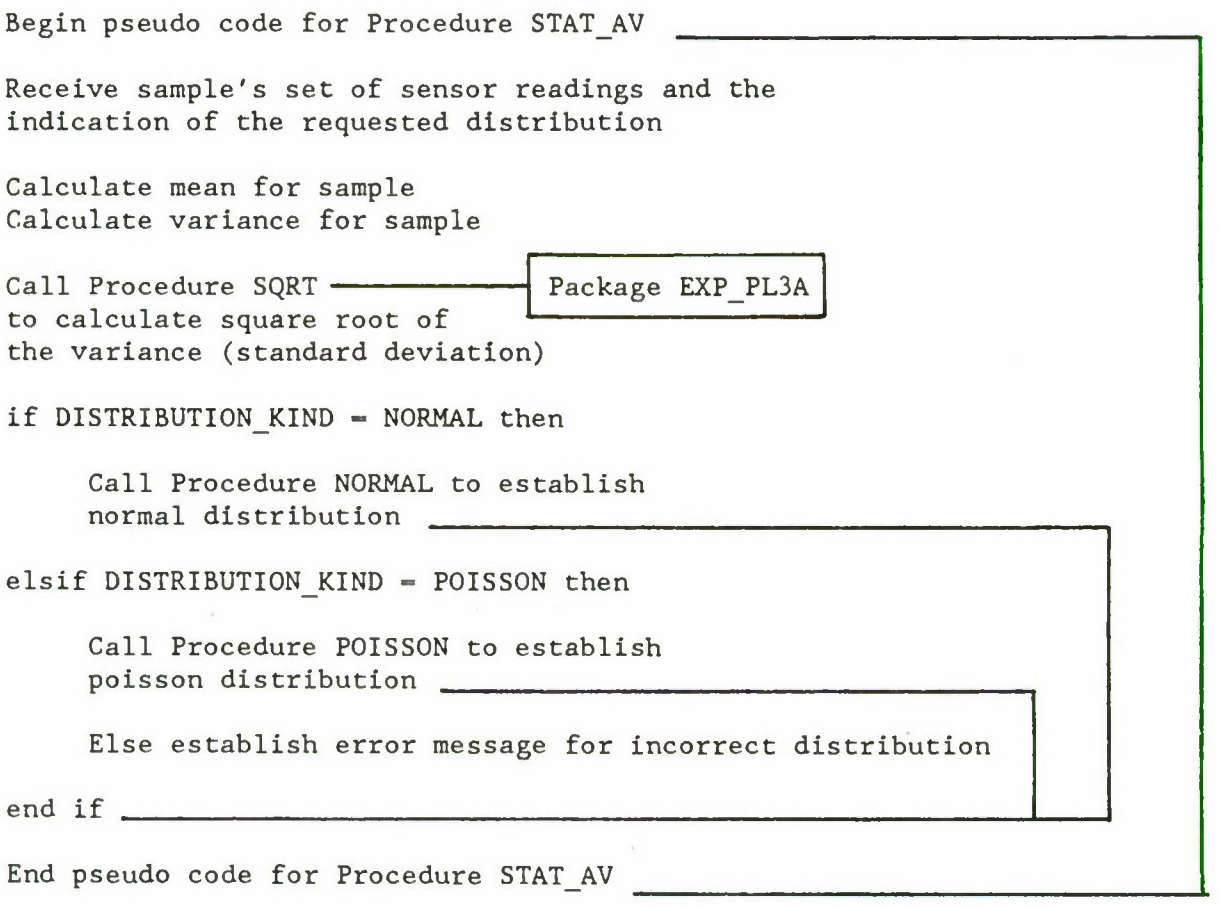
(a) INVOCATION DIAGRAM

FIGURE 77. INTERNAL DESIGN OF THE 'STATISTICAL DISTRIBUTION' OBJECT (STEP 4)



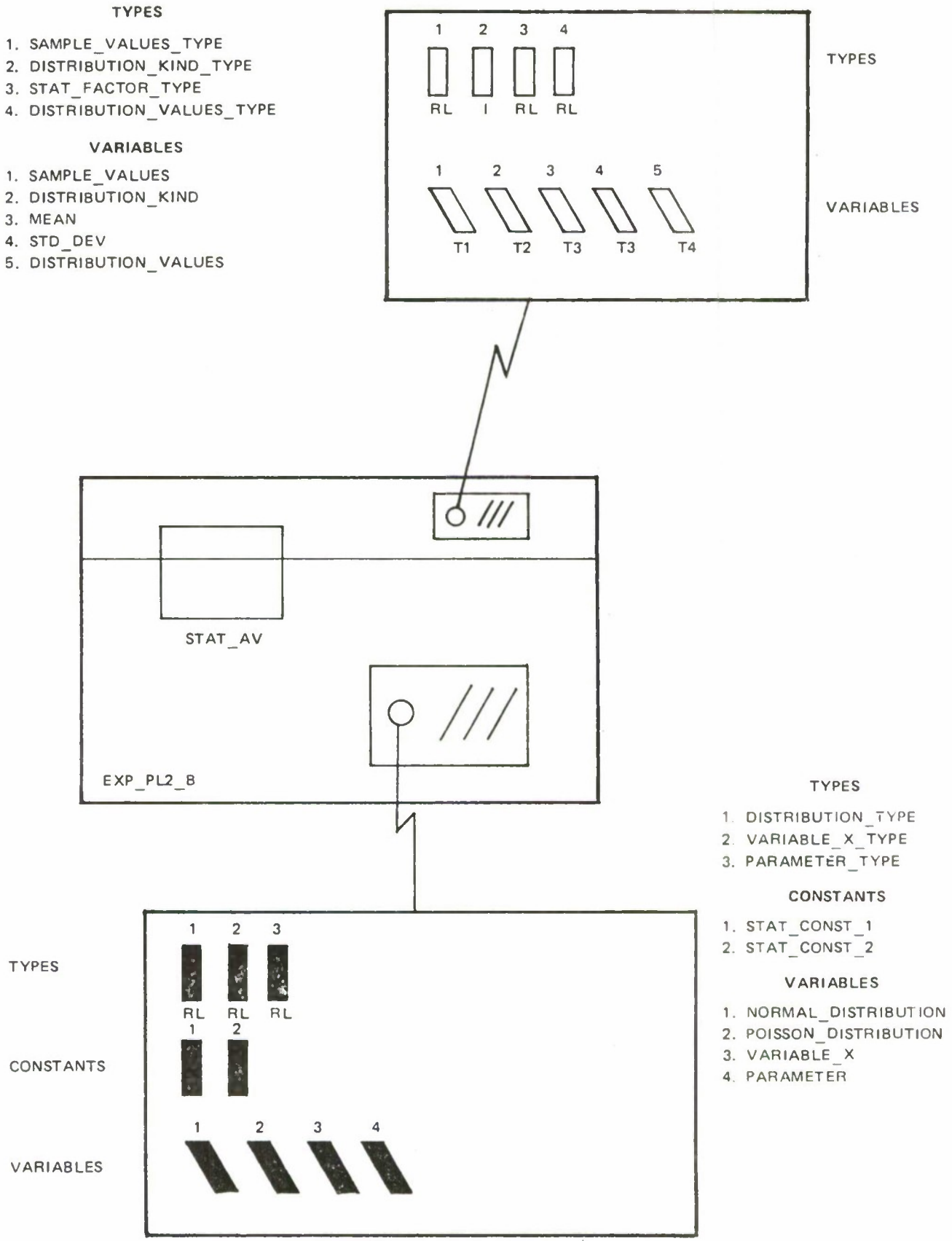
(b) DATA FLOW DIAGRAMS

FIGURE 77. (CONTINUED)



(c) PSEUDO CODE FOR THE BODY OF PROCEDURE 'STAT_AV'

FIGURE 77. (CONTINUED)



(d) DATA STRUCTURE DIAGRAM

FIGURE 77. (CONCLUDED)

4 CHAPTER SUMMARY

SHARP establishes a set of pictorial abstracts that can be used to represent an object-oriented design for an Ada computer program. With such a design, we can compose a large and complex computer program with manageable pieces. More importantly, we can localize design complexity, reducing interdependence relationships and thereby facilitating cost effective software development and maintenance.

At the highest level of the design, SHARP abstracts represent Ada tasks declared in the main procedure to establish the program's concurrent processing threads (processes). At the next level, the abstracts represent Ada packages and tasks introduced to encapsulate objects that implement each thread's processing requirements. The interaction of the object implementations is represented by SHARP Invocation Diagrams. The complex bodies of program units responsible for communication between object implementations, visible within a package encapsulating an object, can be represented by SHARP Hierarchy and Invocation Diagrams. Abstracts envisioned as "blow ups" of entities identified in an invocation diagram can be used to represent details of task rendezvous, data flow between program units and data structures. The later diagram shows information hiding within data structures local to objects. At the lowest level of SHARP, annotated pseudo code is used to represent operations and logic within the bodies of individual program units.

Reviewing the design of a large computer program (i.e., 200,000 lines or more) is a massive, time consuming and potentially error prone process. An Ada computer program of that magnitude represented exclusively by comprehensive PDL would provide a very large document, perhaps "several feet thick," which would be very difficult to comprehend. SHARP provides abstracts that selectively present levels of design detail that would enable the development team of systems engineers and software engineers to communicate among themselves and with government reviewers, at different levels of abstraction. Abstraction is essential in the management of complexity. SHARP abstracts allow concise communication of relevant detail in the form of pictographs that capture the essential program structure. The annotated pseudo code option gives remaining detail to those personnel who need it.

CHAPTER V

SHARP IN DoD SOFTWARE DESIGN DOCUMENTS

This chapter describes the use of SHARP within software design documents to represent the structure of an Ada computer program to be implemented in an object-oriented manner. Specifically, graphics are described for a Software Top Level Design Document prepared in accordance with DI-MCCR-80012 and a Software Detailed Design Document prepared in accordance with DI-MCCR-80031. These documents are associated with the development of software per Department of Defense (DoD) requirements specified in DOD-STD-2167.

DOD-STD-2167, entitled "Defense System Software Development," is the military standard fundamental to defense system software acquisition. It defines phases of a computer program's life cycle, reviews of computer program development held by the government during the life cycle, and products that must be delivered by a contractor to document the computer program and its development.

Computer software developed in accordance with DOD-STD 2167 is assigned to one or more Computer Software Configuration Items (CSCIs) (formerly referred to as Computer Program Configuration Items or CPCIs). Each CSCI is developed over the six phases shown in Figure 78.

As shown in Figure 79, in addition to source and object code, twenty-four principle products are developed to document software plans, requirements, design, test consideration and manuals. These products must adhere to requirements specified in the specific Data Item Descriptions (DIDs) indentified in the figure. The products provide documentation needed by the government (a) to verify that the software adheres to contractual requirements, and (b) to maintain the software after delivery by the contractor.

1 INTRODUCTION

1.1 BACKGROUND

In 1977, the Joint Logistics Commanders formed the Computer Software Management Subgroup. In 1979, they decided at the Monterey 1 Software Workshop that tri-service software standards and data item descriptions (DIDs) were needed to standardize the DoD software development process among the three services. This objective was met on June 4, 1985 when the DoD released the following set of software development standards (SDS):

- DOD-STD-2167, Defense System Software Development
- MIL-STD-483A, Configuration Management Practices for Systems, Equipment, Munitions and Computer Programs
- MIL-STD-490A, Specification Practices

SSR = Software Specification Review
PDR = Preliminary Design Review
CDR = Critical Design Review
TRR = Test Readiness Review
FCA = Functional Configuration Audit
PCA = Physical Configuration Audit

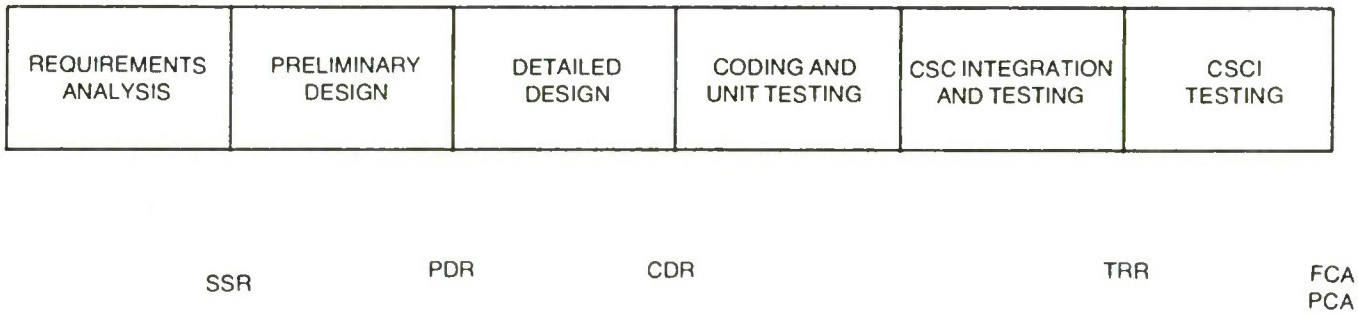
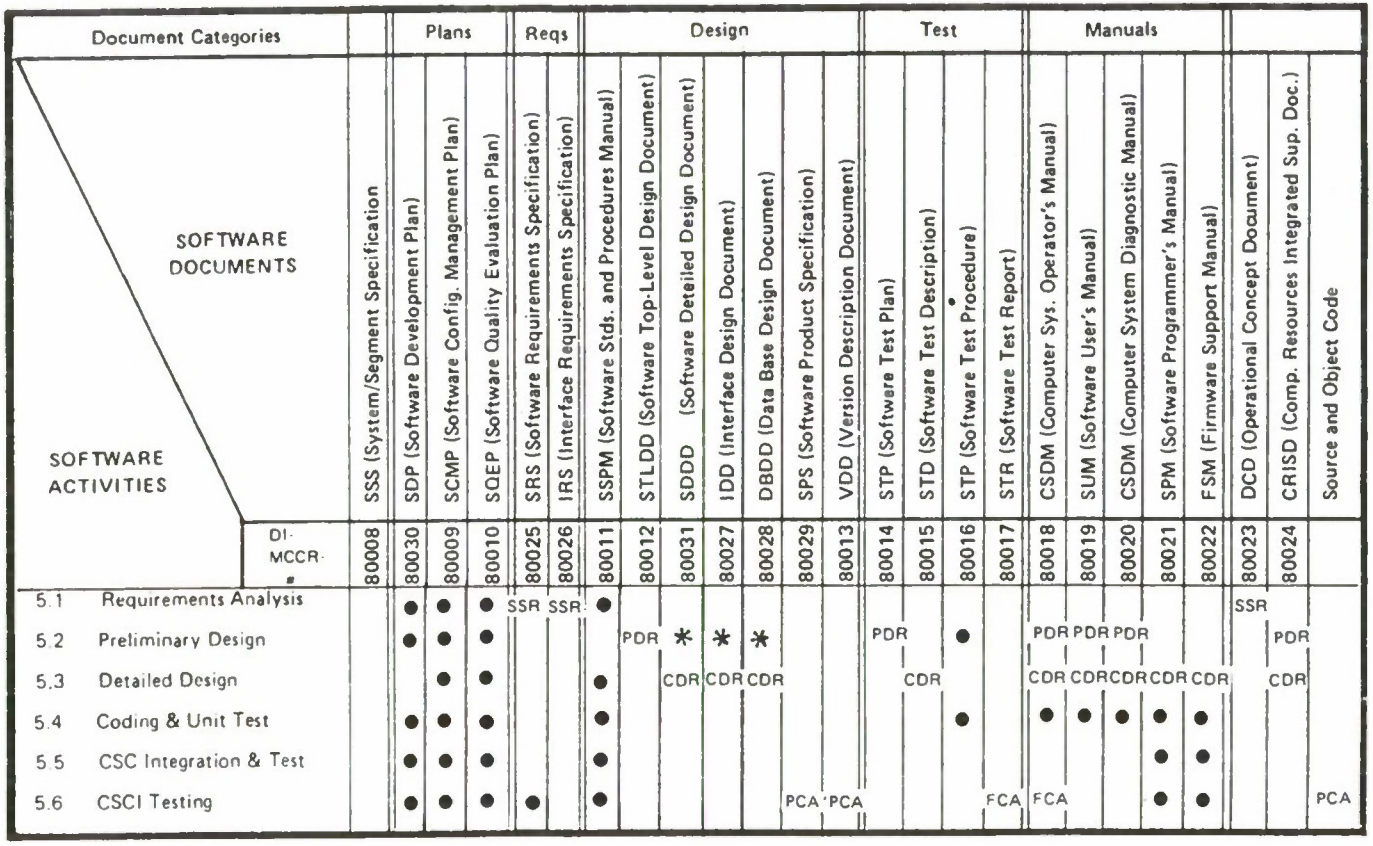


FIGURE 78. DOD-STD-2167, DEFENSE SYSTEM SOFTWARE, PHASES OF SOFTWARE DEVELOPMENT AND FORMAL REVIEWS



● = Submit initial or updated version of document

* = Submit preliminary versions for critical lower-level elements of the CSCI

FIGURE 79. SOFTWARE PRODUCTS AND DESIGN REVIEW BY PHASES

- MIL-STD-1521B, Technical Reviews and Audits for Systems, Equipment and Computer Programs

- An integrated set of related data item descriptions (DIDs)

DoD had already decided by 1977 to fund the development of a new high-order language called Ada for the implementation of DoD mission-critical software. The develop of Ada had commenced in the middle 1970s. It was completed in the early 1980s.

In 1983, Ada was granted American National Standards Institute (ANSI) standardization and its definition was formally documented in ANSI/MIL-STD-1815A.

Thus, the new software development standards and the Ada language were developed over the same time period. Unfortunately, these efforts primarily proceeded independently of one another. During SDS review cycles, Ada-related input was minimal. For example, the Council of Defense and Space Industry Association (CODSIA) reviews of SDS did not directly involve either the Association for Computing Machinery (ACM) or the Special Interest Group Ada (SIGAda).

Nevertheless, DoD has mandated that all new embedded computer programs for mission critical systems must be written in Ada and is also mandating the use of DOD-STD-2167. Other governmental agencies, such as NASA and the FAA, have accepted or are seriously considering the use of both Ada and DOD-STD-2167.

Since the development of Ada and DOD-STD-2167 were essentially independent, explicit and implicit inconsistencies between them are likely. As might be expected, the graphics suggested in design-related DIDs of SDS do not effectively represent Ada-unique designs.

To resolve this problem, the graphics of SHARP can be used to uniquely represent Ada computer programs in SDS design documentation.

1.2 CHAPTER SCOPE

This chapter discusses the use of SHARP in design documentation prepared in accordance with SDS. Section 2.1 addresses a Software Top-Level Design Document (DI-MCCR-80012) and Section 2.2 addresses a Software Detailed Design Document (DI-MCCR-80031). Discussion applicable to the latter is presented in the context of both traditional and object-oriented designs.

2. APPLYING SHARP SOFTWARE DESIGN DOCUMENTS

Block diagrams have been used to represent the architecture of computer programs designed in a conventional manner. For example, Figure 80 shows a CSCI architecture diagram provided in Figure 1 of DI-MCCR-80012, "Software Top Level Design Document."

We feel that such a conventional block diagram does not effectively represent the structure of a computer program to be written in Ada. It does not specifically represent Ada subprograms, tasks and packages -- the basic architectural building blocks of an Ada computer program. Nor does it account for concurrent program unit execution using Ada tasks, or rendezvous between tasks. Also, it does not directly represent unique capabilities of Ada used to implement object-oriented designs, such as the encapsulation of program units in packages. The use of packages to hide information is fundamental to software implementation in Ada.

2.1 APPLICATION OF SHARP IN SOFTWARE TOP-LEVEL DESIGN DOCUMENTS

2.1.1 Sample CSCI Architecture Diagram

DI-MCCR-80012 specifies requirements for a Software Top Level Design Document (STLDD) in which the structure of a Computer Software Configuration Item (CSCI) is documented. For Top Level Computer Software Components (TLCSCs) paragraph 10.2.5.1 of DI-MCCR-80012, in part, states:

> "The relationship among these TLCSCs and critical lower-level computer software components (LLCSCs) and units, if known, shall be described. The CSCI top-level architecture diagram (see Figure 1)."

The figure referenced is provided in Figure 80. We feel this representation has several significant shortcomings. Not only does it not account for Ada-unique program units, it also does not represent the process abstraction aspects of a high level Ada design.

As discussed in Chapter I, an Ada designer uses process abstraction to design a large and complex Ada computer program. In a computer system, resident software typically has to satisfy multiple demands. For example, user commands and communication interface requests may simultaneously compete for a computer's processing time. A computer program must respond in a timely manner to the commands and requests, even when they are received at about the same instant in time.

With languages such as FORTRAN, the threads of nested program units are assigned to processes and execute concurrently under operating system control. An Ada designer implements each thread within the body of an Ada task, declared in the main program.

Figure 81 provides an example of the SHARP tasks nested in the main program. It indicates the external entities serviced by the tasks including communication links, multiple terminals, and work stations.

The diagram also represents access to Ada packages through the Ada "with" clause. Specifically, the main program accesses package TEXT_IO_P1, to provide general purpose I/O capabilities.

The diagram intentionally does not show the structure of each TLCSC Task. The structure may consist of program units interconnected in a traditional top-down manner, as discussed in Section 2.2.1. Alternatively, especially if large and complex, requirements to be implemented in a TLCSC task may be distributed among objects and implemented in an object-oriented manner, as discussed in Section 2.2.2.

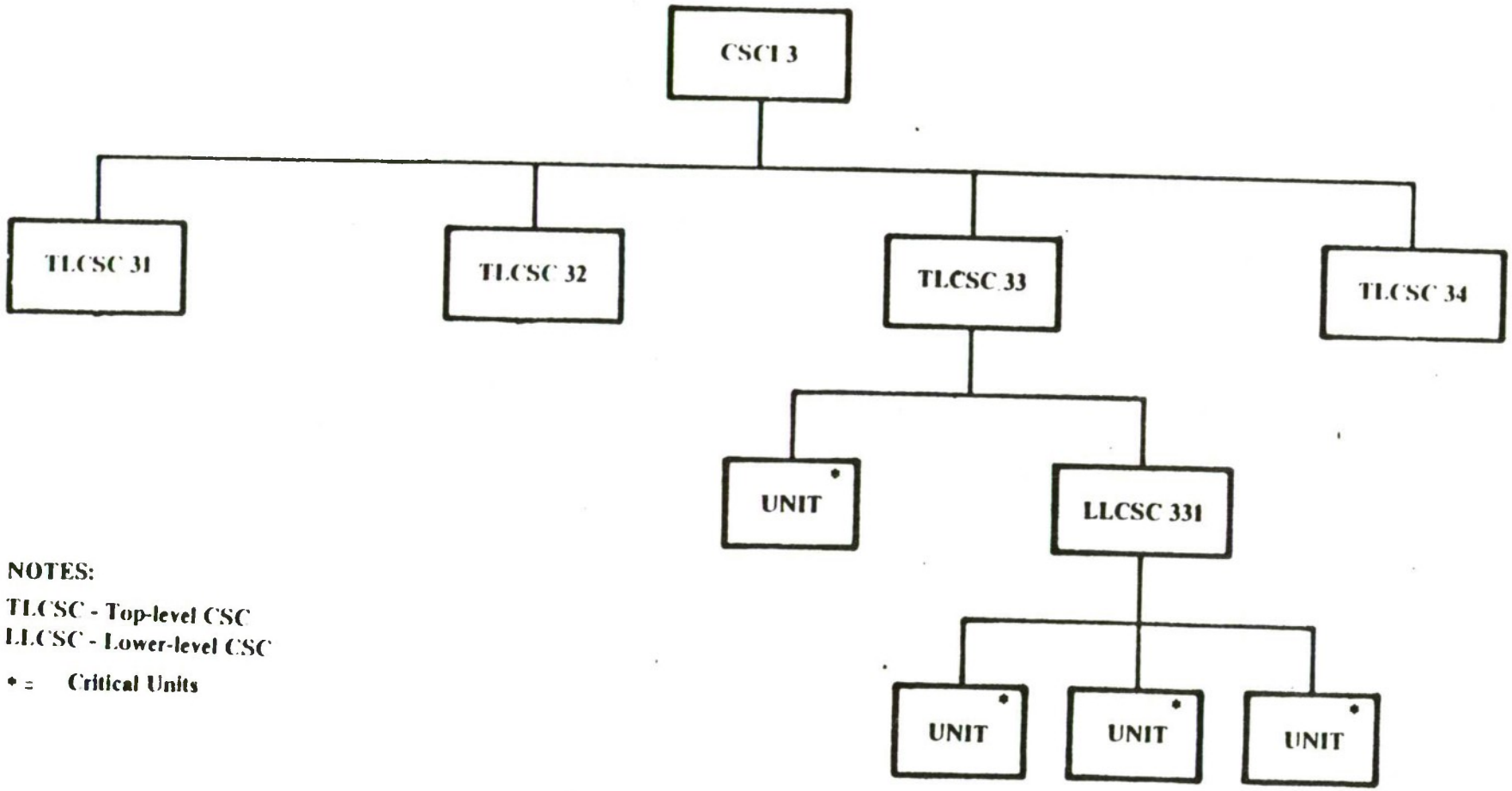


FIGURE 80. CONVENTIONAL CSCI ARCHITECTURE DIAGRAM

This diagram is Figure 1 in DI-MCCR-80012

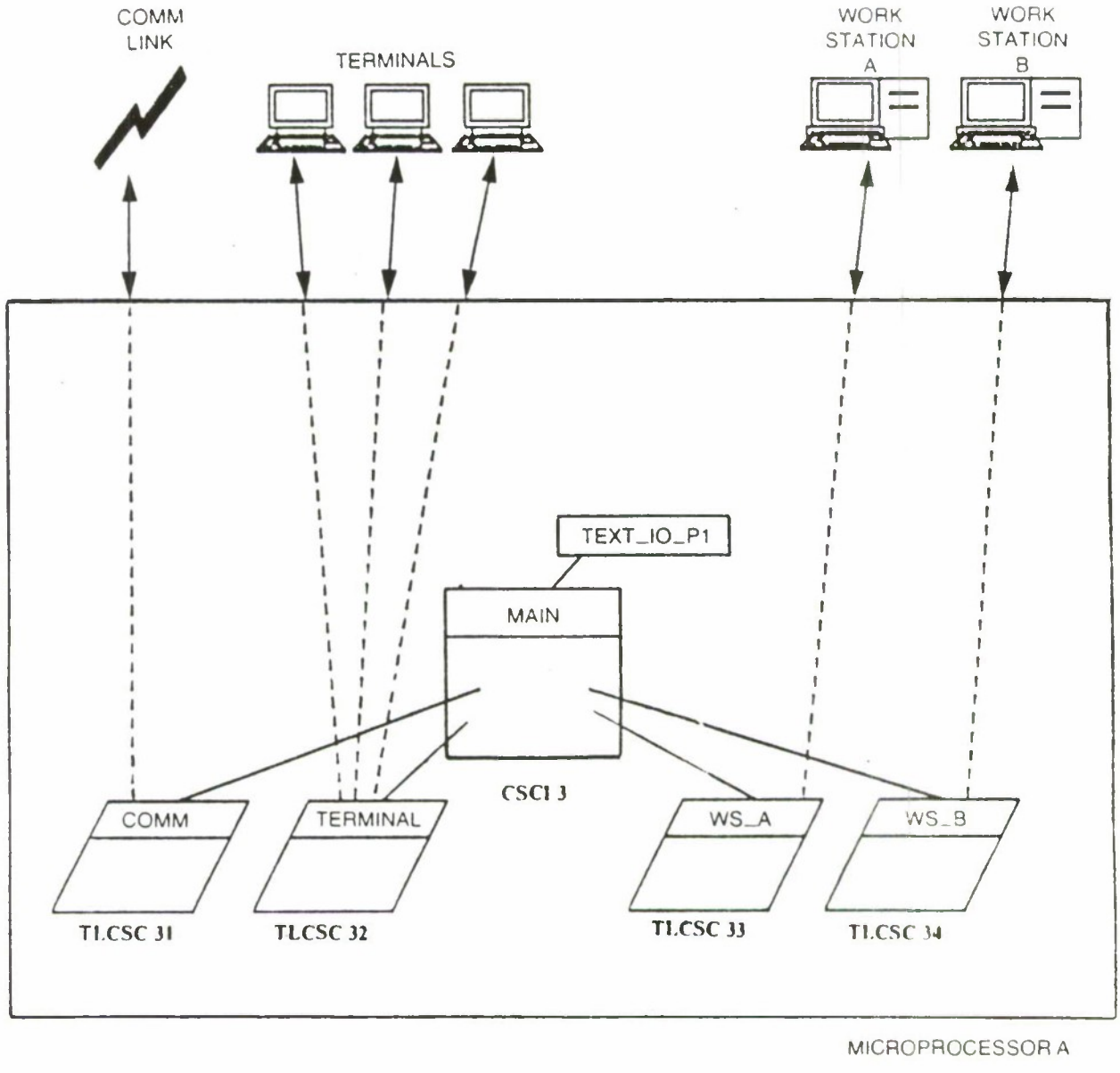


FIGURE 81. SHARP REPRESENTATION OF A CSCI

2.1.2 Sample Diagram for Control and Data Flow Between TLCSCs

Paragraph 10.2.5.4 of DII-MCCR-80012, in part, states:

> "A control flow diagram between TLCSCs may be used to illustrate top-level execution control. An example of a control flow diagram between top-level TLCSCs may be used to illustrate top-level data flow. An example of a data flow diagram is provided in Figure 3."

The first figure referenced is provided in item a Figure 82. The second figure referenced is provided in item b. We feel these figures do not effectively represent the interaction of TLCSC tasks, which establish processes in the high levels of an Ada design. The interaction of tasks is accomplished via a task rendezvous. In a task rendezvous, a caller task initiates the rendezvous while the callee (or acceptor task) consummates the rendezvous.

SHARP provides a specific graphical representation of Ada task rendezvous, as discussed in Section 2.4.3 of Chapter II. Figure 83 provides an example of the SHARP representation of TLCSC task rendezvous.

2.2 APPLICATION OF SHARP IN SOFTWARE DETAILED DESIGN DOCUMENTS

Paragraph 10.2.5.3 of DI-MCCR-80031, in part, states:

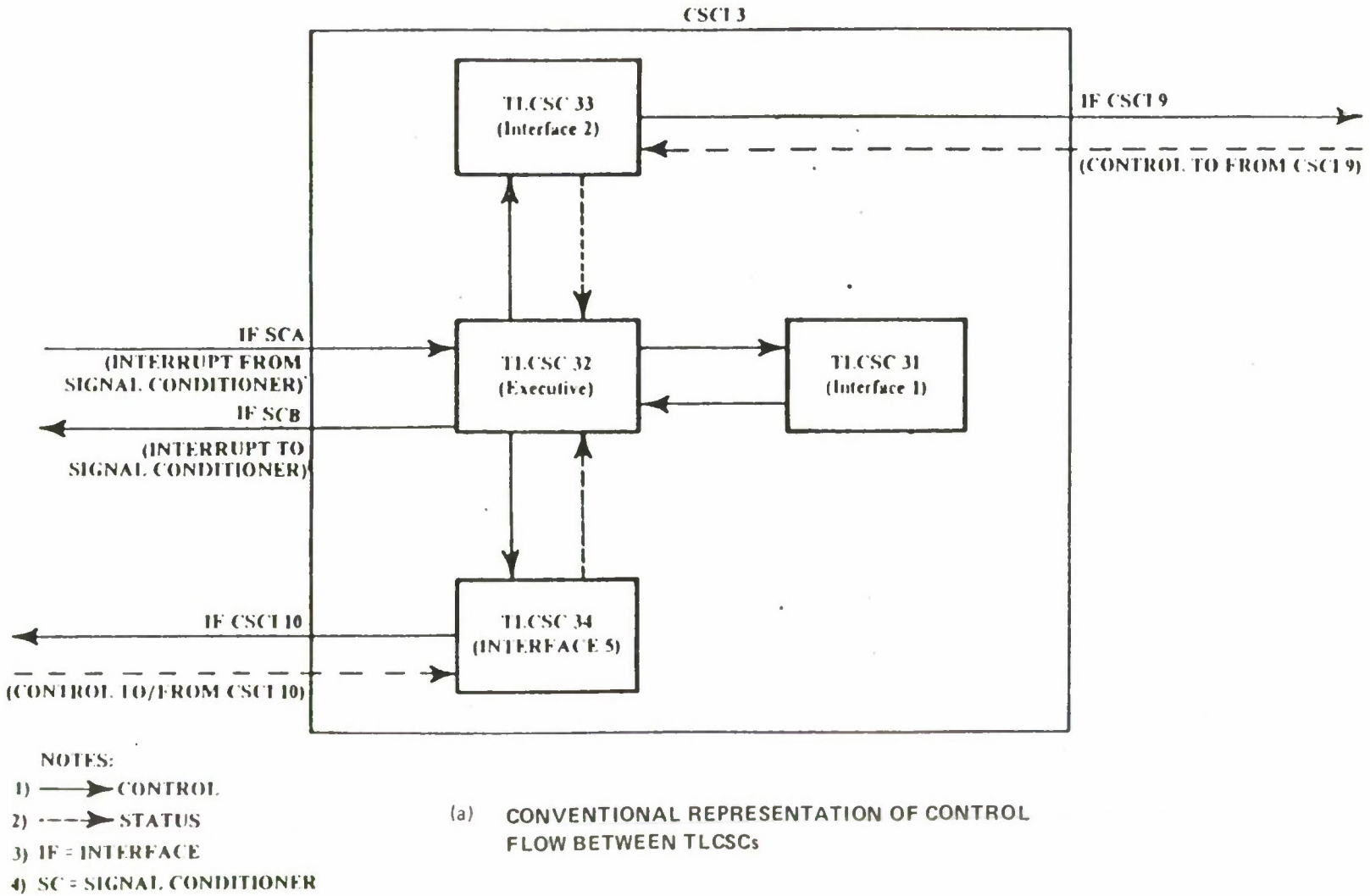
> "This subparagraph shall describe the decomposition of TLCSC W into LLCSCs and Units. This description may be provided by a TLCSC decomposition chart (or a series of charts). Figure 1 is an example of a TLCSC decomposition chart."

The figure referenced is provided in Figure 84. We feel that this figure does not effectively represent the decomposition of a TLCSC, regardless of whether it is constructed using traditional techniques or object-oriented techniques. Not only does it not account for Ada subprograms, it does not represent information hiding in packages and tasks, which is basic to software implementation with Ada.

2.2.1 Sample Diagram Applicable to Decomposition in a Traditional Manner

When a TLCSC is decomposed using traditional techniques, a designer typically will abstract the implementation of applicable requirements in a top-down manner. For example, a relatively small and easily comprehended portion of the requirements can be implemented at one level, with the rest of the requirements implemented at other levels.

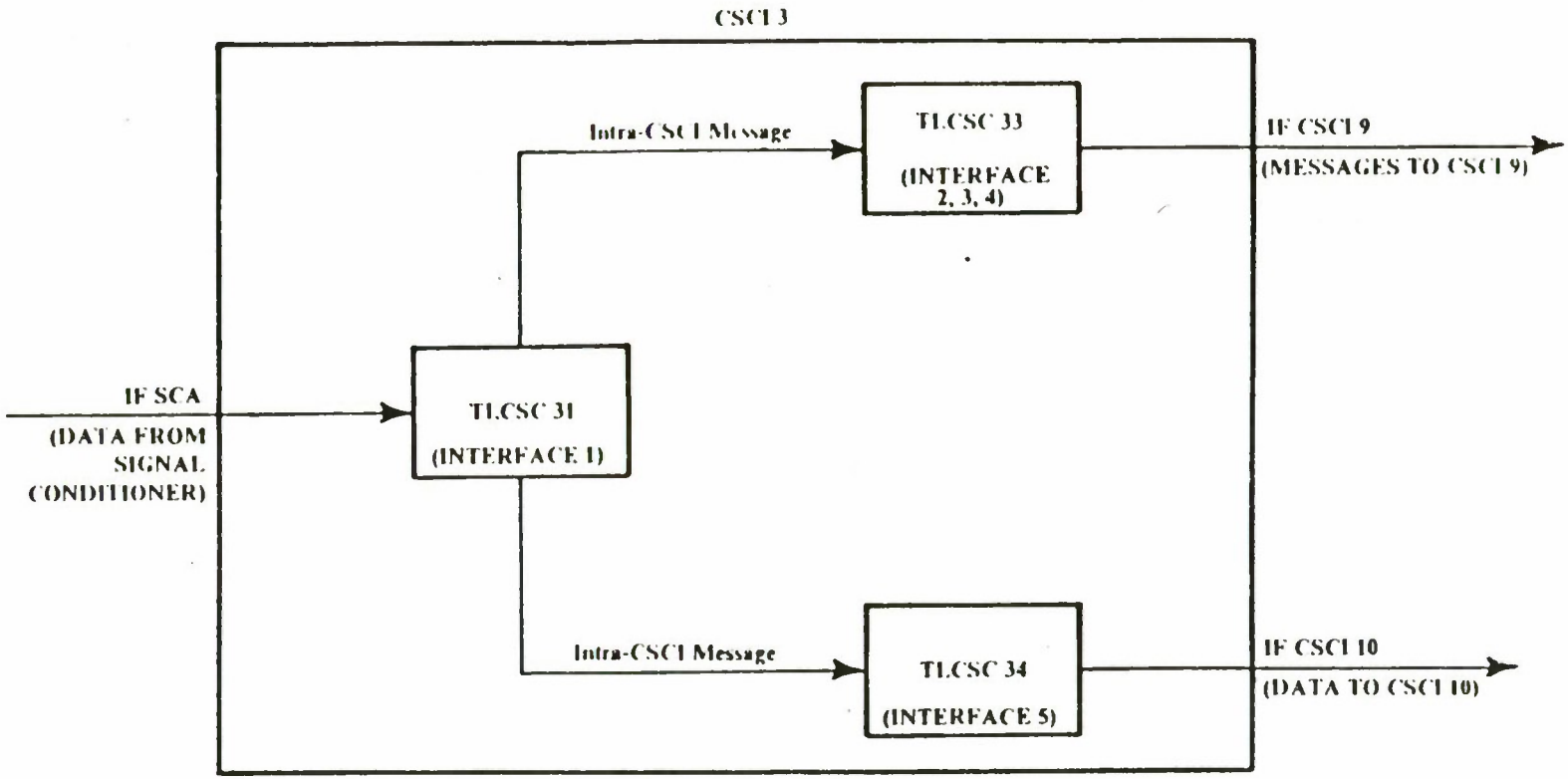
Using this approach, the body of each TLCSC task, activated in procedure MAIN to account for a process, is abstracted by constraining the amount of detail, within it to an easily understood amount. Excluded detail can be passed to the bodies of called program units. The called program may be contained in an Ada package, which is made available though the use of the Ada "with" clause.



(a) CONVENTIONAL REPRESENTATION OF CONTROL FLOW BETWEEN TLCSCs

FIGURE 82. CONVENTIONAL REPRESENTATION OF CONTROL AND DATA FLOW SUGGESTED IN DOD-STD-2167

This diagram is Figure 2 in DI-MCCR-80012



(b) CONVENTIONAL REPRESENTATION OF DATA FLOW BETWEEN TLCSCs

FIGURE 82. (CONCLUDED)

This diagram is Figure 3 in DI-MCCR-80012

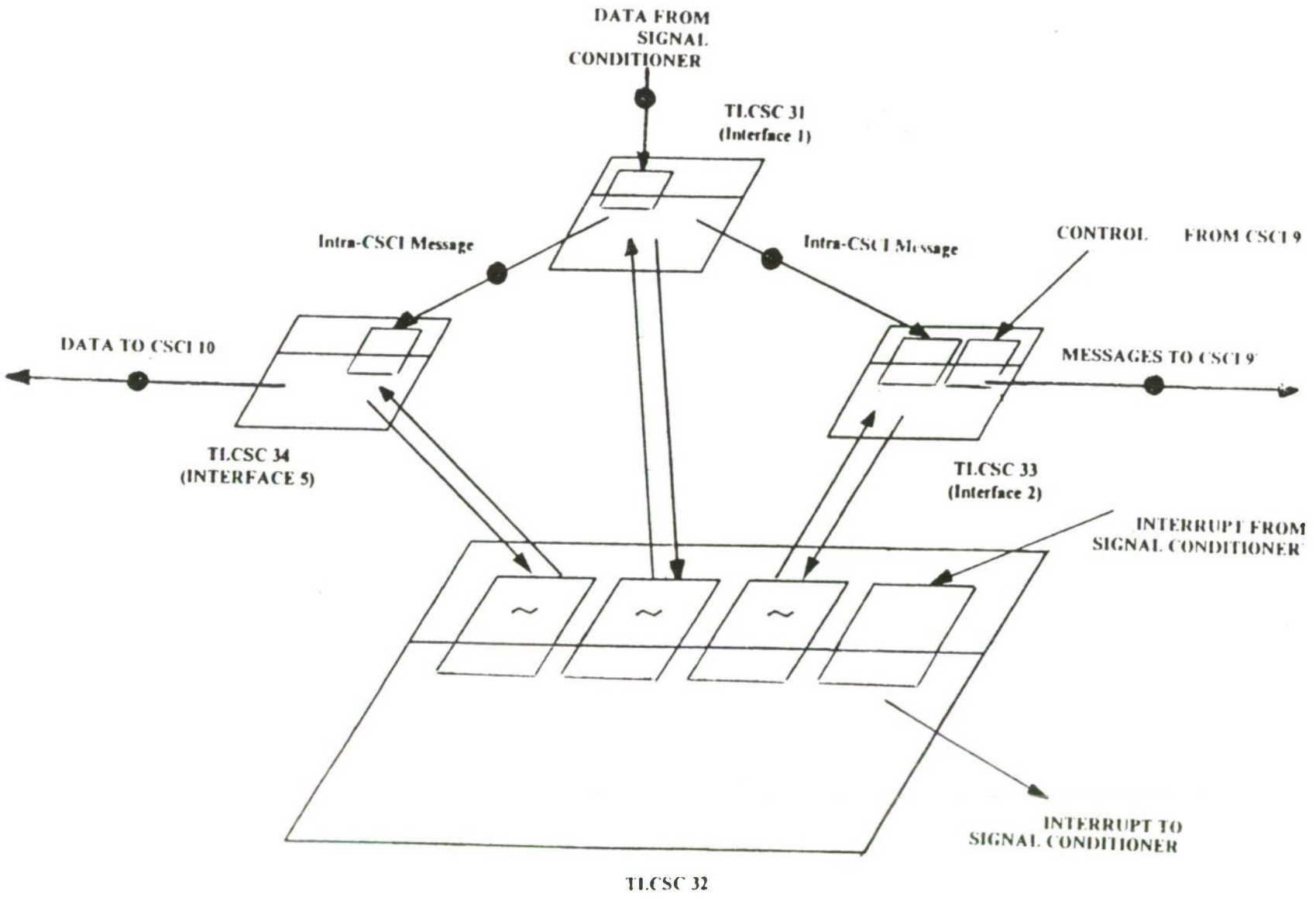


FIGURE 83. SHARP REPRESENTATION OF TLCSC ADA TASK RENDEZVOUS

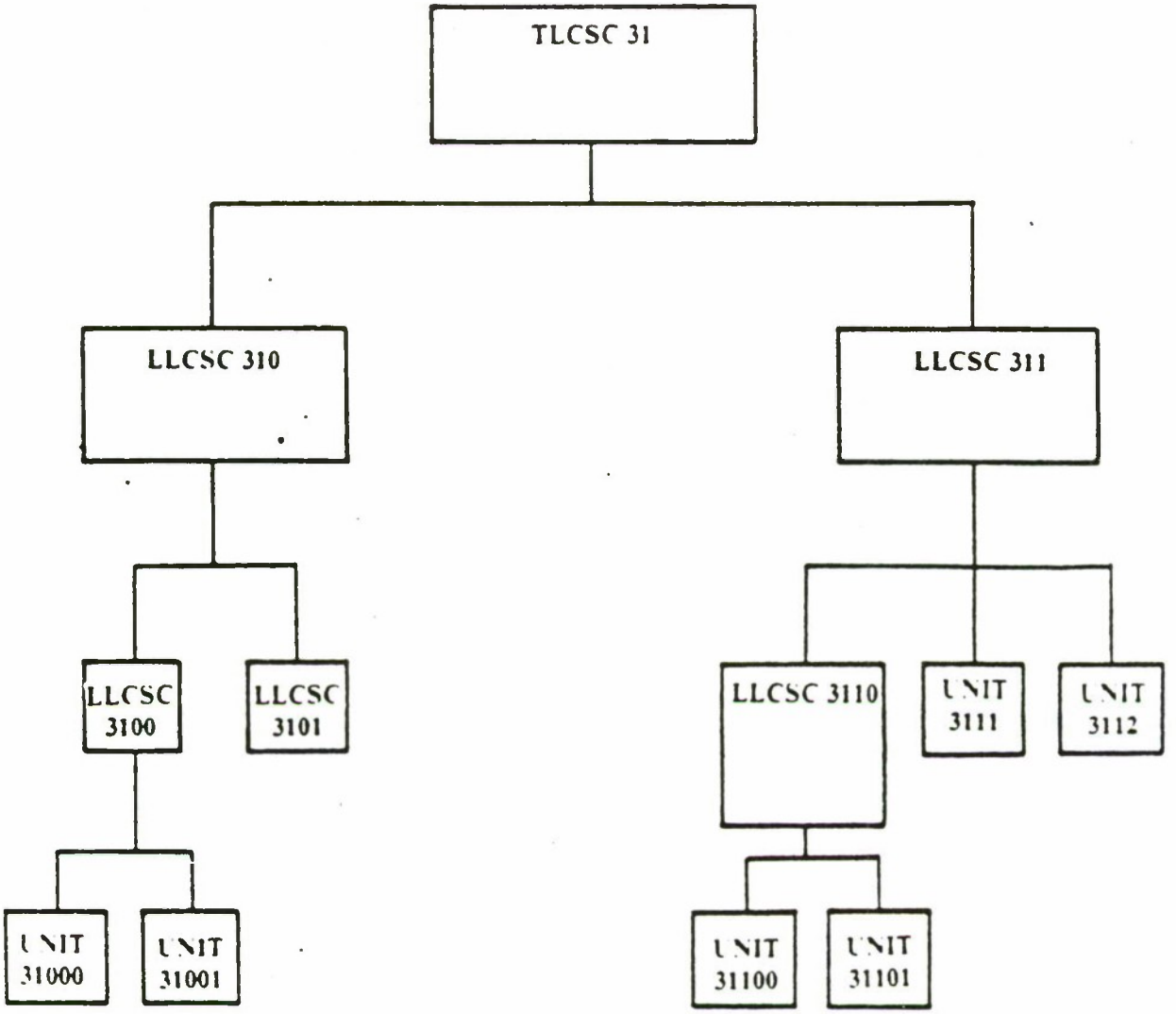


FIGURE 84. CONVENTIONAL TLCSC DECOMPOSITION CHART

This diagram is Figure 1 in DI-MCCR-80031

The deferred bodies of the called program units can be designed subject to the same constraints that applied in the design of the task's body. Therefore, these bodies are also constrained to an easily understood amount of detail, with lower detail moved again to called program units.

A possible Ada-unique structure for the set of program units identified in Figure 84, is shown by the SHARP Hierarchy Diagram in Figure 85. A SHARP Hierarchy Diagram is described in detail in Section 2.3.1 of Chapter II.

Figure 86 illustrates an associated SHARP Invocation Diagram, which is used to show the sequence of calls for the program unit identified in the hierarchy diagram, and program units visible in packages to be accessed using the Ada "with" clause. A SHARP Invocation Diagram is described in detail in Section 2.3.2 of Chapter II.

2.2.2 Sample Diagram Applicable to Decomposition in an Object-Oriented Manner

Requirements for a TLCSC task can be distributed among objects and implemented in an object-oriented manner. Each object has a set of operations unique to it and a local state defined in a data structure. The implementation of the set of operations is not accessible by other objects. With Ada, each object can be implemented in Ada program units encapsulated in an Ada package. In consideration of terminology established in DI-MCCR-80031, these packages are designated as Top Level Computer Software Components (TLCSCs). Access to each object can only be made via calls to procedures in TLCSC packages.

SHARP can be used to represent this object-oriented approach to the decomposition of a TLCSC task, as shown in Figure 87. In this figure, TLCSC packages encapsulating the implementation of object requirements are shown. In Figure 88, the internal structure of program units visible in the TLCSC packages are shown. The use of SHARP in this way to represent an object-oriented design is discussed in detail in Chapters II, III and IV.

3. CHAPTER SUMMARY

SHARP abstracts can be effectively used in a Software Top Level Design Document (DI-MCCR-80012) and in a Software Detailed Design Document (DI-MCCR-80031) to graphically represent the design of an Ada computer program. In this manner, inconsistencies between the graphics appropriate for traditional designs and graphics appropriate for Ada-oriented designs are removed. In particular, the SHARP abstracts can be introduced into a Software Detailed Design Document to represent an object-oriented Ada design. This is especially significant since it is expected that object-oriented techniques will be widely used in conjunction with reusable software components, in the implementation of large and complex Ada computer programs, with significant software development cost savings projected.

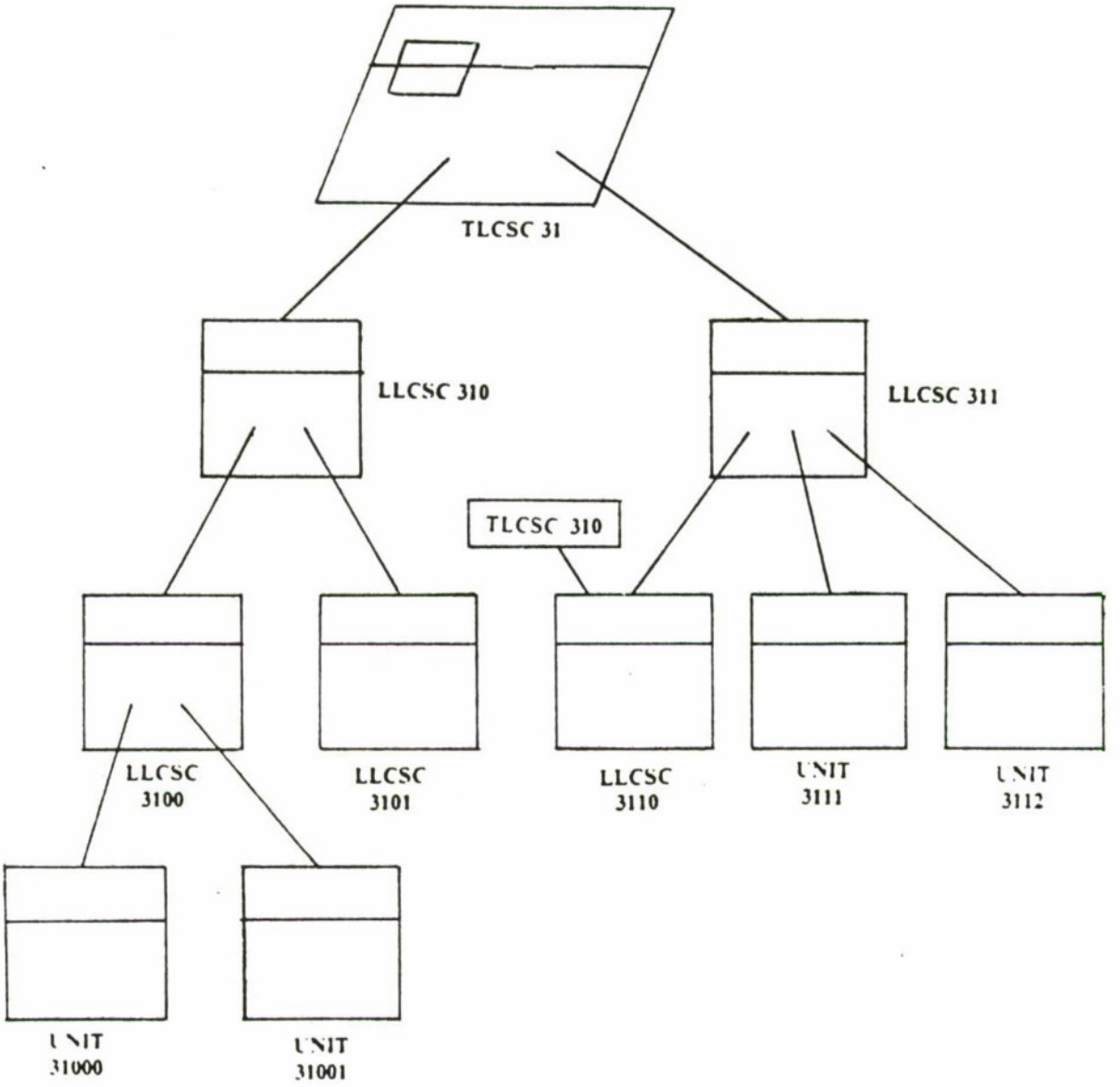


FIGURE 85. SHARP REPRESENTATION FOR A TLCSC TASK HIERARCHY DIAGRAM

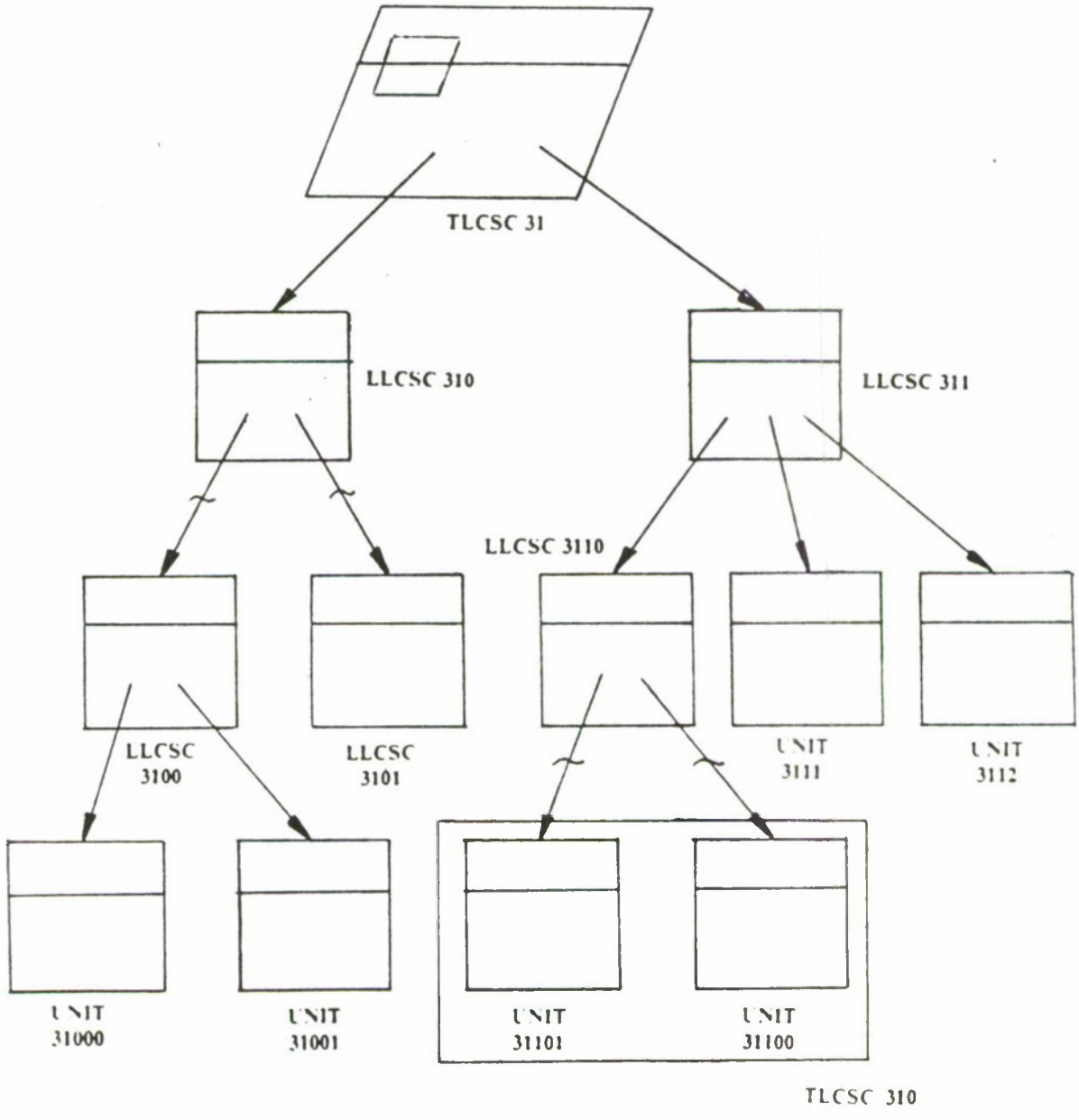
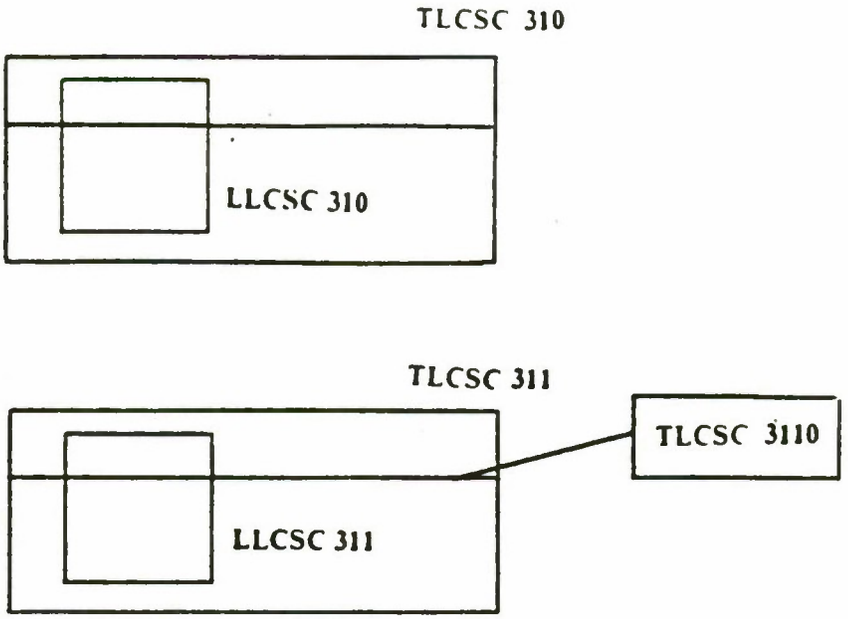
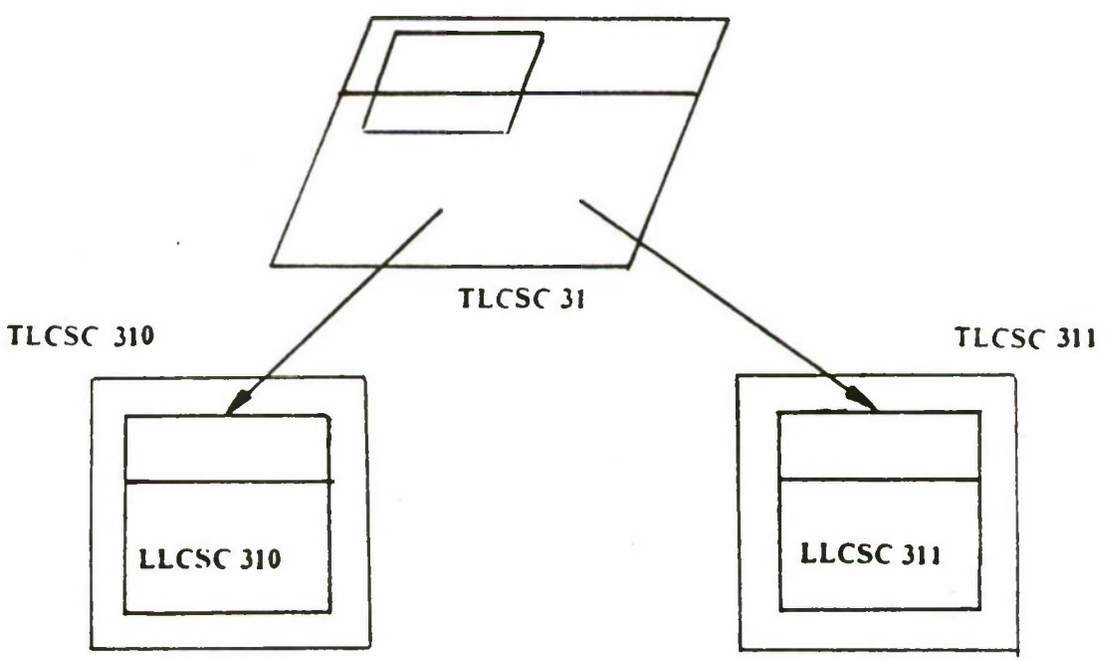


FIGURE 86. SHARP REPRESENTATION FOR A TLCSC TASK INVOCATION DIAGRAM



(a) LLCSC ADA PACKAGES USED TO IMPLEMENT OBJECTS



(b) INVOCATION OF LLCSC ADA PACKAGES

FIGURE 87. SHARP REPRESENTATION OF TLCSC ADA PACKAGES (OBJECT-ORIENTED DESIGN)

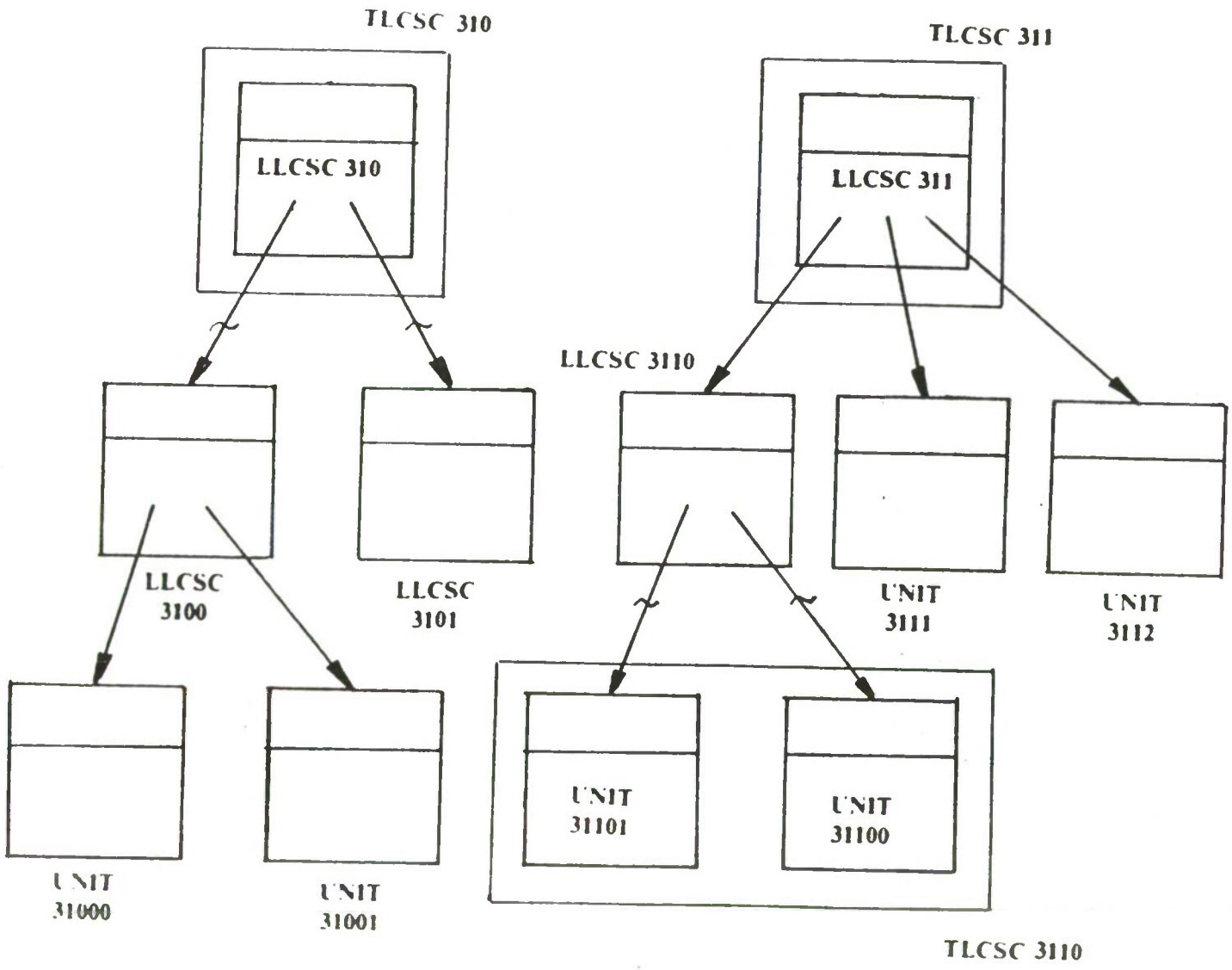


FIGURE 88. REPRESENTING THE INTERNAL COMPLEXITIES OF AN OBJECT IMPLEMENTATION

CHAPTER VI

Testing Object-Oriented Ada Software

This chapter describes a systematic approach to testing an Ada computer program that has been designed in an object-oriented manner. Object implementations can be tested by verifying that their responses to environmental stimuli are correct. To comprehensively test the implementations, the environmental stimuli should establish both nominal conditions and conditions that stress their interaction.

In order to facilitate such testing, we describe basic Ada test packages that can be introduced to simulate environmental stimuli and to record parameters passed from one object to another. These test packages provide the framework for verification of object interaction at a high level, suitable for government formal test of Computer Software Configuration Items (CSCIs) established in accordance with DOD-STD-2167.

This chapter also addresses lower level testing, which typically precedes the high level testing or is introduced to assess problems revealed by the high level testing. Lower level tests check the performance of the implementation of each object individually.

For both the high level and lower level testing, we describe the use of SHARP abstracts to explicitly represent test software and its interaction with the operational software to be exercised. We illustrate that SHARP diagrams are effective, both to indicate high level tests of object interaction and lower level tests of program units used to implement the complex requirements assigned to an object.

This chapter demonstrates that the testing of software designed in an object-oriented manner can be explicitly represented using SHARP abstracts. Such a pictorial representation significantly helps us comprehend the scope of tests being undertaken.

1. INTRODUCTION

1.1 BACKGROUND

The localization of design complexity is the important feature of an object-oriented design that simplifies testing during software development and maintenance. With the localization of design complexities within object implementations, and the establishment of straightforward interfaces between the implementations, software testing becomes relatively straightforward and easy to undertake in practice. This is because undesirable dependency relationships within a computer program have been constrained.

In the past, global parameters and routines were shared among many program units. If during testing any one of the global parameters or routines were modified to correct an error, the change could adversely affect several different parts of the computer program. When one error was corrected by a

change, several others often were introduced. Seemingly innocent changes, at times, caused serious problems. However, by localizing design complexity with an object-oriented design, the effect of a change is trapped within the implementation of an object itself.

In Ada, packages are important to the implementation of object-oriented designs and the testing of the implementations. Only program units declared in a package's specification can be directly accessed. Operations and data can be hidden within a package's body, where they cannot be directly accessed by program units in other packages. Thus, an Ada package is an ideal program unit for encapsulating special test software and an object implementaion.

1.2 CHAPTER SCOPE

In this chapter, a systematic approach to testing object-oriented Ada software using special test software is described. High level and lower level tests are identified. Special test software is identified and described. The representation of the special test software and its interaction with object implementations is graphically represented using SHARP abstracts.

2. TESTING A COMPUTER PROGRAM IMPLEMENTED IN AN OBJECT-ORIENTED MANNER

In the development of object-oriented software, a team of programmers will develop software needed to implement requirements assigned to individual objects. This team must extensively test the implementation of an individual object and verify that integrated object implementations perform correctly. Then, the programming team typically will turn over the software to a test team.

The test team will execute the software in conjunction with system hardware. It will run tests designed to verify that the integrated object implementations satisfy predefined requirements. If incorrect performance is detected, the software must be returned to the programming development team.

The programming development team must run new tests on the individual object implementations (e.g., enhancements of their previous tests) to determine the implementations not performing correctly. Once a faulty object implementation has been detected, its internal structure must be tested to isolate faulty program units within it and the incorrect code within the faulty program units.

This section discusses the ramifications of such testing of object-oriented software. Specifically, paragraphs in Section 2.1 consider high level tests of object interaction and paragraphs in Section 2.2 address low level tests of the implementation of a single object and its internal structure. In both cases, SHARP abstracts are used to represent special test software and its interaction with Ada application software.

2.1 HIGH LEVEL TESTS OF THE INTERACTION OF OBJECT IMPLEMENTATIONS

For a large and complex Computer Software Configuration Item (CSCI) to be implemented in Ada, an object-oriented approach to design is expected to be widely applied. DOD-STD-2167 and its companion test unique data item descriptions (DIDs), describe government requirements for testing the CSCIs and test documentation that must be prepared. Specifically, the following documents must be prepared as part of the formal test of a CSCI:

- Software Test Plan (to define the scope of testing per DI-MCCR-80014)
- Software Test Description (to identify input data, expected output data and evaluation criteria per DI-MCCR-80015)
- Software Test Procedure (to describe test steps, expected results for each step and test data sheets per DI-MCCR-80016)
- Software Test Report (to document test results and provide an analysis of CSCI performance, including any detected deficiencies, limitations or constraints, all per DI-MCCR-80017)

Testing of a CSCI an be envisioned as a mapping of software requirements specified in a Software Requirements Specification (DI-MCCR-80025) into test stimuli for a set of test cases. When a large and complex CSCI consists of several interacting object implementations, the test stimuli are used to exercise the set of implementations under both nominal and stress conditions. In addition, spectrum of tests can be performed on critical implementations, to the extent time and money permit. The performance of the object implementations is measured by recording parameters passed between the implementations and the process tasks declared in the main program. The recorded data is compared to expected values, either directly or after data reduction.

This section discusses such testing of object implementations in Paragraph 2.1.3. As a prerequisite to this discussion, it describes in Paragraph 2.1.1 basic concepts for object implementation in Ada; and describes in in Paragraph 2.1.2 special Ada-unique test packages introduced to generate test stimuli and record information being passed between object packages.

2.1.1 Layers of Object Packages

Grady Booch suggests that a large software system should be built with layers of abstraction.[6] He feels that each layer should account for one or more objects.

With SHARP, abstracts can be used to represent the layers of packages and tasks typically used to implement objects. For objects implemented using packages, the SHARP Ada Package Content Diagram (Option B in Chapter II) can be used to represent the package layers, as illustrated in Figure 89.

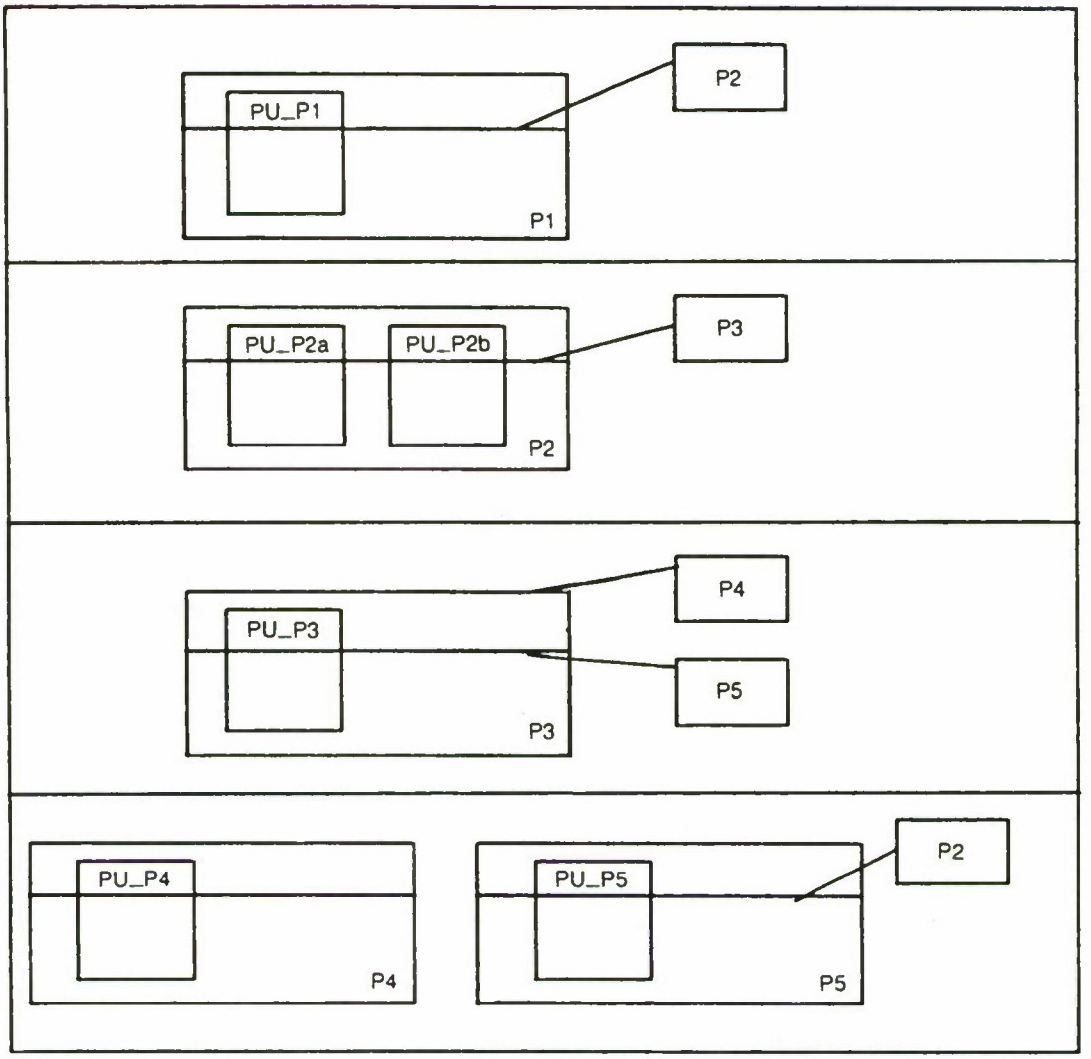


FIGURE 89. LAYERS OF ADA PACKAGES

Program units declared in the specifications of packages facilitate interfaces between the packages. The information transfer between these program units establishes object interaction. This interaction can be represented in a SHARP Invocation Diagram. As shown in Figure 90, a designer will typically require the highest level object implementation, of a set of layered implementations, to interact with a task declared in the main program. This task acts as a substitute for a process directly controlled by an operating system, which is the technique used to facilitate for real-time control with a language like FORTRAN. The task provides a dynamic interface with environmental entities external to a computer (e.g., an interface with a work station, terminal or communication link), as discussed in Section 3.1 of Chapter I.

2.1.2 Special Test Software

Special test software can be introduced to verify the correct performance of object-oriented Ada software. With SHARP, the special test software can be shaded to distinguish it from applications software, as illustrated by the SHARP diagrams provided in this section.

2.1.2(a) Test Driver Package

As part of the framework for the testing of object interaction, an Ada package can be introduced to generate test case stimuli. Using this package, various test cases can be initiated under operator control. A task declared in the specification of this package can stimulate environmental stimuli. Utilizing task rendezvous, it can pass this stimulate to the environmental interface task, as illustrated in Figure 91.

Since the interaction of object implementations may be conditional, only a subset of the implementations may respond to stimuli produced for a particular test case. The subset of objects exercised during a particular test case can be shown in bold face, as illustrated in Figure 92.

2.1.2(b) Environmental Simulator Package

To test object interaction in a more realistic and dynamic manner, an Ada package can be used to encapsulate Ada program units that generate a sequence of stimuli over time. An Ada task declared in the specification of this package can interface with visible program units in object implementation packages, and can respond to feedback received from the object implementations under test, as shown in Figure 93.

The environmental simulator package is initialized by a tester to establish the sequence of stimuli to be generated. During test execution, it typically performs independent of the tester.

2.1.2(c) Data Recording Package

In conjunction with a test driver or environmental simulator package, a data recording package can be introduced to record parameters passed between object implementations. This can be accomplished with a set of program units that are called just prior to the passing of data. The

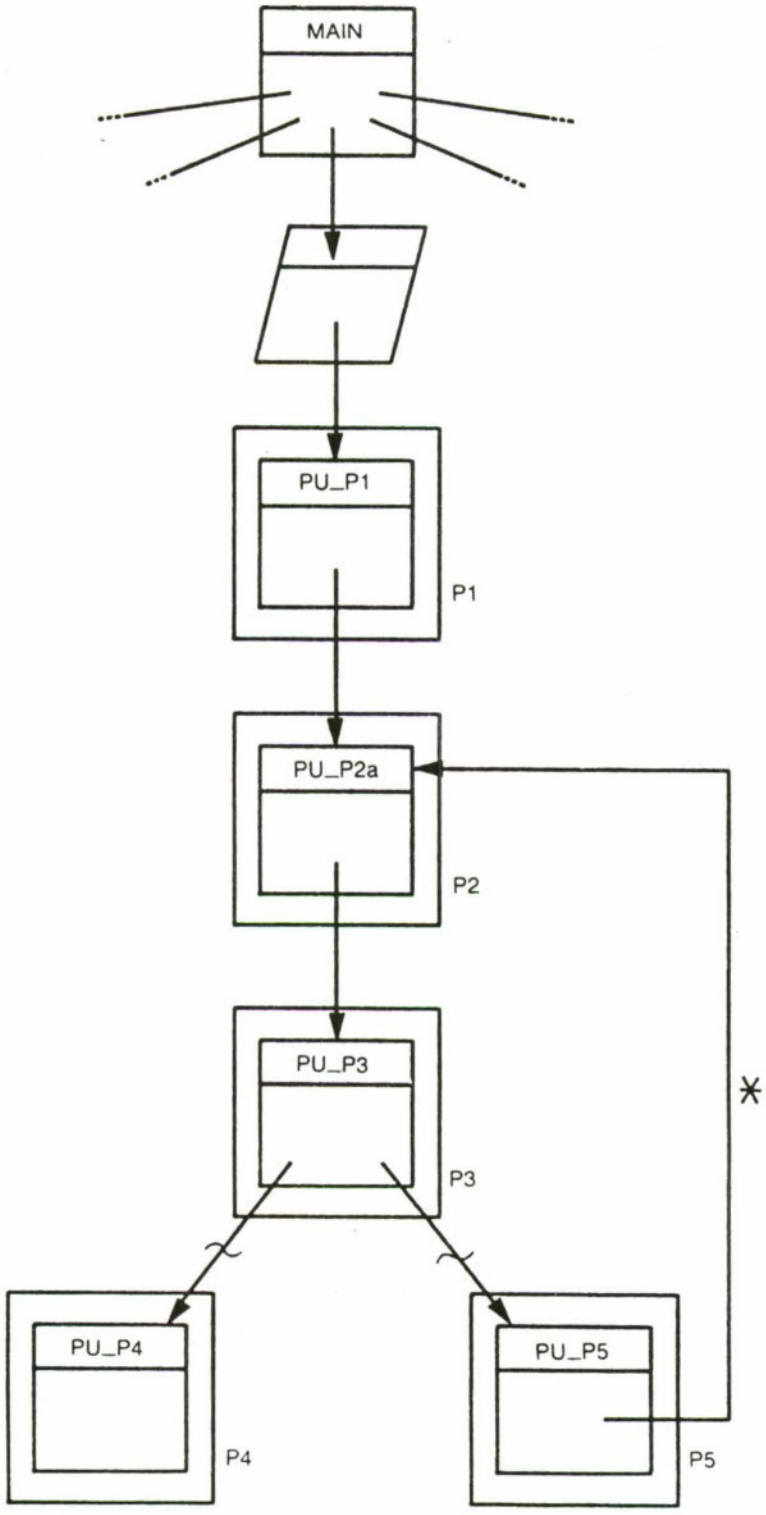


FIGURE 90. OBJECT INTERACTION

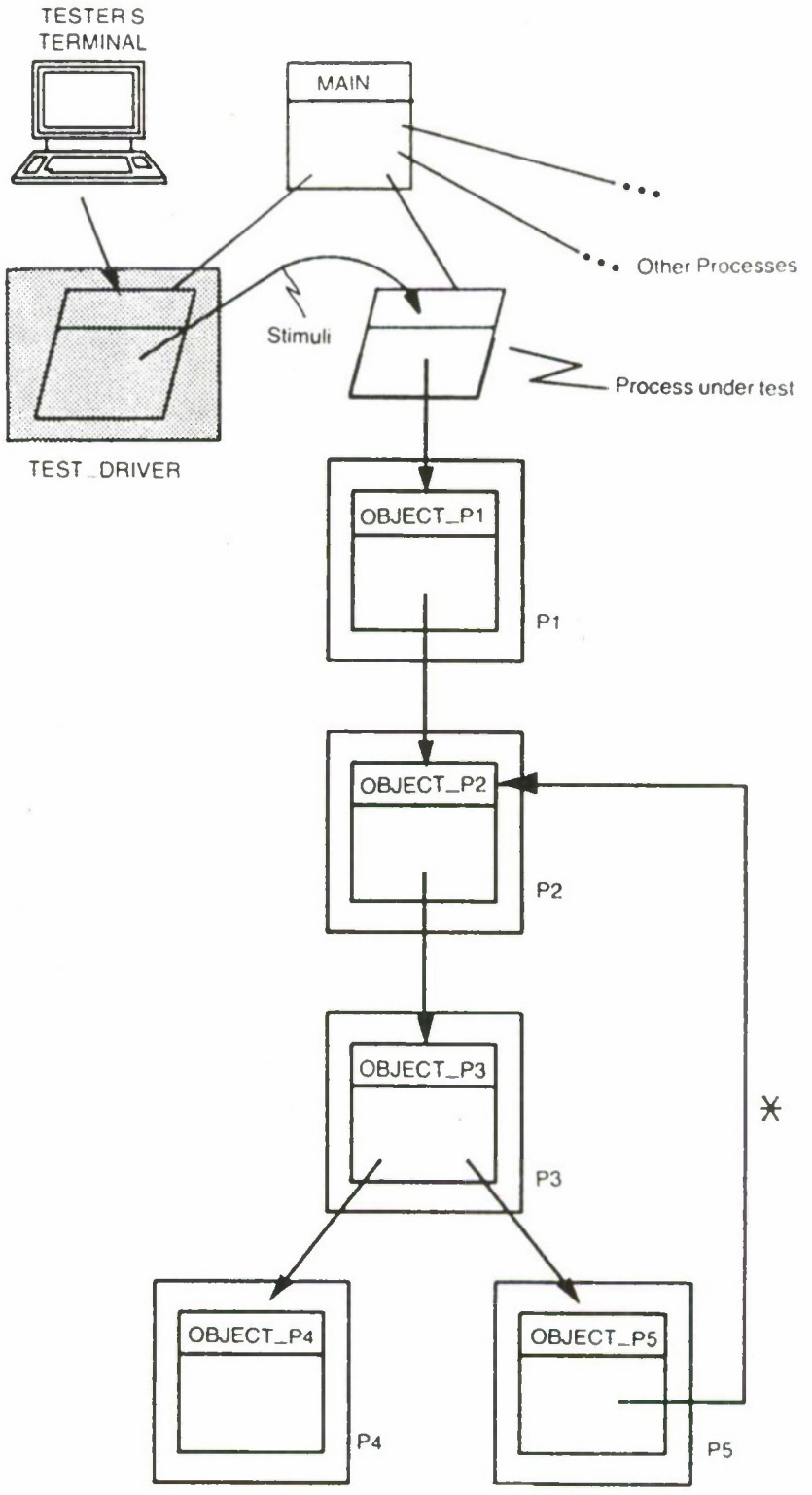


FIGURE 91. TEST OF OBJECT INTERACTION WITH A TEST DRIVER

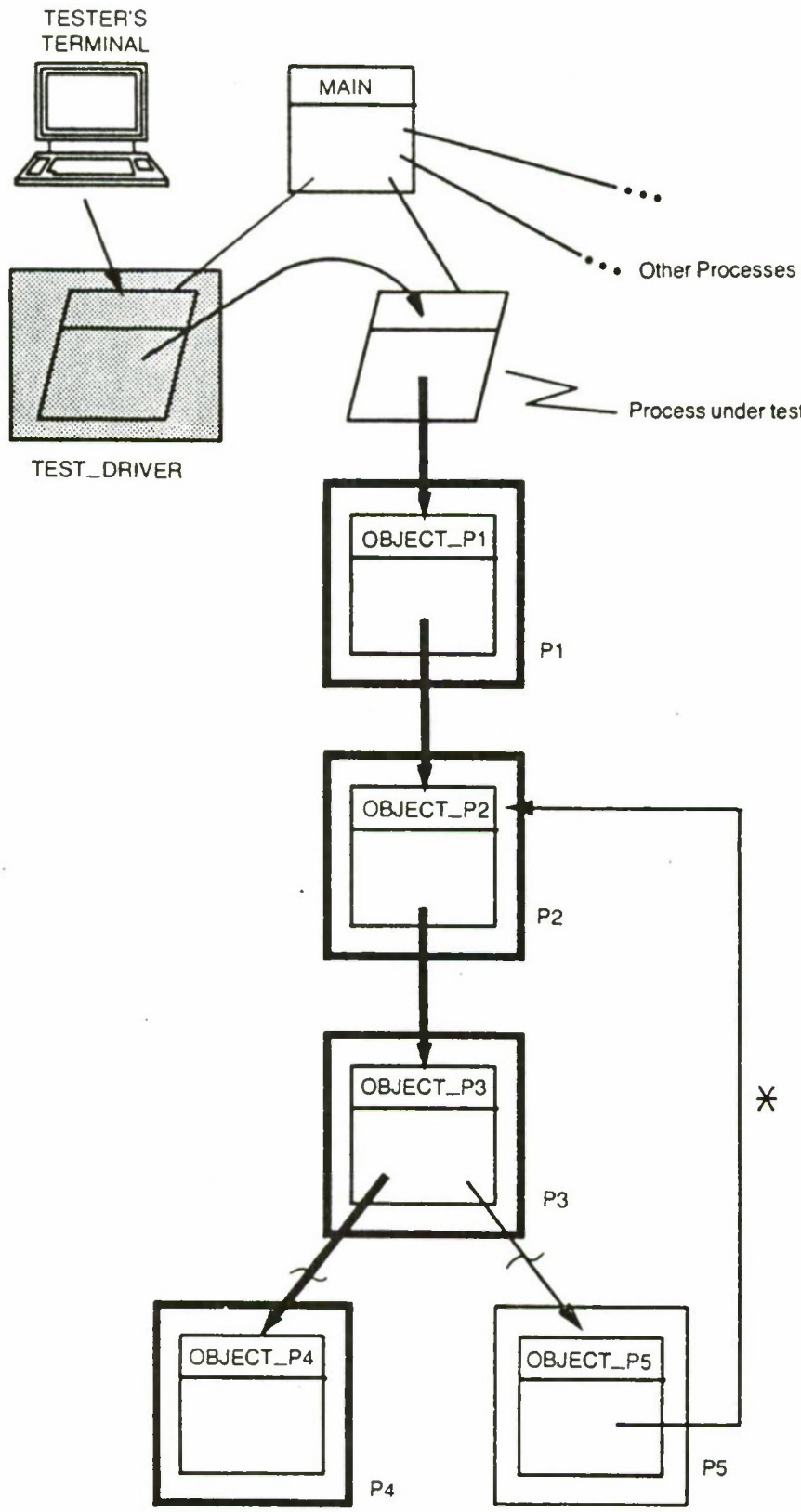


FIGURE 9 2. REPRESENTING TEST OF A SUBSET OF OBJECTS

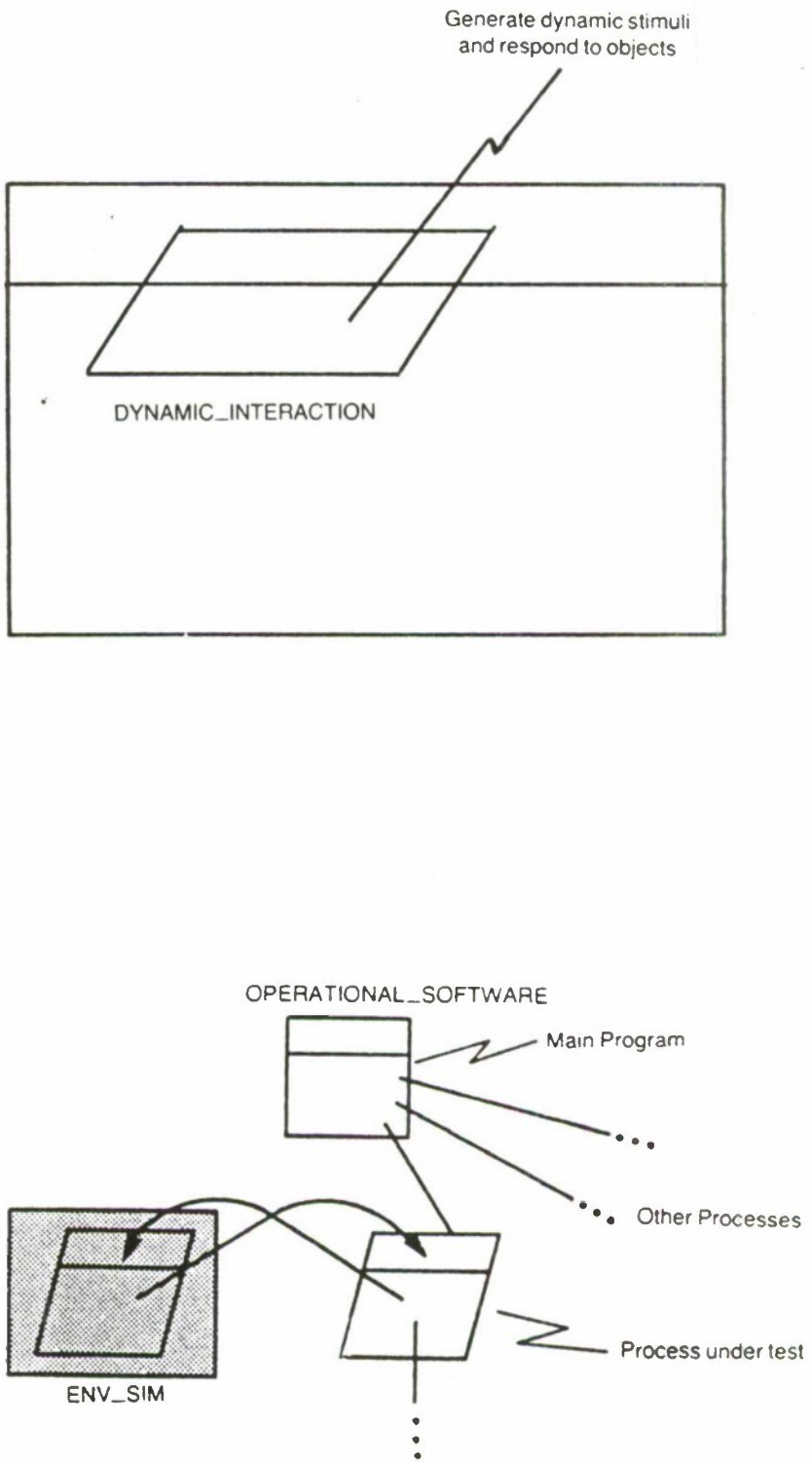


FIGURE 93. TEST OF OBJECT INTERACTION WITH AN ENVIRONMENTAL SIMULATOR

specification of these program units must be identical to the interfacing program units that facilitate data passage. The bodies of these program units establish a record of data being passed and the time of the passage.

The introduction of such recording program units is illustrated by the SHARP pictographs in Figure 94. As shown, the call to a recording program unit may be made conditional. In this way, appropriate conditions can be established prior to test execution, defining what data passage is to be recorded and what data passage is not to be recorded.

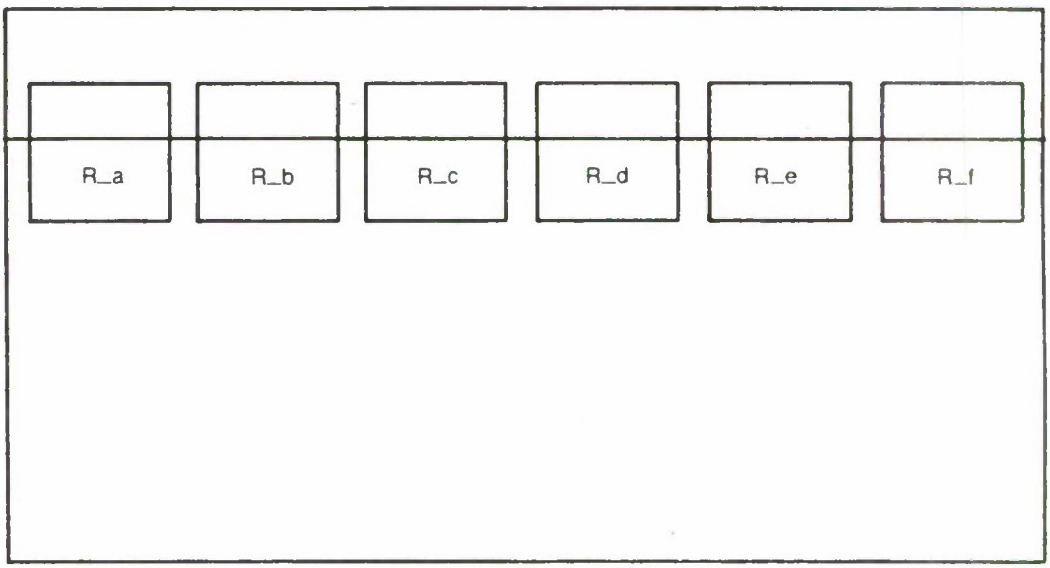
2.1.3 Testing Object-Oriented Software

To test a set of objects, stimuli are introduced by the test driver package (essentially for one test case at a time) or by the environmental simulator package (for more complex dynamic interaction between the object implementations and the environment external to them).

The stimuli introduced should relate to a complete set of nominal software requirements. Nominal testing introduces stimuli and operating conditions typical of those to be experienced by the software in an operational "real world" environment. This testing is designed to exercise the software under expected conditions, to ensure that all object implementations are performing as required under such conditions.

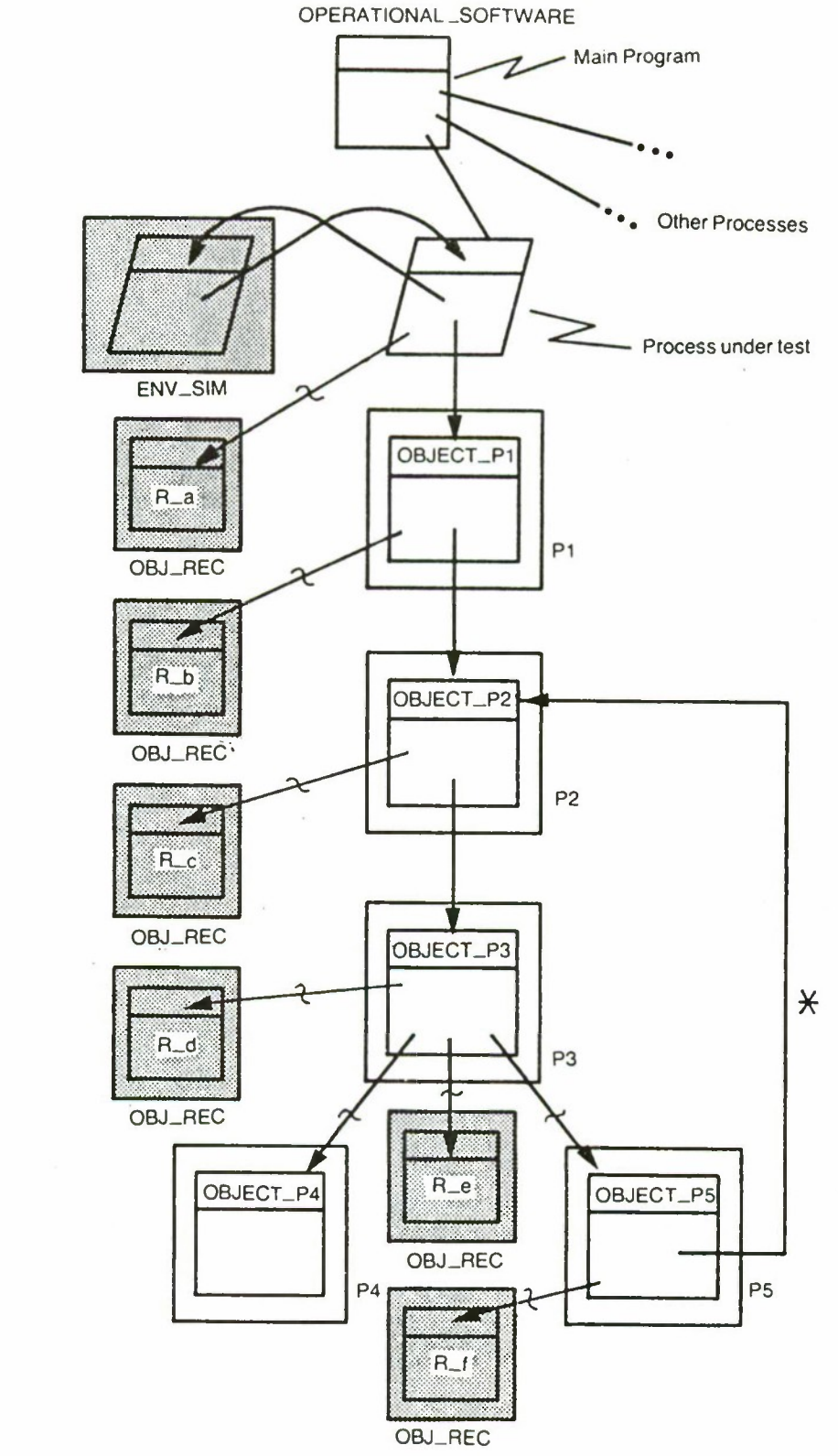
Also, extreme values of the stimuli (e.g., sets of boundary conditions) should be generated to stress the layered object implementations. Stress testing introduces stimuli and operating conditions that will subject the software to possible extreme conditions. This testing makes maximum, overload or even erroneous demands on object implementations. It can include simulated breakdown of interfacing hardware and other unexpected conditions that could occur in the operational environment. Attention is often given to unusual combinations of events that should have been anticipated in the object-oriented design. These extreme values of stimuli can be generated by either the test driver package or the environmental simulator package. Furthermore, the environmental simulator package can also stress the layered object implementations with respect to the frequency at which stimuli are presented to them.

In addition, if time and money are available, the set of object implementations can be further assessed by monitoring their performance over a spectrum of tests. Such testing is especially applicable to critical interaction of object implementations (e.g., objects associated with life dependent events). Performance spectrum testing uses a large number of test cases designed to exercise software over its full range of input values. Analysis can be performed on the results to evaluate overall behavior of integrated object implementations. Test cases are carefully selected to incorporate both nominal and extreme conditions, and to include as many combinations of representative input values as possible. Tests results are analyzed, either manually or with the aid of statistical analysis tools, to identify patterns, detect anomalies and biases in overall results, and draw conclusions about program performance. Graphs and tables of results typically are prepared to aid in the analysis.



(a) Packages ENV_SIM and OBJ_REC

FIGURE 94. RECORDING DATA PASSED BETWEEN OBJECTS



(b) Introducing Data Recording Subprograms

FIGURE 94. (CONCLUDED)

Results may be used to identify special cases requiring additional testing, to define the range of missions that can be successfully accomplished by the object-oriented software, and to compare the performance of different software versions.

2.2 LOWER LEVEL TESTS OF A SINGLE OBJECT IMPLEMENTATION

In the development of object-oriented software, object implementations are tested individually and then integrated prior to formal testing. If during trial runs of formal tests, incorrect performance is detected, lower level tests of individual object implementations must be enhanced and repeated to determine the reason for the incorrect performance.

Each object implementation can be individually tested to determine if it is causing the poor performance. When a faulty object implementation is detected, its internal structure can be tested to isolate faulty program units within it. The faulty program units can, in turn, be tested to finally isolate incorrect code. In this manner, problems can be systematically assessed and isolated.

2.2.1 Testing a Single Object Implementation as a Whole

To identify a faulty object implementation, each of a set of implementations must be individually tested. To test a single implementation, a tester needs a test driver package and a recorder package. The complex hidden structure of the object implementation along with the test driver and recorder packages can be graphically represented in a SHARP Invocation Diagram, such as in the manner shown in Figure 95.

Extreme values of stimuli can be generated to stress the object implementation. These stimuli can be introduced either statically or at high frequency rates. Pictographs for the set of program units exercised in a particular test case can be shown in bold face within the SHARP representation of the test case. As illustrated in Figure 96 for two of a set of multiple test cases, a set of test cases is needed to exercise various paths through the hierarchy of program units used to implement the internal structure of a complex object. Within a SHARP Invocation Diagram, bold face can be used to indicate program units exercised in a particular test case.

2.2.2 Testing the Internal Structure of an Object Implementation

Once a faulty object implementation has been detected, its internal structure must be exercised to isolate the problem. When an object itself consist of a set of objects, the interaction of the internal implementation of objects can be tested in the manner discussed in Subsection 2.1.

In practice, an object typically is implemented using several levels of abstracted subprograms encapsulated in an Ada package (or task). Detail contained within each subprogram's body is constrained to an easily understood amount. Excluded detail is passed to the bodies of lower level subprograms. These bodies are also constrained to an easily understood amount of detail, with lower detail moved again to yet lower level subprograms.

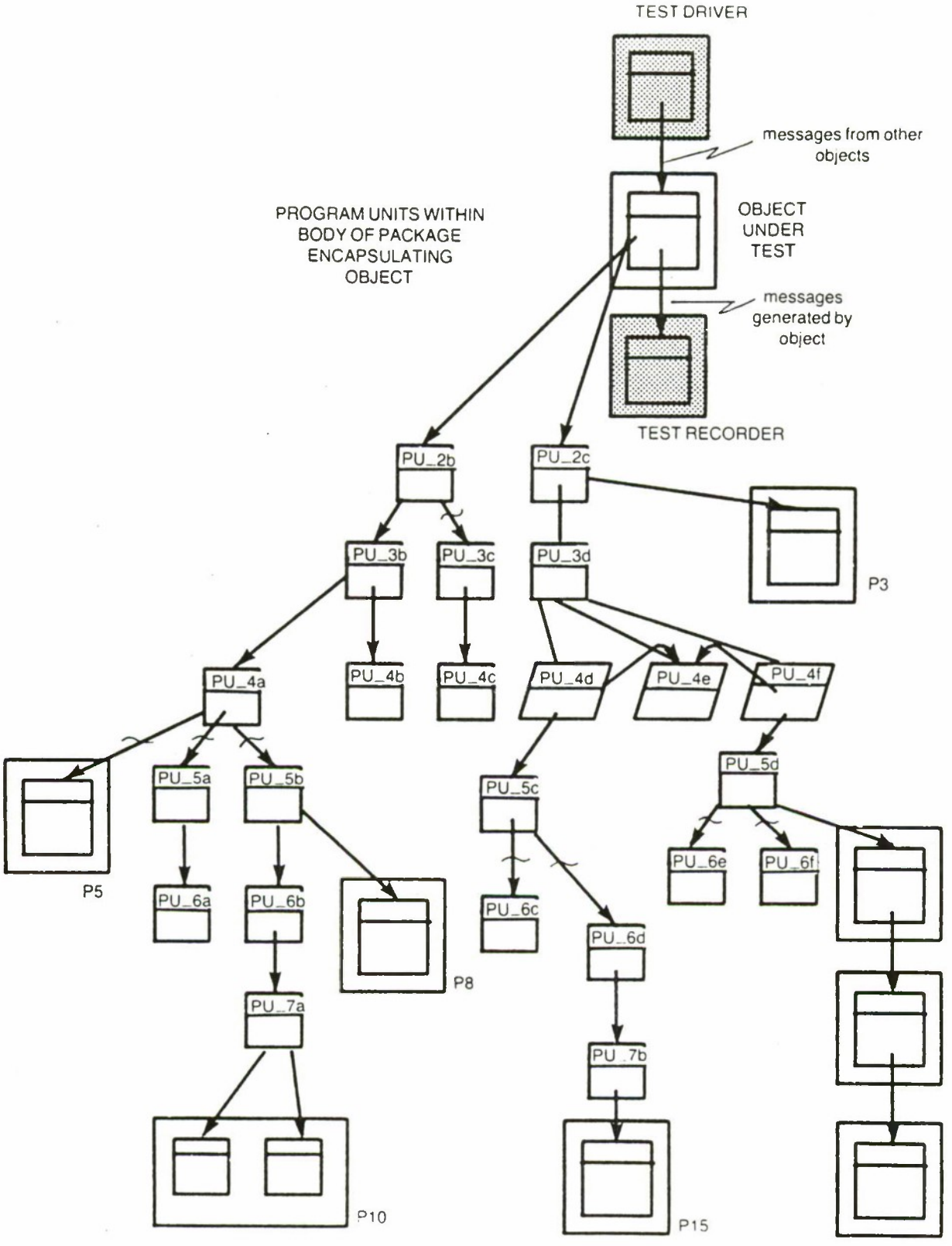


FIGURE 95. REPRESENTING TEST OF A SINGLE OBJECT

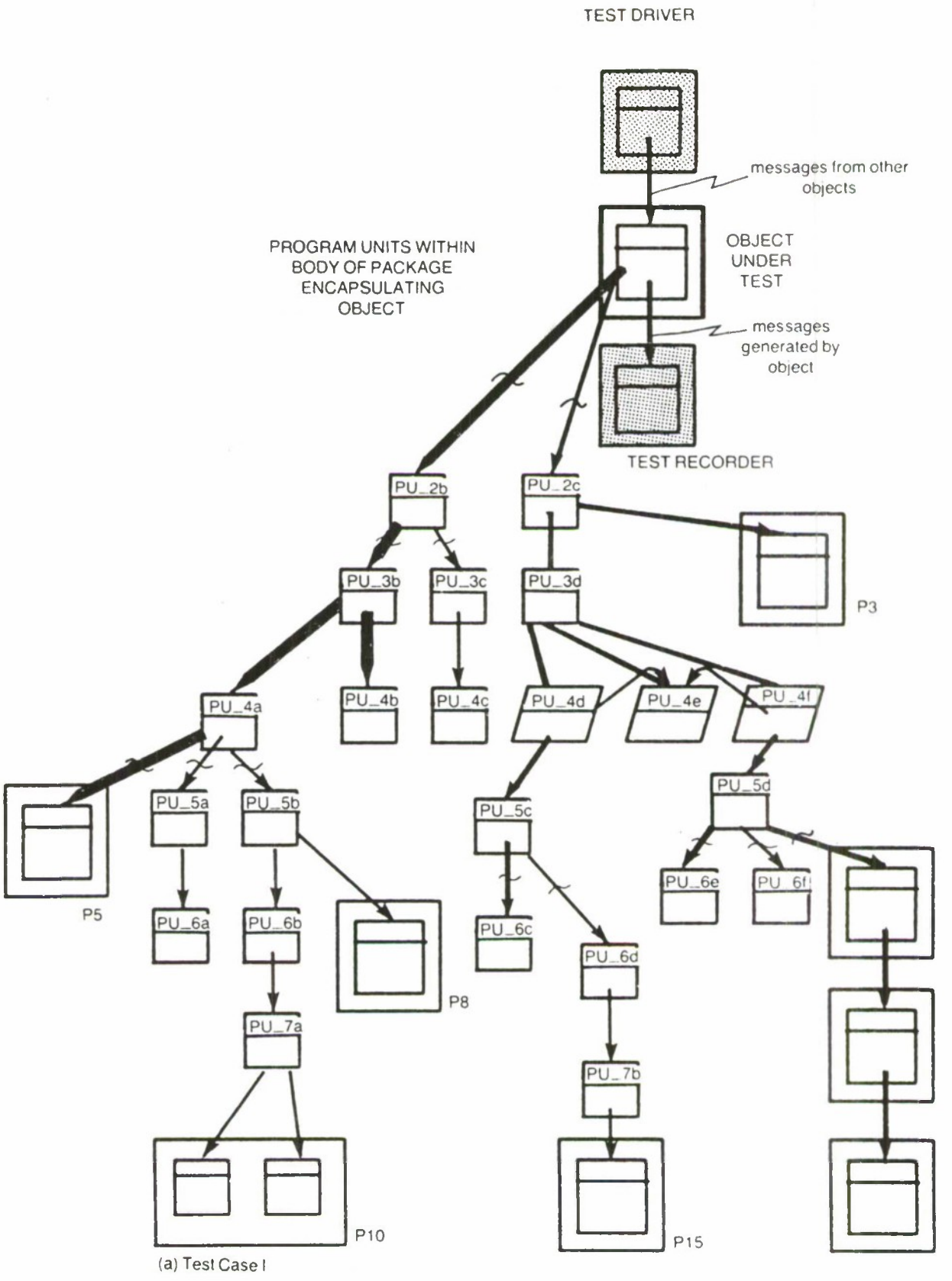


FIGURE 96. TEST CASES FOR TESTING A SINGLE OBJECT

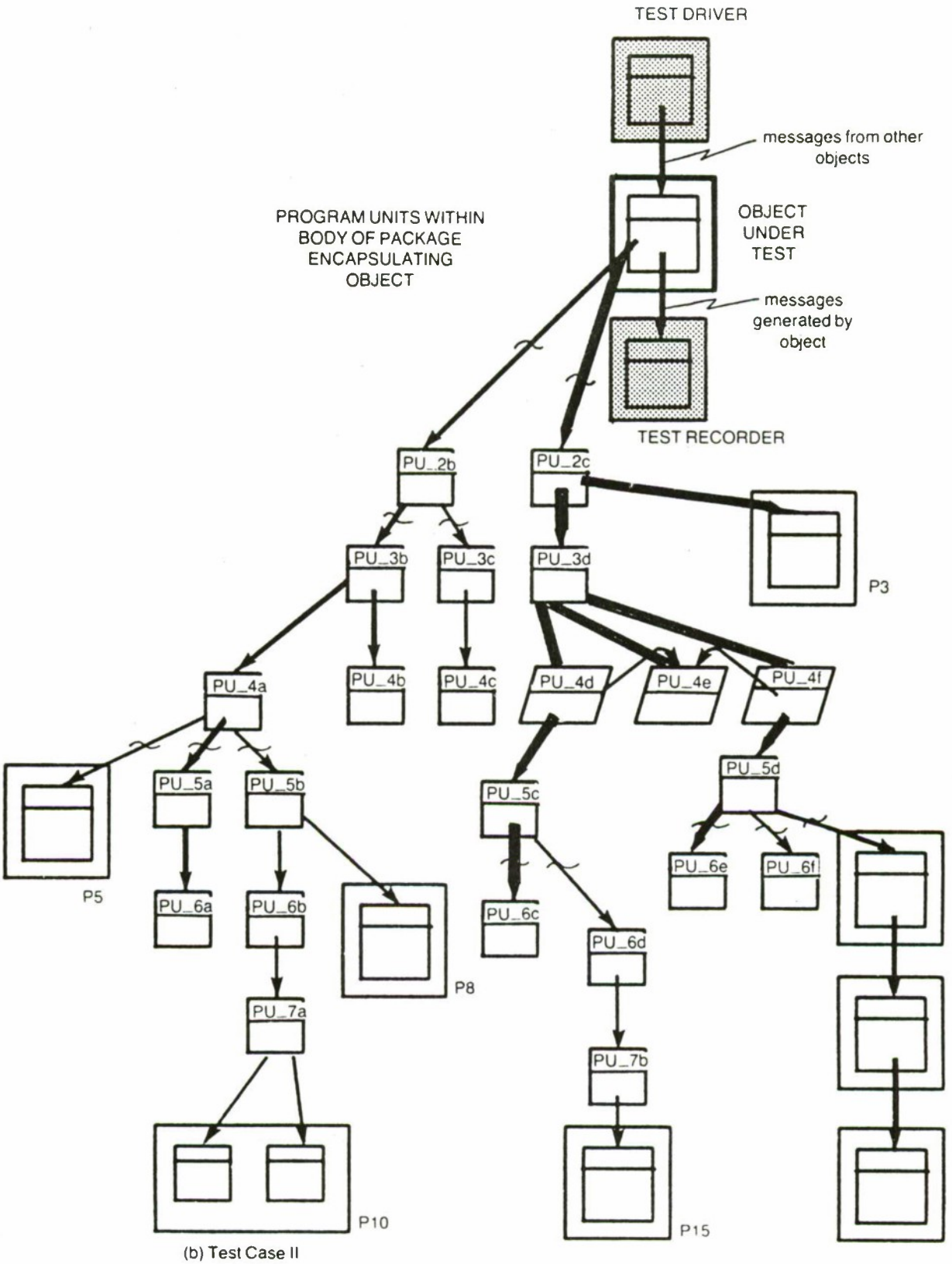


FIGURE 96. (CONCLUDED)

Classical computer programmer testing can be used to test such sequential subprogram interaction. Such testing can be accomplished using a debugger and other test tools provided in an Ada Programming Support Environment (APSE). For example, parameters passed between subprograms can be checked, internal branching within each subprogram can be verified, and data generated in arrays or records can be checked.

The object may also be implemented using multiple tasks in addition to several subprograms. The testing of task performance is not as straightforward as the testing of subprogram performance, due to concurrency associated with task execution. Although classical techniques using an APSE debugger can be applied to assess task performance to a certain extent, other time dependent tests typically will have to be conducted to check the temporal aspects of task execution.

To establish time dependent tests, the environmental simulator package can be used to generate a sequence of stimuli in a predetermined manner and over a specific time period. The faulty object implementation can be tested in conjunction with other implementations and the performance of tasks within the faulty implementation can be monitored.

Such monitoring can be accomplished using a test recording package to record the values of parameters received by task entry points during rendezvous. As shown in Figure 97, the recording package can contain a set of subprograms. Calls can be made to the subprograms from the receiving task, as part of the rendezvous accept statement. This is illustrated by the following Ada code for the task rendezvous example shown in Figure 97.

```
task TASK_D;
   entry ENTRY1: (PAR1: in INTEGER;
                  PAR2: in INTEGER)
   entry ENTRY2: (PAR3: in INTEGER)
   entry ENTRY3: (PAR4: in INTEGER;
                  PAR5; in INTEGER;
                  PAR6: in INTEGER)
   . . .
end     TASK_D;

task body TASK_D is

   . . .

   accept ENTRY1 (PAR1: in INTEGER;
                  PAR2: in INTEGER) do
   RECORD_ENTRY1 (PAR1, PAR2); -- Call subprogram to record
        -- values of the parameters received by entry point ENTRY1
        -- and the time they were received
   . . .

   end ENTRY1;
```

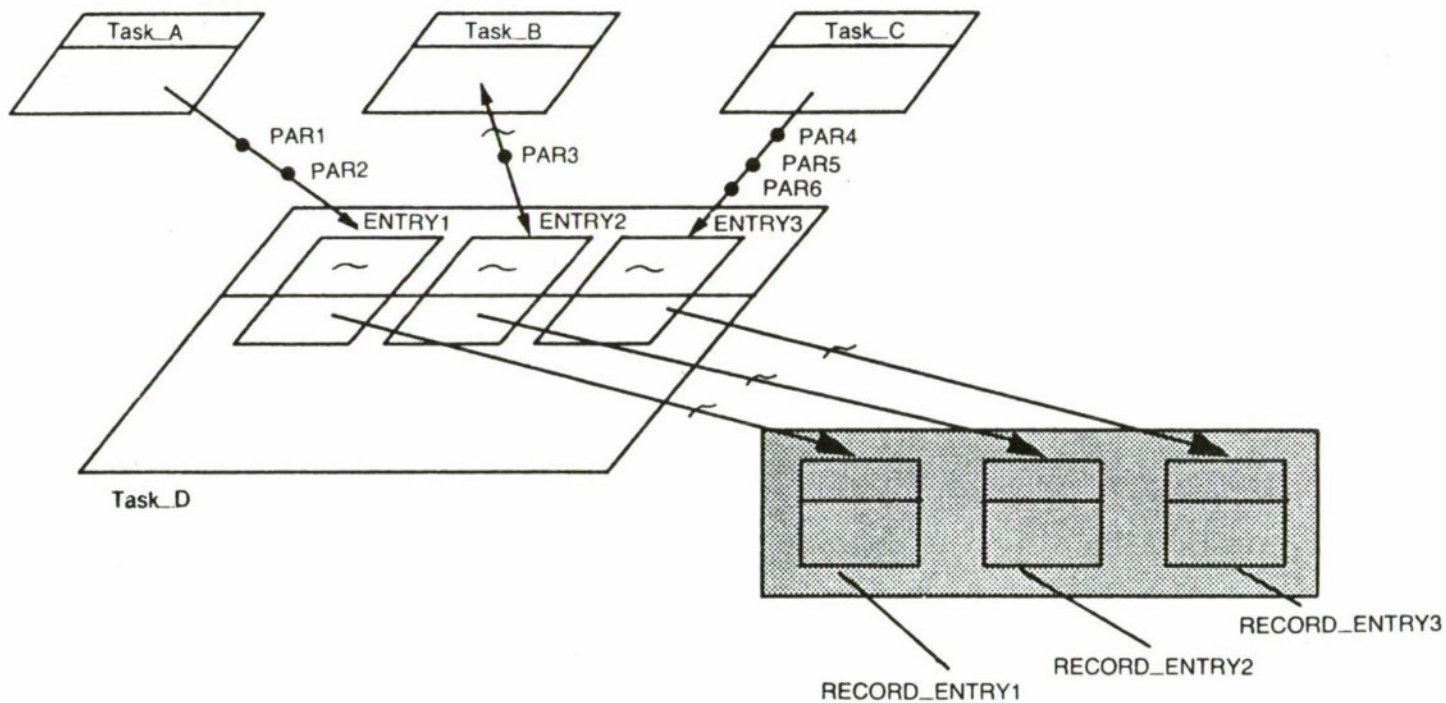


FIGURE 97. TEST OF TASKS ENCOMPASSED BY AN OBJECT

```
accept  ENTRY2 (PAR3: in out INTEGER) do

RECORD_ENTRY2 (PAR3); -- Call subprogram to record
     -- values of the parameters received by entry point ENTRY2
     -- and the time they were received
. . .

end ENTRY2;

accept ENTRY3 (PAR4: in INTEGER;
               PAR5: in INTEGER;
               PAR6: in INTEGER) do

RECORD_ENTRY3 (PAR4, PAR5, PAR6); -- Call subprogram to record
     -- values of the parameters received by entry point ENTRY2
     -- and the time they were received
. . .

end ENTRY3;
     o
     o
     o

end TASK_D;
```

Using this recording capability, a record of parameter values passed during task rendezvous and the time of task rendezvous can be established. The record can then be assessed to check the validity of parameter values passed between tasks, and if they are passed in a timely manner, in compliance with mission requirements.

As with other test data recording, the calls to recording subprograms can be made conditional. In this way, the tester can establish the conditions needed to provide test recording needed for the particular test case being run.

3. CHAPTER SUMMARY

Ada computer programs that have been designed in an object-oriented manner can be systematically tested. Tests can be conducted at a high level to exercise interacting object implementations used to construct the computer program.

If high level tests reveal poor performance, lower level tests can be conducted on individual object implementations to identify faulty implementations, and yet lower level tests can be conducted to isolate the problem in the internal working of a faulty implementation. Changes introduced to correct the faulty implementation will not affect other object implementations that have been performing correctly, because of the localized nature of operations and data bases unique to each object.

To provide a framework for such tests, special purpose Ada test packages must be introduced to generate environmental stimuli and record parameters passed between program units. For a particular test case, the specific program units exercised, as well as the special test packages introduced, can be clearly indicated using SHARP abstracts. These explicit pictorial representations of testing significantly help in comprehending the scope of the tests being undertaken. As such, they will be very helpful to government reviewers of contractor testing activities.

CHAPTER VII

Estimating the Cost of Object-Oriented Ada Software

Existing cost models have been formulated for software designed in a traditional manner, and therefore cannot be directly used to estimate the cost of Ada software developed in an object-oriented manner. In this chapter, an algorithm is established to directly project the cost of object-oriented Ada-unique software. Cost factors within the algorithm are projected using the Constructive Cost Model (COCOMO) but in a special manner.

In addition, algorithms are established to project the size of the implementation of an object-oriented Ada design. Factors in the size projection algorithm can be established by inspection of an object-oriented Ada-unique design represented by SHARP abstracts. This is important since the size of a program is a key input to cost/schedule estimation models. The models can provide meaningful results only if the size metric is accurately estimated.

The algorithms are applied to project the costs to implement a hypothetical embedded computer program using Ada in an object-oriented manner. The cost to implement the same computer program using FORTRAN and assembly language is projected using COCOMO directly. The results indicate that when a large and complex computer program is developed in an object-oriented manner using Ada, significant cost savings can be expected relative to traditional development approaches.

1. INTRODUCTION

1.1 BACKGROUND

1.1.1 Cost Savings Expected Due To Ada Standardization

In order to promote standardization, DoD has mandated the use of Ada in the implementation of mission-critical software. Standardizing to a single high-order language will contribute to lower software life-cycle costs. For example, the use of many different languages necessitates the development and maintenance of several different compilers and programming support tools. With Ada as the standard computer programming language for DoD mission critical systems, fewer compilers will have to be developed.

DoD has initiated an Ada compiler validation program to help ensure that all features of the standard language are being correctly implemented by various compilers. In time, proponents of Ada expect that a variety of Ada programming support tools (e.g., editors and debuggers) will also become available. DoD has taken steps to help ensure that tools that are not architecture dependent will have general applicability to different computing configurations.

In the long run, it is expected that Ada standardization will lead to labor force savings since a large number of personnel will have on-the-job experience using the Ada language and associated programming support tools. These capabilities are expected to be essentially portable from one project to the next, significantly reducing programmer training costs. The experience base of the DoD programming community will not be subdivided among several languages.

1.1.2 Cost Savings Expected Due to Ada Technical Features

In addition to savings associated with standardization, proponents of Ada expect that its technical features will also help reduce software development costs. Several of these technical features, and their graphical representation with SHARP, are described in Chapter II. For example, in Section 2.2.1 of Chapter II, the use of Ada packages to encapsulate reusable software (and represent them in an Ada Package Catalog Diagram) is described. It is envisioned that software contractors will construct and apply in practice a library of Ada packages for such things as hardware drivers, communication protocols, high and low level I/O, math functions and special purpose algorithms.

As another example, in Section 2.4.2 of Chapter II, the SHARP representation of Ada generics is described. With Ada generics, an existing Ada package and its contents become more general purpose. For example, the name of a generic subprogram, and typically the definition of its types and the range of permissible values for passed parameters, are created during compilation. (The process of creating a particular instance of the generic program unit is referred to as generic instantiation.)

1.1.3 Cost Savings Expected Due to Object-Oriented Software Development

Possibly the most important technical features of Ada are its ability to facilitate data abstraction and information hiding. These factors can be used in the implementation of object-oriented designs, which provide a means for controlling dependency relationships between variables, types and program units.

The control of complex dependency relationships between variables, types and program units with an object-oriented design is discussed in Section 2.1 of Chapter III in the context of software maintainability and in Section 1.2 of Chapter V in the context of software testing.

The object-oriented approach is also critical to the development of Ada software because of the complexities of Ada compilers, which make extensive checks of the dependency relationships. Ada compilers are slow relative to compilers for older languages (e.g., FORTRAN). With Ada, implementation of objects are encapsulated in loosely coupled Ada packages and tasks, which the development team can code and test independently. By constraining the size of these object program units and stubbing program units interacting with them, the Ada packages and tasks can be separately compiled and recompiled in a timely manner during their development.

1.1.4 Accounting for Ada Savings in Cost/Schedule Estimation Models

In order to estimate the cost of Ada software developed using traditional design approaches, an analyst can make use of existing models. They have been calibrated for software implementations using the traditional approaches, and account for the high costs incurred in part because of the complex dependency relationships inherent in the traditional approach to designing and implementing software.

However, the existing models cannot be directly used in the estimation of software to be designed and implemented in an object-oriented Ada-unique manner. Rather, a new model is needed that will take into account the cost savings introduced due to control of the complex dependency relationships.

In addition, regardless of which model is used, the size of the program must be accurately established, whether measured in terms of the number of source statements to be delivered or in some other way (e.g., the number of function points). Most models do not provide a mechanism for accurately projecting the size of a computer program. However, by using SHARP abstracts, such projections can be rapidly and accurately made.

1.2 CHAPTER SCOPE

In this chapter, we present a new model for projecting the cost to develop object-oriented Ada software. In conjunction with this model, the estimation algorithms of COCOMO are used in special ways to estimate cost factors in the model. Section 2 describes algorithms associated with Basic, Intermediate and Detailed COCOMO. Section 3 provides an algorithm that projects the cost of object-oriented Ada software directly accounting for the inherent cost advantages associated with this software. It provides algorithms that can be used to project the size of the software and discusses establishing other model inputs in a manner unique to Ada. Section 4 provides an example of applying these the algorithms in practice.

2. OVERVIEW OF THE CONSTRUCTIVE COST MODEL (COCOMO)

2.1 INTRODUCTION

The Constructive Cost Model (COCOMO) is an empirical model used to predict the cost of software development efforts. Data collected from sixty-three projects at TRW, Inc. was used to formulate and calibrate COCOMO. This section provides an overview of COCOMO. For a detailed description of the model and its derivation, see Barry Boehm's book, Software Engineering Economics.[7]

2.1.1 Versions of COCOMO

Three different versions of the COCOMO model have been formulated. They are referred to as Basic, Intermediate and Detailed COCOMO. Each version predicts the cost of the software effort in units of manmonths, but with different degrees of accuracy. Basic COCOMO predicts software development cost as a function of the expected size of the software product measured in

source instructions. It is meant to provide a rough order of magnitude estimates. When applied to the TRW, Inc. data base, it estimates software development effort within a factor of 1.3, 29% of the time; and within a factor of 2, 60% of the time.

Intermediate COCOMO predicts software cost as a function of attributes of the product, computer, personnel and project, as well as the number of source instructions to be developed. It can be used to estimate the cost of the total software package or components of the total software package. When estimating on a component by component basis, attributes selected can vary from one component to another as appropriate. With respect to the TRW data base, Intermediate COCOMO estimates are within 20% of actual project costs 68% of the time.

Detailed COCOMO extends the Intermediate model by taking into account life-cycle phase dependencies of cost drivers.

2.1.2 Modes of Software Development

All three versions of COCOMO distinguish between three modes of software development, differing in scope and intrinsic difficulty. The modes are as follows:

- Organic mode (small to medium size in-house projects undertaken by persons familiar with the application and experienced in developing software for related applications)
- Embedded mode (projects developing a strongly coupled complex of hardware and software that is difficult to change or fix, consists of new architecture, and is tightly constrained by reliability, memory and speed of execution)
- Semi-detached mode (projects halfway between a familiar, in-house organic project and an unfamiliar, innovative embedded project).

2.1.3 Phases of Software Development

The phases of software development covered by each version of COCOMO encompass:

- Product design specification
- Detailed design specification
- Code and unit test
- Integration and test

A separate formulation is provided to estimate the costs of maintenance work. The specific activities accounted for in the model are shown in the work breakdown structure shown in Figure 98.

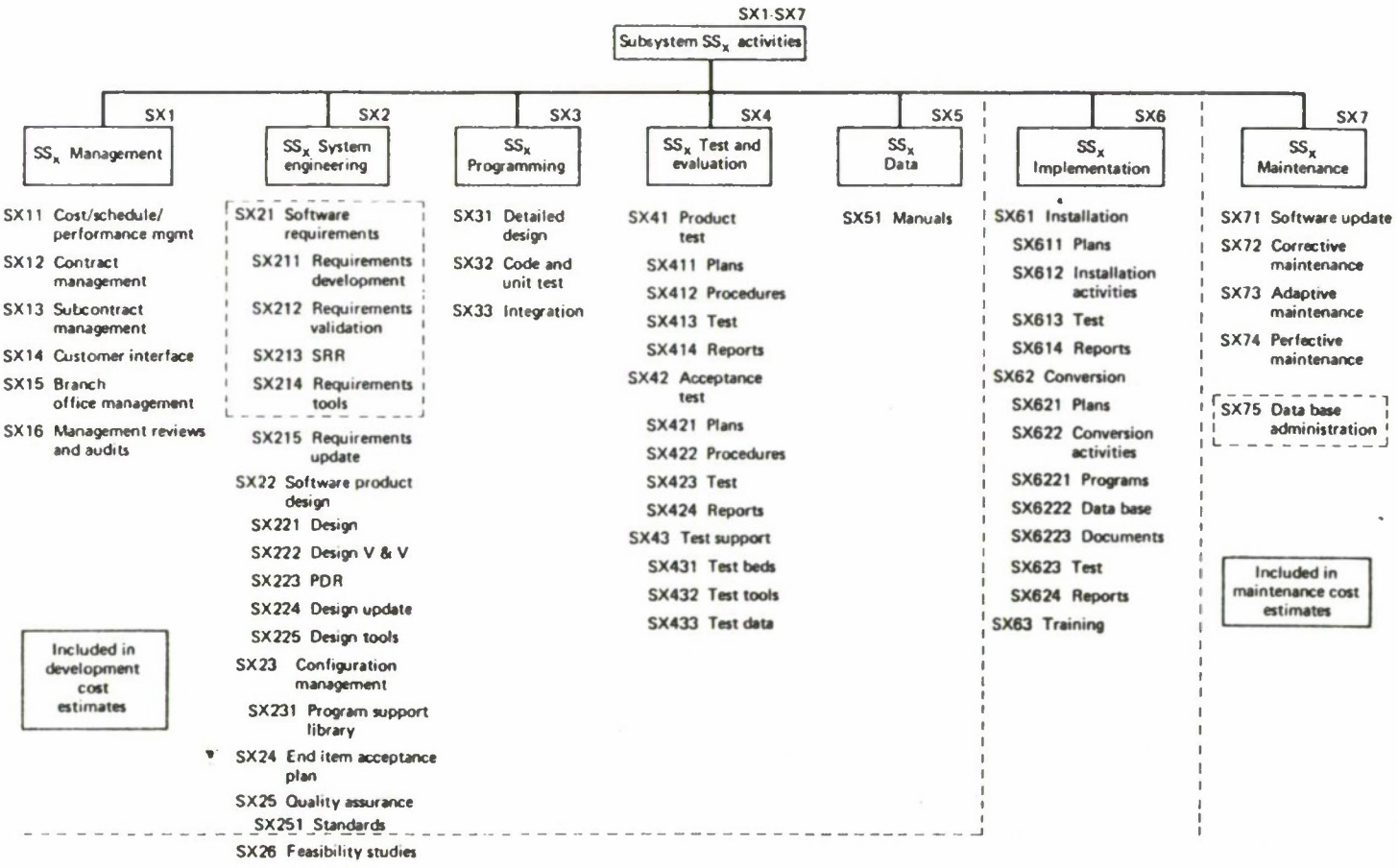


FIGURE 98. SOFTWARE-RELATED ACTIVITIES ACCOUNTED FOR BY COCOMO

This diagram is Figure 4-6(b) in Software Engineering Economics [1]

2.1.4 Definitions and Assumptions of COCOMO

COCOMO defines delivered source instructions to include all source instructions translated by a compiler into machine code, excluding comment lines and lines of unmodified library software. Job control language, format statements and data declarations are included in the instruction count. Support software (e.g., test drivers) is excluded, unless it is to be developed with the same care and documentation as the deliverable product.

Basic assumptions made by COCOMO include the following:

1) A manmonth consists of 152 hours of working time.

2) Project management undertaken by both developer and customer is good.

3) The requirements specification is not substantially changed after the plans and requirements life-cycle phases.

2.2 BASIC COCOMO

2.2.1 Projecting Development Costs with Basic COCOMO

Basic COCOMO uses the following algorithm to estimate the effort to develop software in units of man-months:

$$MM = K(DSI/1000)^{**}E \qquad (2\text{-}1)$$

where values of the coefficient K and the exponent E are given in Table 2. As indicated in this table, a unique value for K and E are selected as a function of the software development mode (i.e., organic, semi-detached or embedded).

TABLE 2 COEFFICIENTS AND EXPONENTS OF THE BASIC COCOMO ESTIMATION ALGORITHM

DEVELOPMENT MODE	ESTIMATING DEVELOPMENT COST	
	K	E
Organic	2.4	1.05
Semi-Detached	3.0	1.12
Embedded	3.6	1.20

2.2.2 Projecting Software Maintenance Costs with Basic COCOMO

COCOMO assumes that the developed software product has been tested to the point where only residual software bugs remain, bugs that were not revealed by software testing undertaken during development. Once this software is delivered to a user for site operation, maintenance of it is necessary. By maintenance, COCOMO means the process of removing residual bugs while

leaving the primary aspects of the software intact. Therefore, maintenance encompasses needed modification of the software product's code, documentation and data base structure associated with the software "repair." COCOMO also includes in maintenance the update and redesign of small portions of the software product.

The cost to perform this maintenance is estimated as a function of the development estimate; and expected additions and projected modifications, which are referred to as the Annual Change Traffic. The Annual Change Traffic (ACT) is the percentage of the software product's source instructions that are projected to undergo change during a (typical) year. Basic COCOMO estimates annual maintenance effort using the following equation:

$$MM(MAINT) = MM(DEV)*ACT \qquad (2\text{-}2)$$

where MM(DEV) is the development effort estimate in manmonths, established using Equation 2-1.

2.3 INTERMEDIATE COCOMO

Intermediate COCOMO estimates software development cost as a function of the size of the product and the development mode, as does the basic model. However, the Intermediate COCOMO estimate also accounts for attributes characterizing the development effort.

2.3.1 Projecting Development Costs with Intermediate COCOMO

The estimation algorithm of intermediate COCOMO is as follows:

$$MM = (C_1 * C_2 * \ldots C_{15})K(DSI/1000)**E \qquad (2\text{-}3)$$

where the coefficient K and the exponent E vary as a function of the development mode, as shown in Table 3. The coefficients C_n are functions of attributes of the development effort:

$$C_n = f(A_n) \qquad (2\text{-}4)$$

The first three attributes account for characteristics of the software product. They are:

A_1 = required software reliability

A_2 = data base size

A_3 = product complexity

The next four attributes account for the computer used. They are:

A_4 = execution time constraint

A_5 = main storage constraint

A_6 = virtual machine volatility

A_7 = computer turnaround time

The next five attributes account for the development personnel. They are:

A_8 - analyst capability

A_9 - applications experience

A_{10}- programmer capability

A_{11}- virtual machine experience

A_{12}- programming language experience

The final three attributes are as follows:

A_{13}- use of modern programming practices

A_{14}- level of tool support

A_{15}- schedule constraint

TABLE 3 COEFFICIENTS AND EXPONENTS OF THE INTERMEDIATE COCOMO ESTIMATION ALGORITHM

DEVELOPMENT MODE	ESTIMATING DEVELOPMENT EFFORT K	E
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

COCOMO requires the estimator to rate each attribute (e.g., very high, high, nominal, low or very low) and provides a value $f(A_n)$ as a function of the rating. The meaning of each attribute rating and the associated values for $f(A_n)$ are provided in Appendix D.

2.3.2 Projecting Software Maintenance Costs with Intermediate COCOMO

Intermediate COCOMO derives software maintenance costs from software development costs using Equation (2-2), in the same manner as Basic COCOMO. However, the coefficient C_{15} is set to unity since it accounts for development schedule constraints, which are not applicable. Also, C_1, which accounts for software reliability and C_{13}, which accounts for the use of modern software practices, are given different values than those applicable to development. These differences are explained in Appendix D.

2.3.3 Intermediate COCOMO and Component Estimation

Intermediate COCOMO can be applied to modules of a software package as well as the entire package itself. Equation (2-3) applies to modules equal to or greater than 2000 source statements.

For smaller modules, the following steps must be taken to estimate their costs:

a. Calculate nominal effort for the whole product using the following algorithm:

$$MM_N = K(DSI/1000)**E \qquad (2\text{-}5)$$

b. Calculate nominal productivity as follows:

$$\text{NOM PROD} = DSI/MM_N \qquad (2\text{-}6)$$

c. Divide the nth module's size by nominal productivity yielding the nominal component effort estimate.

$$MM_N(n) = DSI(n)/\text{NOM PROD} \qquad (2\text{-}7)$$

d. For each module, establish ratings for each of the 15 development attributes; then establish 15 coefficients unique to each component using Equation 2-4 and Appendix D.

e. For the nth module, refine its nominal estimate by applying the appropriate cost driver factors as follows:

$$MM(n) = C_1(n)*C_2(n)*\ldots C_{15}(n)*MM_N(n) \qquad (2\text{-}8)$$

2.4 DETAILED COCOMO

In practice, software development factors (e.g., required reliability, applications experience and interactive software development) affect some phases more than others. Detailed COCOMO provides a mechanism for taking into account these phenomena.

For a computer program that has been partitioned into n modules, the development cost for the nth module is calculated using the following relationship:

$$MM(n) = \sum_{p=1}^{P} C_1(n,p)*C_2(n,p)*\ldots * C_{15}(n,p) * MM_N(n,p) \qquad (2\text{-}9)$$

where

$$MM_N(n,p) = MM_N(n) * P(p)/100 \qquad (2\text{-}10)$$

$MM_N(n)$ is the nominal development cost of the nth module, calculated using Equation 2-7 with DSI(n) set to the number of delivered source instructions projected for the nth module. P(p) is the percentage of the development effort associated with the product design phase (i.e., p=1), the detailed design phase (i.e., p=2), the code and unit test phase (i.e., p=3) or the integration and test phase (i.e., p=4). The values of P(p) are given in Table 4.

The coefficients $C_i(n,p)$ (i=1,2,...15) are selected to be unique to the nth module and the pth phase.

TABLE 4 VALUES OF P(p) AS A PERCENTAGE

MODE	p	EFFORT	VALUES OF P(p 2 KDSI	8 KDSI	32 KDSI	128 KDSI
Organic	1	Product design	16	16	16	16
	2	Detailed design	26	25	24	23
	3	Code and unit test	42	40	38	36
	4	Integration and test	16	19	22	25
Semidetached	1	Product design	17	17	17	17
	2	Detailed design	27	26	26	24
	3	Code and unit test	37	35	33	31
	4	Integration and test	19	22	25	28
Embedded	1	Product design	18	18	18	18
	2	Detailed design	28	27	26	25
	3	Code and unit test	32	30	28	26
	4	Integration and test	22	25	28	31

3. COCOMO AND THE COST/SCHEDULE ESTIMATION OF OBJECT-ORIENTED ADA SOFTWARE

3.1 INTRODUCTION

This section describes the application of COCOMO in the estimation of the cost of object-oriented Ada software. Section 3.2 introduces a special technique for estimating the cost of object-oriented Ada software. This technique establishes an estimation algorithm unique to object-oriented Ada computer programs.

As do other software estimation models, the algorithm unique to object-oriented software operates on a size metric and attributes of the Ada-unique implementation of the design. Typically, software cost estimation models describe how to establish implementation attributes but do not address the more difficult issue of how to project the number of source statements. Our discussion covers both topics. Specifically, the algorithms for projecting the number of source statements needed to implement an object-oriented design are introduced in Section 3.2.2, and a discussion of attribute selection is provided in Section 3.2.3.

3.2 ESTIMATING THE COST OF OBJECT-ORIENTED ADA SOFTWARE

COCOMO has been formulated and calibrated using data measuring the development of computer programs that have not been designed in an object-oriented manner. For such software, complex dependency relationships typically exist between types, variables, and program units. Accordingly, development costs for such software has increased exponentially with computer program size.

As described in earlier chapters, an object-oriented design is introduced to minimize complex dependency relationships. Therefore, COCOMO and other existing software cost/schedule estimation models cannot be directly applied to the estimation of software to be developed in an object-oriented manner. However, it is still possible to estimate the costs of object-oriented Ada software using algorithms that take advantage of COCOMO in the manner outlined in this section.

To establish these estimates, an estimation algorithm unique to software implemented in an object-oriented manner is formulated in Section 3.2.1. The algorithm operates on the number of source instructions required to implement objects, which can be estimated using the techniques presented in Section 3.2.2. The algorithm also requires the selection of COCOMO development attributes, which is discussed in Section 3.2.3.

3.2.1 Algorithm Unique to Estimating Object-Oriented Ada Software Development Costs

We can state that the cost to develop object-oriented Ada software is given by the following relationship:

$$\{\text{COST}\} = \{\text{DESIGN COST}\} + \{\text{OBJECT IMPLEMENTATION COST}\} + \{\text{OBJECT IMPLEMENTATION INTEGRATION COST}\} \quad (3\text{-}1)$$

Assuming that the cost of a traditional design is essentially equivalent to that of an object-oriented design, design costs can be estimated using COCOMO as discussed in Paragraph 3.2.1a. In practice, it should not be either significantly harder or easier to design object-oriented software than software designed using other approaches (e.g., structured top-down).

Since objects are essentially independent computer programs that are loosely coupled, the implementation of each object can be assumed to be an independent effort. These implementation costs can be estimated using COCOMO, as discussed in Paragraph 3.2.1b.

The integration of these objects is not directly accounted for by COCOMO. However, as a lower limit, this effort can be assumed to be essentially equivalent to that of integrating program units, since loosely coupled object interfaces ideally should not be more complicated than typical program unit interfaces. Making this assumption, a lower limit on the cost of object integration can be estimated, as discussed in Paragraph 3.2.1c.

3.2.1a Establishing Design Cost of Object-Oriented Ada Software

The design costs for object-oriented Ada software can be assumed equivalent to the costs incurred with other design methodologies. In practice, there is nothing significantly more or less difficult about mapping software requirements into loosely coupled objects or high coupled modules (e.g., modules associated with a top-down design).

Accordingly, COCOMO can be applied to establish object-oriented design costs using the following relationship:

$$\{\text{DESIGN COST}\} = (\%\ \text{DESIGN}/100)*\text{MM(TOTAL)} \qquad (3\text{-}2)$$

where MM(TOTAL) is the COCOMO software development estimate established using Equation 2-3 (for Intermediate COCOMO) or Equation 2-9 (for Detailed COCOMO) applied to the total number of source statements to be developed for all objects; and % DESIGN is given by the following:

$$\{\%\ \text{DESIGN}\} = P(1) + P(2) \qquad (3\text{-}3)$$

where P(1) is the percentage of the software development cost associated with product design and P(2) is the percentage associated with detailed design, as given in Table 4. However, it is anticipated that computer aided design (CAD) systems will lower design costs, whether traditional or object-oriented. Since, as COCOMO predicts, design costs account for more than 40% of software development costs, these savings will be significant. We discuss CAD and its ramifications with SHARP in Chapter IX.

3.2.1b Establishing Object Implementation Costs

Each object can be assumed to be developed essentially as an independent computer program. With this assumption, COCOMO can be applied to establish implementation costs using the following relationship:

$$\{\text{OBJECT IMPLEMENTATION COST}\} = P(3)* \sum_{p=1}^{p} \sum_{o=1}^{O(p)} \text{MM}(p,o) \qquad (3\text{-}4)$$

where P(3) is the percentage of the software development cost associated with program unit code and test; and MM(p,o) is the development cost for the oth object of the pth process. The factor P(3) is obtained using Table 4. The factors MM(p,o) are calculated using Intermediate COCOMO applied to each object, as described in Section 2.3, or using Advanced COCOMO applied to each object, as described in Section 2.4.

3.2.1c Establishing Object Implementation Integration Costs

As already stated, COCOMO does not predict object implementation integration costs. In practice, these costs will be relatively low if the objects are loosely coupled and will increase as the degree of coupling increases. The cost to develop strongly coupled objects will approach the cost associated with traditional software efforts. This represents an upper limit on the cost of software developed in an object-oriented manner.

As a lower limit, the cost of object integration costs can be estimated using COCOMO by assuming that this effort is essentially equivalent to integrating program units. This is the case for Ada-unique object-oriented designs when objects are loosely coupled. For this situation, the following relationship can be used to establish the lower limit:

$$\{\text{LOWER LIMIT FOR OBJECT IMPLEMENTATION INTEGRATION}\} = P(4)*\text{MM(EQUIV)} \qquad (3\text{-}5)$$

where P(4) is the percentage of software development cost associated with program unit integration, from Table 4; and MM(EQUIV) is the development cost for a number of program units equal to the number of objects to be developed (for all processes). The later factor is calculated again using Equation 2-3 (for Intermediate COCOMO) or Equation 2-9 (for Detailed COCOMO), applied to the number of source instructions obtained by multiplying the number of objects by the size of an individual program unit. As discussed in Paragraph 3.4.2(c), program units typically average approximately 49 source statements. Thus, as a lower limit, we assume that the cost to integrate '0' objects is equivalent to the cost to integrate '0' program units of 49 source statements each.

3.2.2 Estimating the Number of Source Statements of Object-Oriented Ada Software

In order to apply COCOMO with any degree of accuracy, a meaningful estimate of the number of Ada source statements is needed. For an object-oriented Ada computer program, this can be effectively accomplished by examining the object-oriented design established using the steps discussed in Section 2.2 of Chapter IV.

Specifically, the main Ada computer program will declare tasks for P processes (Step 1) and each of the P processes will be partitioned into O objects (Step 2). The number of source statements (i.e., #SS) needed to implement the object-oriented Ada design is given by the following:

$$\#SS = \sum_{p=o}^{p=P} \sum_{o=1}^{o=O(p)} \#DSI(p,o) \qquad (3\text{-}6)$$

where #DSI(p,o) is the number of source statements for the oth object in the pth process. The index p equal to zero refers to subprogram declared in the body of the main program. With Ada, the execution of these subprograms is undertaken until completion concurrently with processes established by tasks declared in the main program (in the time slice sense of the word). They, therefore, act for a period of time like a defacto process. Accordingly, we consider them the process associated with p equal to zero.

The remainder of this subsection discusses three different methods that can be used to estimate #DSI(p,o). Method 1 utilizes algebraic relationships, which establish a count of the typical number of Ada statements needed to implement code represented by the graphics. Method 2 is based upon analogies to existing systems. Method 3 is based upon the "Theory of Sevens." Method 3, is the easiest to apply in practice, but the least accurate. Method 2 would be the most accurate, if SHARP were applied to several acquisitions in conjunction with a cost/metric data collection system so that a meaningful data base of design, cost and metric information became available. With such historical information, meaningful analogies could be made.

3.2.2a Method 1 - Algebraic Count Relationships

The implementation of an object normally is encapsulated in an Ada package or task, and has a local state unique to it defined in a data structure

established in the package or task. An object package declares one or more procedures in its specification to facilitate inter-object communication. The bodies of the communication program units are implemented using a traditional structured/top-down design, with detail abstracted into levels. Each level contains one or more program units. Therefore, the number of delivered source instructions needed to implement the oth object of the pth process is given by the relationship:

$$\#DSI(p,o) = \#DATA(p,o) + \#PU(p,o) + \#OPERS(p,o) \qquad (3\text{-}7)$$

where for the oth object in the pth process, #DATA(p,o) is the number of source statements needed to establish the data structure; #PU(p,o) is the number of source statements needed to implement program unit calls and #OPERS(p,o) is the number of source statements to implement program unit bodies, exclusive of the data structure. The factor #DATA(p,o) is given by:

$$\#DATA(p,o) = \#TYPES(p,o) + \#CONSTS(p,o) + \#VARS(p,o) \qquad (3\text{-}8)$$

where for the oth object in the pth process, #TYPES(p,o) is the number of source statements used to establish type definitions; #CONSTS(p,o) is the number of source statements used to define constants; and #VAR(p,o) is the number of source statements used to establish variables for the oth object of the pth process.

The factor #PU(p,o) is given by:

$$\#PU(p,o) = \sum_{l=1}^{L(p,o)} \#SUBP(p,o,l) + 2*\#WITHS(p,o,l) + \#TASKS(p,o,l) \qquad (3\text{-}9)$$

where for the lth level of the oth object in the pth process, #SUBP(p,o,l) indicates the number of subprograms used, #WITHS(p,o,l) indicates the number of Ada "with clauses" introduced, and #TASKS(p,o,l) indicates the number of tasks introduced. The factor "2" in Equation (3-9) is introduced to account for the assumption that Ada "use" clause typically will be applied in conjunction with the "with" clause.

The factor #OPERS(p,o) in Equation 3-7 indicates the number of source statements needed to implement processing in the oth object of the pth process. It is derived from SHARP Annotated Pseudo Code (i.e., Step 7). For example, for each program unit within the lth level of abstraction, the number of decisions can be counted, and the number of instructions needed to implement algorithms, generic instantiation and exception handling can be estimated by inspection of the annotated pseudo code. As an alternative, the 'Theory of Sevens' could be applied to help project the size of each program unit, as explained in Paragraph 3.2.2c.

This method of source instruction estimation requires a competent designer who can map the software requirements into an object-oriented Ada design, for example, using an automated CAD system for establishing SHARP graphics. Then, by examination of the SHARP abstracts (either manually or automatically), the factors associated with Equations 3-7 to 3-9 can be established.

3.2.2b Method 2 - Analogies to Existing Software

We could also estimate the number of deliverable source instructions by comparing System A to System B. In practice these systems are never equivalent (or why would one need to build the new system). Accordingly, parts that are similar need to be identified and isolated from parts that are different. With SHARP abstracts, this can be effectively accomplished by comparing the SHARP representation for System A to the SHARP representation for System B. For example, the number of objects used in each system can be compared along with the number of program units used to implement each object.

The number of source instructions associated with similar objects can be established, while new (or significantly different versions of existing objects) can be isolated. The number of source statements needed to implement the new software can then be established using Method 1.

3.2.2c Theory of Sevens and Projecting the Number of Source statements Used to Implement a Program Unit's Body

The psychologist George Miller performed studies in the 1950s that led him to the conclusion that the number of entities humans can comprehend at one time is seven, plus or minus two.[9] Beyond this limit, a concept is typically too complex for a human to understand.

Booch in his book Software Engineering with Ada calls this the Hrair limit and states the following:

> "Clearly, developing software systems is a problem-solving activity, and so the Hrair limit seems to apply. We suggest that the principles of software engineering can help us decompose systems so that, at each level of the solution, the number of entities we must deal with at one time lies within the Hrair limit."

Assuming this to be true, we can establish bounds on the number of source instructions that typically will be used by programmers to implement the bodies of Ada subprograms and tasks. As discussed in Chapter II, a designer will use abstraction in the design of a program unit's body. A small and easily comprehended number of source statements will typically be needed to implement the body of each Ada program unit, since lower level detail is passed to other called program units. In establishing and implementing the body of each program unit, we might hypothesize that the comprehension of programmers will be limited to 7 ± 2 instructions per Ada block of code, and 7 ± 2 blocks of code per Ada program unit. This results in the boundaries on the number of Ada source statements a programmer will use the implementation of Ada blocks, and program units as follows:

Ada Block:	5 (low)	7 (medium)	9 (high)
Ada Program Unit:	25 (low)	49 (medium)	81 (high)

These factors could be used to form the basis for estimating the number of Ada source statements that are expected to be used to implement the bodies of Ada subprograms and tasks.

The estimator could rate the size and complexity of each program unit as either very low, low, nominal, high or very high. Then, based upon the Theory of Sevens, project the program unit's size using the source statement counts shown in Table 5.

TABLE 5 SOURCE INSTRUCTIONS FOR THE BODIES OF ADA SUBPROGRAMS AND TASKS BASED UPON THE THEORY OF SEVENS

	Number Ada Source Statements	
Size/Complexity Rating	Ada Block	Body of Ada Subprogram or Task
Very High	--	81
High	9	65
Nominal	7	49
Low	5	37
Very Low	--	25

3.2.3 Selecting Attributes for Object-Oriented Ada Software Development

Once the size of an Ada computer program has been established, other information characterizing the acquisition must be considered. With COCOMO, this information is accounted for in cost-driver attributes. Values selected for many of these attributes are driven by unique Ada factors.

Initially, cost drivers will be affected by the lack of personnel with Ada experience, compiler problems, incomplete Ada Programming Support Environments, uncertainties with the compiler validation process and the inefficiency of the new compilers. In the future, a base of Ada experienced personnel with knowledge of comprehensive and standard support environments will affect the cost drivers. As Ada technology matures, cost drivers for target machine characteristics and tool support level must reflect that maturity.

3.2.3a Product Attributes

The COCOMO attributes characterizing the product to be developed as follows are:

- A_1 (Required Software Reliability)
- A_2 (Data Base Size)
- A_3 (Product Complexity)

Values for attributes A_1 and A_2 are essentially a function of product requirements and not the implementation methodology. The value for attribute A_3 can be selected to reflect the possible simplification of a product's complexity introduced by an object-oriented design, since (a) the design is well structured, (b) complex dependency relationships have been reduced and (c) potential compilation problems have been diminished (i.e., by restricting object implementations to sizes that will compile in a timely manner).

3.2.3b Computer Used

The COCOMO attributes characterizing the computer to be used are as follows:

- A_4 (Execution Time Constraints)
- A_5 (Main Storage Constraints)
- A_6 (Virtual Machine Volatility)
- A_7 (Computer Turnaround Time)

The value for attribute A_4 can be selected to reflect inefficiencies on an Ada compiler, although such inefficiencies will diminish somewhat with time. The value for attribute A_5 can be selected to reflect the verbosity of the Ada language relative to languages used to calibrate COCOMO. The value for attribute A_6 can be selected to reflect high volatility in the short-term, although historic norms should be reached in the long-term. The value for attribute A_7 can be selected to reflect the processing capabilities available relative to the amount of Ada code to be produced. This value can take into account the reduction in compilation problems introduced by proper object-oriented designs, or the magnification of compilation problems introduced by inappropriate design approaches.

3.2.3c Development Personnel

The COCOMO attributes characterizing personnel to undertake the software development effort are as follows:

- A_8 (Analyst Capability)
- A_9 (Application Experience)
- A_{10} (Programmer Capability)
- A_{11} (Virtual Machine Experience)
- A_{12} (Programming Language Experience)

The value for these attributes can be selected to reflect the inherent capabilities of the contractor responsible for the development of the Ada software. It is anticipated that these capabilities will improve with time, as the DoD software community becomes knowledgeable in Ada.

3.2.3d Other Attributes

Other COCOMO attributes are as follows:

- A_{13} (Use of Modern Programming Practices)
- A_{14} (Level of Tool Support)
- A_{15} (Schedule Constraint)

The fact that Ada facilitates several modern programming capabilities should be reflected in the value selected for attribute A_{13}. For example, in addition to providing a mechanism for implementing object-oriented designs, Ada promotes general purpose program units through its generics capabilities, facilitates exception handling, and facilitates abstraction within data structures and the implementation of object complexities.

The high level of tool support provided by Ada Programming Support Environments (APSEs) can be reflected in the value selected for attribute A_{14}. When making this selection, an estimator must make sure that selected APSE tools apply to the target computing system as well as the host. The value selected for attribute A_{15} can take into account the amount of time available relative to the time required to build object-oriented Ada software, which should prove to be substantially less than traditional techniques.

3.3 ESTIMATING THE TIME DURATION FOR SOFTWARE DEVELOPMENT (SCHEDULE)

COCOMO can be used to project the time duration expected for a software development effort. This projection is made as a function of the cost estimate using the following relationships:

$$\begin{aligned} TDUR &= 2.5(MM)^{0.38} && \text{(Organic Mode)} \\ TDUR &= 2.5(MM)^{0.35} && \text{(Semidetached Mode)} \\ TDUR &= 2.5(MM)^{0.32} && \text{(Embedded Mode)} \end{aligned} \qquad (3\text{-}10)$$

In practice, one should estimate, on an object-by-object basis, the time required to complete the various major activities associated with the development of a large and complex Ada computer program, including the development of a non-deliverable support software needed to develop the deliverable computer program. Then, using a PERT diagram (or equivalent), the time duration of the various software development tasks should be interrelated to establish the time duration in months for the overall software development project.

4. EXAMPLE

4.1 INTRODUCTION

Consider the hypothetical situation where the government must make a decision as to (a) what programming language it will specify in the development of a large and complex, embedded computer program; and (b) when the development effort should start.

In order to make this decision, the government contracts with an engineering firm, which is to establish cost estimates for different possible object-oriented Ada efforts. Specifically, projections are to be made for an effort to start in the short-term and for an effort to start at a later date. Also, as a possible alternative, the engineering firm is to estimate the cost to develop the software if a traditional top-down structured design is implemented using a FORTRAN/assembly language combination. Historically, this combination has been applied in about 2/3 of embedded systems acquired at the Electronic Systems Division (ESD) of the Air Force Systems Command. [10]

As a first step in establishing these estimates, the estimator establishes cost driver attributes characterizing the software acquisition for the application of Ada on the short-term and in the long-term, and for the application of the FORTRAN/assembly language combination. The results of this analysis are shown in Section 4.2.

Second, the estimator works with a software designer to project the number of source statements required to implement the object-oriented Ada software and the FORTRAN/assembly language software. The results of this analysis are shown in Section 4.3.

Then, the estimator applies algorithms described in Sections 2 and 3 to establish cost projections.

4.2 ESTABLISHING ATTRIBUTES

The COCOMO attribute inputs selected for each effort by an estimator are shown in Table 6. He set ten of the COCOMO attribute inputs to "nominal." Thus, these attributes were not varied as a function of the programming language to be used in relation to time of the development effort (i.e., short-term or long-term). Rather, they were used to establish a nominal definition of the software product (i.e., its required reliability, data base size and complexity), the processing capability available (i.e., main storage capacity, execution time capacity, and turnaround time), the inherent ability of development and maintenance personnel (i.e., their capability and experience in the application problem domain to be programmed), and the schedule followed in the development of the software product.

The other five COCOMO attributes were assumed to account for cost differences due to the language used in relation to the time the development effort was to be undertaken. The attribute Virtual Machine Volatility accounts for the relative frequency of changes that must be made to maintain the virtual machine, defined to be the hardware and software utilized in conjunction with the execution of the application software (i.e., the computer hardware, operating system, run-time system, and compiler). This factor was set too high (i.e., one week to two months between changes) for the development of Ada software in the short-term, and too low (i.e., one month to twelve months between changes) for development of Ada software in the long-term. In the short-term, the estimator anticipates that the application of Ada will uncover errors in the software of the virtual machine. It has been found in the past that during initial applications of a new language, such software is error prone. Critical bugs have to be fixed and necessary features have to be added. In the long-term, the estimator assumed that most of these problems will have been resolved.

The attributes Virtual Machine Experience and Programming Language Experience were varied in the same manner to account for the general experience of the available work force in the application of Ada. These attributes were set to low (i.e., four months average experience) for development of Ada software in the short-term, and to high (i.e., three years or more average experience) for the development of Ada software in the long-term. Initially, few experienced Ada programmers will be available, but this will not be the case in the long-term.

TABLE 6 NUMERICAL VALUES SELECTED FOR COEFFICIENTS

Category	Input Attribute	Coeff.	Coeff. Values -- Development: Range Ada Short Term	Range Ada Long Term	Fortran & Assembly
Product	Required Software Reliability	C_1	1.00	1.00	1.00
	Data Base Size	C_2	1.00	1.00	1.00
	Product Complexity	C_3	1.00	1.00	1.00
Computer	Execution Time Constraint	C_4	1.00	1.00	1.00
	Main Storage Constraint	C_5	1.00	1.00	1.00
	Virtual Machine Volatility	C_6	1.12	0.87	0.87
	Computer Turnaround Time	C_7	1.00	1.00	1.00
Personnel	Analyst Capability	C_8	1.00	1.00	1.00
	Applications Experience	C_9	1.00	1.00	1.00
	Programmer Capability	C_{10}	1.00	1.00	1.00
	Virtual Machine Experience	C_{11}	1.08	0.90	1.00
	Programming Lang. Experience	C_{12}	1.06	0.95	1.00
Project	Modern Programming Practices	C_{13}	0.89	0.82	1.00
	Level of Tool Support	C_{14}	0.89	0.83	1.00
	Required Development Schedule	C_{15}	1.00	1.00	1.00

The attribute Modern Programming Practices was varied to indicate the extent to which such capabilities of Ada are to be exploited. This attribute was set to high (i.e., above average application) for the development of Ada software in the short-term. This attribute was set to very high (i.e., extensively and efficiently used) for the development of Ada software in the long-term.

The attribute Use of Software Tools was varied to indicate the extent to which Ada programming support tools are applied. This attribute was set to high (i.e., strong use of tools) for the development of Ada software in the short-term. This attribute was set to very high (i.e., heavy use of advanced tools) for the development of Ada software in the long-term. The estimator assumed that programmers will take advantage of Ada programming support. He also assumed that in the short-term, APSEs will encompass a necessary and sufficient set of tools needed to develop Ada software, and in the long-term, a comprehensive set of such tools will be incorporated into APSEs, for both host and target machines.

The numerical values chosen for the coefficients, corresponding to the COCOMO input attributes, are also shown in Table 6. They were established by relating attribute ratings to the corresponding coefficient numerical values defined in Software Engineering Economics, which are shown in Appendix D. The coefficients associated with the ten COCOMO attributes, not varied as a function of the language used in relation to when the effort was undertaken, were all set to unity (i.e., the coefficient numerical value for a nominal attribute setting). Each of the coefficients associated with the other five COCOMO attributes, were assumed to account for cost differences due to the language to be used in relation to the time the development effort was to take place.

4.3 ESTIMATING THE SIZE METRIC

As a prerequisite to establishing an accurate projection for the size metric, a designer establishes an object-oriented design for the embedded software using a new automated SHARP system (e.g., the system described in Chapter IX). The automated system produces an estimate of the size metric. In order to understand this projection, the estimator decides to inspect the set of SHARP abstracts produced to represent the object-oriented Ada design.

First, the estimator reviews the SHARP representation of the main program to identify the number of processes established by the designer. He finds that the requirements for the embedded computer program have been distributed into 10 concurrently executing processes (e.g., in a manner similar to the processes shown in Figure 59 in Chapter IV).

Second, the estimator reviews SHARP object layer diagrams established by the designer for each process. He finds that requirements assigned to each process have been distributed to several objects (e.g., in the manner shown in Figure 60 in Chapter IV). Specifically, the number of objects per process, O(p), are as follows:

O(1)=10	O(6)=12
O(2)=12	O(7)=14
O(3)= 9	O(8)= 9
O(4)= 8	O(9)= 8
O(5)=11	O(10)= 7

Third, the estimator reviews the SHARP data structure diagram prepared for each object (e.g., in the manner shown in Figure 64 in Chapter IV). Using Equation 3-8, he estimates the number of Ada source statements needed to implement the data structures, #DATA(p,o). The results are shown in Table 7. For example, for the first object of the first process, he finds the following:

#TYPES(1,1) = 20
#CONSTS(1,1) = 15
#VARS(1,1) = 55

Therefore, using Equation 3-8:

$$\#DATA(1,1) = \#TYPES(1,1) + \#CONSTS(1,1) + \#VARS(1,1) = \underline{90}$$

Fourth, the estimator reviews SHARP hierarchy diagrams established by the designer to represent the hierarchy of program units used to implement the internal complexities of each object (e.g., in the manner shown in Figure 62 of Chapter IV). Using Equation 3-9, he estimates the number of Ada source statements needed to facilitate the structures shown for oth object in the pth process, #PU(p,o). For example, for the first object of the first process, he finds the following:

#SUBP(1,1,1) = 1	#WITHS(1,1,1) = 0	#TASKS(1,1,1) = 0
#SUBP(1,1,2) = 2	#WITHS(1,1,2) = 1	#TASKS(1,1,2) = 2
#SUBP(1,1,3) = 4	#WITHS(1,1,3) = 0	#TASKS(1,1,3) = 0
#SUBP(1,1,4) = 3	#WITHS(1,1,4) = 2	#TASKS(1,1,4) = 4
#SUBP(1,1,5) = 4	#WITHS(1,1,5) = 2	#TASKS(1,1,4) = 0
#SUBP(1,1,6) = 6	#WITHS(1,1,6) = 0	#TASKS(1,1,6) = 0
#SUBP(1,1,7) = 2	#WITHS(1,1,7) = 2	#TASKS(1,1,7) = 0

Therefore, using Equation 3-9, the following applies to the first object of the first process:

$$\#PU(1,1) = \sum_{l=1}^{7} \{\#SUBP(1,1,l) + 2*\#WITHS(1,1,l) + \#TASKS(1,1,l)\}$$

$$= 1 + 6 + 4 + 11 + 8 + 6 + 6 = \underline{42}$$

TABLE 7

SIZE DATA

p	o	# Source States to Implement: Data Structure #DATA(p,o)	Program Unit Calls #PU(p,o)	Operations/ Logic in Bodies #OPERS(p,o)	Total # Source Statement #DSI(p,o)	
1	1	90	42	1385	1517	
	2	150	30	1250	1430	
	3	75	20	1550	795	
	4	75	20	700	795	
	5	95	25	1020	1140	
	6	107	30	1275	1412	
	7	95	40	1405	1540	
	8	85	38	1500	1623	
	9	95	40	1600	1735	
	10	100	45	1800	1945	13,932
2	1	160	40	1200	1400	
	2	40	10	300	350	
	3	80	20	600	700	
	4	120	30	950	1100	
	5	90	20	620	730	
	6	80	15	475	570	
	7	60	12	380	452	
	8	90	25	750	865	
	9	120	30	800	950	
	10	80	20	500	600	
	11	150	40	1100	1290	
	12	50	10	300	360	9367

TABLE 7

(Continued)

		# Source States to Implement				
p	o	Data Structure #DATA(p,o)	Program Unit Calls #PU(p,o)	Operations/ Logic in Bodies #OPERS(p,o)	Total # Source Statemnts #DSI(p,o)	
3	1	40	12	336	338	
	2	90	38	1330	1458	
	3	80	27	783	890	
	4	70	33	1023	1126	
	5	90	29	870	1019	
	6	90	18	576	684	
	7	130	35	1365	1530	
	8	140	30	1020	1190	
	9	55	32	1184	1271	9506
4	1	75	37	1036	1148	
	2	92	27	891	1010	
	3	110	32	992	1134	
	4	120	29	1073	1222	
	5	80	16	496	592	
	6	70	12	360	442	
	7	60	19	627	706	
	8	50	15	540	605	6,859

TABLE 7

(Continued)

		# Source States to Implement				
p	o	Data Structure #DATA(p,o)	Program Unit Calls #PU(p,o)	Operations/ Logic in Bodies #OPERS(p,o)	Total # Source Statemnts #DSI(p,o)	
5	1	90	35	1085	1215	
	2	102	37	1036	1175	
	3	125	40	1160	1325	
	4	85	31	1023	1139	
	5	132	38	1026	1196	
	6	96	31	775	902	
	7	87	25	800	912	
	8	50	34	1122	1206	
	9	40	16	448	504	
	10	90	27	837	954	
	11	120	29	986	1135	11,663
6	1	90	17	544	651	
	2	80	15	435	530	
	3	95	25	835	955	
	4	115	29	986	1130	
	5	130	32	864	1026	
	6	110	28	840	978	
	7	80	12	432	524	
	8	50	14	434	498	
	9	40	16	528	584	
	10	60	23	713	796	
	11	70	31	961	1062	
	12	50	13	351	414	9,148

TABLE 7

(Continued)

		# Source States to Implement				
p	o	Data Structure #DATA(p,o)	Program Unit Calls #PU(p,o)	Operation/ Logic in Bodies #OPERS(p,o)	Total # Source Statemnts #DSI(p,o)	
7	1	110	31	992	1133	
	2	95	27	783	905	
	3	55	14	504	573	
	4	75	16	528	619	
	5	95	19	551	665	
	6	85	23	736	844	
	7	115	27	842	984	
	8	110	25	754	889	
	9	60	12	371	443	
	10	80	17	522	619	
	11	90	16	514	620	
	12	120	34	1020	1174	
	13	70	18	579	685	
	14	135	37	1147	1319	11,472
8	1	55	15	467	537	
	2	75	18	601	694	
	3	110	24	768	902	
	4	50	17	559	626	
	5	55	19	645	718	
	6	50	23	713	786	
	7	40	16	443	499	
	8	65	31	926	1022	
	9	70	32	878	980	6,765

TABLE 7

(Concluded)

		# Source States to Implement				
p	o	Data Structure #DATA(p,o)	Program Unit Calls #PU(p,o)	Operation/ Logic in Bodies #OPERS(p,o)	Total # Source Statemnts #DSI(p,o)	
9	1	40	12	363	415	
	2	45	17	527	589	
	3	35	21	693	749	
	4	55	19	608	682	
	5	60	13	403	476	
	6	40	10	324	374	
	7	50	15	519	584	
	8	40	13	403	456	4,325
10	1	70	15	452	537	
	2	80	16	480	576	
	3	90	22	682	704	
	4	60	14	420	494	
	5	75	31	868	974	
	6	65	28	840	933	
	7	90	30	960	1080	5,298

The results for all objects are shown in Table 7.

Fifth, the estimator reviews annotated pseudo code prepared to represent the bodies of program units used to implement each object. He establishes an estimate of the number of source statements for each body, knowing through experience the relationship between pseudo code and actual Ada code. For example, for the first object of the first process, he finds the following:

$$\#BOD(1,1,1) = 55$$
$$\#BOD(1,1,2) = 180$$
$$\#BOD(1,1,3) = 260$$
$$\#BOD(1,1,4) = 350$$
$$\#BOD(1,1,5) = 250$$
$$\#BOD(1,1,6) = 375$$
$$\#BOD(1,1,7) = 115$$

Therefore, it follows that:

$$\#OPERS(1,1) = \sum_{l=1}^{7} \#BOD(1,1,l) = \underline{1385}$$

The results for all objects are also shown in Table 7.

Sixth, the estimator uses Equation 3-7 to estimate the number of delivered source statements to implement each object in each process, #DSI(p,o). For example, for the first object of the first process, he finds the following:

$$\#DSI(1,1) = \#DATA(1,1) + \#PU(1,1) + \#OPERS(1,1)$$

or

$$\#DSI(1,1) = 90 + 42 + 1385 = \underline{1517}$$

The results of all objects are shown in Table 7.

Seventh, the estimator uses the results in Table 7 and Equation 3-6 to estimate the number of source statements to implement the object-oriented Ada design, #SS, as follows:

$$\#SS = \sum_{p=o}^{P} \sum_{o=1}^{o(p)} \#DSI(p,o)$$

$$\#SS = 400 + \sum_{o=1}^{10} \#DSI(1,o) + \sum_{o=1}^{12} \#DSI(2,o) + \sum_{o=1}^{9} \#DSI(3,o)$$
$$+ \sum_{o=1}^{8} \#DSI(4,o) + \sum_{o=1}^{11} \#DSI(5,o) + \sum_{o=1}^{12} \#DSI(6,o)$$
$$+ \sum_{o=1}^{14} \#DSI(7,o) + \sum_{o=1}^{9} \#DSI(8,o) + \sum_{o=1}^{8} \#DSI(9,o) + \sum_{o=1}^{7} \#DSI(10,o)$$
$$= 400 + 13,932 + 9,367 + 9,506 + 6,859$$
$$+ 11,663 + 9,148 + 11,472 + 6,765 + 4,325 + 5,298 = \underline{88,335}$$

4.4 PROJECTING DEVELOPMENT COSTS

Having established attributes and a size metric, the estimator can now calculate projections for the cost of the object-oriented Ada-unique versions of the embedded computer program.

4.4.1 Object-Oriented Ada (Short-Term Costs)

With the chosen cost driver attributes for the software development effort to be undertaken in the short-term and the estimate of the size metric for each object, the estimator proceeded with the short-term cost calculation. Specifically, he established the 3 cost factors of Equation 3-1:

- Design cost
- Object implementation cost
- Object integration cost

4.4.1a Calculating Design Costs

Equation 3-2 was applied to establish the first factor, design costs. The factor '% DESIGN' was calculated by adding together the percentage of software development costs associated with product design, P(1), to the percentage of software development costs associated with detailed design costs, P(2). P(1) and P(2) were selected from Column 4 of Table 4 (under 128 KSDI) resulting in:

$$\% \text{ DESIGN} = P(1) + P(2) = 18 + 25 = 43\%$$

The factor MM(TOTAL) was calculated using Equation 2-3 as follows:

$$MM(TOTAL) = (C_1 * C_2 * \ldots C_{15}) K(DSI/1000)^{**}E$$

$$MM(TOTAL) = (1*1*1*1*1*1.12*1*1*1*1*1.08*1.06*.89*.89*1) *2.8(88,335/1000)^{**}1.2 = 618.2$$

Therefore, applying Equation 3-2:

$$\{\text{DESIGN COST}\} = (\% \text{ DESIGN}/100)*MM(TOTAL) = (43/100)*618.2 = \underline{265.8}$$

4.4.1b Calculating Object Implementation Costs

Equation 3-4 was applied to establish the second factor of Equation 3-1, object implementation costs. The factor P(3), the percentage of the software development cost associated with program unit code and test was found in Column 4 of Table 4 (under 128KDSI) to be 26%. The factors MM(p,o) were calculated using Equations (2-5) to (2-8) of Intermediate COCOMO and the productivity associated with 2000 source statements as follows:

1. Applying Equation 2-5 for a 2000 source statement module,
   $MM_N = 2.8(2000/1000)^{**}1.2 = 6.43$ manmonths
2. Applying Equation 2-6,
   NOM PROD = 2000/6.43 = 311 source statements/manmonth

3. Applying Equation 2-7, the nominal cost of the oth object on the pth process is given by:

$$M_N(p,o) = DSI(p,o)/311$$

4. Applying this result in Equation 2-8,

$$MM(p,o) = C_1(p,o)*C_2(p,o)...C_{15}(p,o)*MM_N(p,o)$$
$$= C_1(p,o)*C_2(p,o)*...C_{15}(p,o)*DSI(p,o)/311$$

5. Applying Equation (3-4), it follows that the Object Implementation Cost (OIC) is calculated as follows:

$$OIC = P(3) \sum_{p=1}^{P} \sum_{o=1}^{O(p)} MM(p,o)$$
$$= P(3)* \sum_{p=1}^{P} \sum_{o=1}^{O(p)} C_1(p,o)*C_2(p,o)....C_{15}(p,o)*DSI(p,o)/311$$

6. Setting $C_1(p,o)*C_2(p,o)$ $....C_{15}(p,o)$ to a constant for all values of p,o with the values shown in Table 4-1, it follows that

$$OIC = P(3)*K \sum_{p=1}^{P} \sum_{o=1}^{O(p)} DSI(p,o)/311$$

$$= P(3)*K*\#SS/311$$
$$= .26*1.0156*88,335/311 = \underline{75.0} \text{ man-months}$$

This calculation was not made directly using Equation (2-3), since the size of Ada object implementation is typically less than 2000 source statements and Equation (2-3) is applicable only for 2000 or more source statements. Accordingly, the component estimation approach described in Section 2.3.3 was applied based upon the productivity associated with a 2000 source statement object.

4.4.1c Calculating Object Implementation Integration Costs

Equation 3-5 was applied to establish a lower limit on the third factor, object implementation integration costs. The percentage of software development costs associated with program unit integration P(4), was found to be 31% in Column 4 of Table 2-3 (under 128KDSI). The factor MM(EQUIV) was calculated using Equation 2-3 as follows:

$$MM(EQUIV) = (C_1 * C_2 * ...C_{15}) \; K \; (DSI(EQUIV)/1000)**E$$
$$= (1*1*1*1*1*1.2*1*1*1*1*1.08*1.06*.89*.89*1)2.8(4900/1000)**1.2 = \underline{19.2}$$

where the integration of 100 objects was assumed equivalent to integrating 100 program units of 49 source statements each (i.e., the probable size of a program unit per the Theory of Sevens, as shown in Table 5). Therefore,

(Lower Limit for Object Implementation Integration) = .31*19.2 = 6.0

4.4.1d Total Short-Term Costs

Thus, the lower limit on the cost in the short-term is established using Equation 3-1 as follows:

Lower Limit on Ada Short-Term Costs = (Design Cost) + (Object Implementation Cost) + (Object Implementation Integration Cost)
= 265.8 + 75.0 + 6.0 = 346.8 manmonths

The upper limit is established directly using Equation 2-3 of Intermediate COCOMO as follows:

Upper Limit on Ada Short-Term Costs = $(C_1 * C_2 * \ldots C_{15})K(DSI/1000)^{**}E$

=(1*1*1*1*1*1.12*1*1*1*1*1.08*1.06*.89*.89*1) 2.8(88,335/1000)**1.2 = 615.5 manmonths

In practice, if the objects are sufficiently decoupled, the lower limit would provide the more accurate estimate of the software development cost. As the coupling of objects increases, the cost of the software development would tend to drift toward the upper limit, depending upon the extent of object coupling introduced.

4.4.2 Object-Oriented Ada (Long-Term Costs)

The estimator then modified his estimate of the object-oriented Ada effort made for the short-term effort to establish the cost estimate applicable to the long-term. Specifically, he changed the cost driver attributes as shown in Table 4-1. With the new cost driver attributes, the estimator established the following projection:

Lower Limit on Long-Term Costs = 172.9 manmonths

Upper Limits on Long-Term Costs= 306.8 manmonths

4.4.3 FORTRAN/Assembly Language Costs

The estimator then applied the cost drivers he chooses to estimate the software development effort undertaken using the FORTRAN/assembly language combination and a traditional structured top-down design.

The estimator was not sure that the size metric used to make the object-oriented Ada projection was applicable to the FORTRAN/assembly language projection, having read the following in The Analytic Sciences Corporation report entitled "The Impact of Ada on Software Development Costs:"[11]

> "Software Size Impact - The number of delivered source lines of code (DSLOC) or delivered, executable, machine-level instructions (DEMI) that will be required to program an application using Ada will differ from the amount required to program the same application in another HOL. Certain features of Ada will cause a reduction in the DSLOC, while others will cause an increase. For example, the richness of the Ada language requires fewer DSLOC to achieve certain functions, while the requirement for disciplined, explicit declaration of data items requires more DSLOC."

The estimator investigated this matter and found that Software Productivity Research, Inc. of Cambridge, Massachusetts had published numbers relating the verbosity of Ada to FORTRAN and assembly language. Specifically, their data indicates that FORTRAN is 48% more verbose than Ada and assembly language is 250% more verbose than Ada.[12] Using these relative verbosity numbers and assuming that 20% of the FORTRAN/assembly language combination would be written in assembly language, the estimator calculated that the FORTRAN/assembly language combination would be 1.9 times more Ada (i.e., .2*3.5 + .8*1.48 = 1.884). He then assumed a ± 20% error in this relative verbosity and established the #SS for the FORTRAN/assembly language combination with the following:

$$\#SS\ (FORTRAN/assembly) = \{1.9 * \#SS\ (Ada)\} \pm 20\% = 167{,}837 \pm 20\%$$

As indicated by Boehm in his book Software Engineering Economics, the cost to develop software is a direct function of the number of source statements that must be developed. The estimator made this assumption and using Equation (2-3), he made the following calculations:

Lower Limit on FORTRAN
Assembly Language Costs = .87*2.8(134,269/1000)**1.2 = 871.5 manmonths

Upper Limit on FORTRAN
Assembly Language Costs = .87*2.8(201,404/1000)**1.2 = 1417.6 manmonths

The results of the set of cost projections are shown in Table 8.

4.5 CONCLUSIONS

An estimator must understand factors that will affect the cost of Ada software development. A cost model must be chosen that can account for the Ada-unique factors.

In this chapter, a special algorithm has been formulated for estimating the cost to develop Ada software in an object-oriented manner. COCOMO was used as the basis for estimating factors within the special algorithm, since:

- Studies have indicated that COCOMO exhibits reasonable accuracy in the estimation of software life-cycle cost when its inputs are chosen correctly.[13]

- COCOMO predicts costs as a function of model inputs that can be selected to account for Ada's expected impact in both the short-term and long-term on compiler performance, programmer experience, use of modern programming practices and software tools. Proponents of Ada predict that these factors have a significant effect on cost savings in the long-run.

TABLE 8

COST/SCHEDULE ESTIMATES FOR HYPOTHETICAL SOFTWARE

DEVELOPMENT APPROACH	COST (ManMonths)
Object-Oriented Ada (Short-Term)	346.8 to 615.5
Object-Oriented Ada (Long-Term)	172.9 to 306.8
FORTRAN/Assembly Language Combination	871.5 to 1417.6

#SS = 88,335
K = 2.8
E = 1.2

$C_1, C_2, \ldots C_{15}$ set per Table 4-6

- Comprehensive documentation exists describing how to select numerical values for input factors.

- COCOMO can be used to project the cost to design Ada software and implement individual objects. These are basic cost factors in the special algorithm applicable to object-oriented design.

The results of applying the special algorithm for the estimation of object-oriented Ada software in a hypothetical system are shown in Table 8. The range of cost values shown for the development object-oriented Ada software account for the extent of decoupling between objects. Objects that are strongly decoupled will cost must less to develop than objects that are strongly coupled. In the limit, the cost to develop strongly coupled objects will approach the cost to develop Ada software in a traditional manner (i.e., with strong dependencies between types, variables and program units). In our example, strongly decoupled objects cost less than 70% of the cost incurred in developing strongly coupled objects.

Additional significant savings with Ada are expected in the long run due to positive affects of a standardization on such software acquisition attributes as virtual machine stability, personnel experience and use of software tools.

5. FUNCTION POINT ANALYSIS

The size metric algorithm presented in Section 3.4.2 can be used in conjunction with SHARP abstracts to project the number of Ada source statements needed to implement an Ada computer program. Factors in the algorithm are derived from the SHARP abstracts that represent the object-oriented Ada design for the computer program.

As an alternative during cost estimation, SHARP abstracts could be used to establish factors associated with function point analysis, to provide a straight forward and fast way to establish the size metric. The size metric can be projected as a function of a metric referred to as Function Point Total (FPT), which was developed by Allen Albrecht of IBM over the last decade. It is defined by the following relationship:

FPT =(Complexity Adjustment) * (4*#Inputs + 5*#Outputs

+ 4*#Inquiries + 10*#Data Files + 7*#Interfaces) (5-1)

where #Inputs refers to both data and control information entering a computer program from the an external source; #Outputs refers to both data and control information leaving the computer program for an external source; #Inquiries refer to such things as 'HELP' screens and selection menus; #DATA Files include flat files on tape or disk, a 'leg' of a hierarchical data base and a table in a relationship data base; and #Interfaces are defined as files passed between or shared among separate applications. The Complexity Adjustment factor ranges from .75 to 1.25. (In 1985, the Albrecht methodology modified the original complexity adjustment to multiple complexity adjustments -- one for #Inputs, one for #outputs, one for #Inquiries, one for #Data Files and one for #Interfaces.)

Software Productivity Research, Inc. of Cambridge, Massachusetts has developed a product that projects the number of source statements it will take to implement a computer program as a function of the Function Point Total. As shown in Table 9, the software product (referred to as SPQR[TM]) establishes the number of source lines per function point for 30 programming languages, including Ada. In Table 9, the 'level' of a language is defined as the approximate number of assembler language statements that an experienced programmer would write to create the effect of one source statement in the language being used. Source code statements are defined in SPQR as consisting of executable statements and data definitions.

The factors used to calculate an object's Function Point Total could be established by inspection of SHARP abstracts. For example, SHARP data flow diagrams for program units facilitating data transfer into and out of an object implementation could be used to identify the factors #Inputs and #Outputs for the object; and the SHARP data structure diagram for an object could be used to help establish #Inquiries, #Data Files and #Interfaces.

6. CHAPTER SUMMARY

COCOMO cannot be directly used to estimate the cost to develop Ada software in an object-oriented manner. However, by applying COCOMO in a special manner, the parts of an object-oriented effort can be individually assessed. Namely, design costs, object implementation costs and object implementation integration costs can be individually addressed. Design costs can be estimated using COCOMO by assuming that these costs would be essentially equivalent to the costs incurred using a traditional design, which is projected by COCOMO. Object implementation costs can be estimated by assuming that the development of each object is essentially an independent effort and applying COCOMO directly to each of these efforts.

Object integration costs cannot actually be estimated using COCOMO. However, as a lower limit, the integration effort can be assumed to be essentially equivalent to that of integrating program units, since loosely coupled object interfaces ideally should not be more complicated than typical program unit interfaces.

Using these assumptions in an example, we found that the cost to develop Ada software consisting of strongly decoupled objects is less than 70% of the cost incurred in developing strongly coupled objects. In practice, the cost of Ada software developed in an object-oriented manner depends upon the extent of decoupling between objects. In the limit, the cost to develop strongly coupled objects would approach the cost to develop highly coupled software with strong dependency relationships.

When applying COCOMO, whether directly in projecting the costs of highly coupled software components or indirectly in projecting the cost of decoupled object-oriented Ada software, the size of the software is the key input parameter to the COCOMO model. Using SHARP abstracts, the size metric can be estimated directly as a function of the computer program's design or the Function Point Total.

TM -- SPQR is a trademark of Software Productivity Research, Inc.

TABLE 9

NUMBER OF COMPUTER PROGRAM SOURCE STATEMENTS
PER FUNCTION POINT TOTAL

SOURCE LANGUAGES AND LEVELS IN SPQR

Language	Level	Source Lines Per Function Point
1. Basic Assembler	1	320
2. Macro Assembler	1.5	213
3. C	2.5	128
4. ALGOL	3	105
5. CHILL	3	105
6. COBOL	3	105
7. FORTRAN	3	105
8. Mixed Languages (Default)	3	105
9. Other Languages (Default)	3	105
10. PASCAL	3.5	91
11. RPG	4	80
12. PL/I	4	80
13. MODULA 2	4	80
14. Ada	4.5	71
15. PROLOG	5	64
16. LISP	5	64
17. FORTH	5	64
18. BASIC	5	64
19. LOGO	5.5	58
20. English-Based Languages	6	53
21. Data Base Languages	8	40
22. Decision Support Languages	9	35
23. Statistical Languages	10	32
24. APL	10	32
25. OBJECTIVE-C	12	27
26. SMALLTALK	15	21
27. Menu-Driven Generators	20	16
28. Data Base Query Languages	25	13
29. Spread-sheet Languages	50	6
30. Graphic Icon Languages	75	4

This table has been taken from "Software Productivity," Volume 1, Number 1, Software Productivity Research, Inc., March/April 1986.

CHAPTER VIII

Teaching Object-Oriented Ada

In this chapter, we address the level and depth of Ada instruction appropriate for different personnel types involved in a DoD software acquisition, including project managers, system engineers, software engineers and programmers. The use of SHARP abstracts is described in the context of instructing students in developing Ada software in an object-oriented manner. Specifically, we address teaching project managers and system engineers (a) object-oriented technology, (b) basic concepts in an Ada implementation of an object-oriented design, and (c) the cost ramifications of this approach. In addition, we address teaching software engineers and programmers the use of Ada to implement an object-oriented design. In both cases, the abstracts of SHARP can be used to graphically illustrate the basic technology, thus establishing a solid knowledge base for learning Ada implementation detail.

We suggest that it is important that appropriate Ada instruction be given to project managers and system engineers, as well as software engineers and programmers. Of course, the level and depth of this instruction should not be the same. However, the gap in knowledge between personnel involved in software acquisition must be constrained to the extent that effective communication can take place during system acquisition.

We further suggest that Ada should be taught in the context of object-oriented design and in a top-down manner (i.e., high-level structure down to low-level implementation detail). This approach can be taken using the abstracts of SHARP, since with SHARP Ada computer programs can be represented by graphics. The graphics can be used to provide clear mental pictures of complex technology. Then, once the technology has been introduced, design implementation in Ada source code can be taught.

1. INTRODUCTION

1.1 BACKGROUND

As discussed in the earlier chapters, during the development of large and complex computer programs, the use of an object-oriented approach is critical because of the need to control complex dependency relationships between types, variables and program units. In addition, the object-oriented approach is critical because of the speed of Ada compilers. With object-oriented Ada, requirements assigned to objects are implemented in loosely coupled Ada packages and tasks, which the development team can code and test independently. By constraining their size and stubbing program units interacting with them, the Ada packages and tasks can be compiled and recompiled in a timely manner during their development. With traditional methodologies and highly coupled program units, the compilation of large and complex Ada computer programs is a significant problem.

It is important that appropriate instruction in the effective use of Ada be given to personnel involved in the acquisition of DoD software, including project managers and system engineers as well as software engineers and programmers. Of course, the level of detail and emphasis of the instruction provided to the different team members should have the appropriate depth and point of view.

Program managers must have sufficient knowledge of an Ada effort to intellectually grasp the problem they must manage. Experience has shown that misunderstood projects tend to go astray. Initial budgets tend to be insufficient and resource allocation during the course of the software development effort may not be appropriate. It is important that program managers and administrators understand the cost/schedule ramifications of object-oriented Ada efforts.

System engineers are responsible for system conception and the specification of system requirements. They must have sufficient knowledge of Ada and an object-oriented approach to software development to properly allocate system requirements to software, to establish effective hardware processing configurations, and to establish appropriate interfaces between hardware and software.

Software engineers must have advanced knowledge of Ada and the object-oriented methodology. They must have the capability to distribute software requirements to objects and must be able to establish a detailed object-oriented design representation that can be easily understood by programmers. They must be clearly aware of the cost/schedule ramifications of their designs.

Programmers must not only have advanced knowledge of how to establish Ada code, but also should have adequate (if not comprehensive) knowledge of the object-oriented approach to the development of Ada software. They must correctly interpret the design representation and transform it into detailed Ada code. They then must rigorously test the code to assure that it accurately implements the design established by a software engineer.

Software test engineers must have sufficient knowledge of the object-oriented approach to the development of Ada software to effectively integrate and test implemented objects, upon their release by programmers. They must be able to map software requirements specified in a Software Requirements Specification (DI-MCCR-80025) into test stimuli that exercise integrated object implementations.

Thus, instruction must be provided to different personnel associated with an object-oriented Ada development effort. It is generally recognized that such instruction must cover methodology, Ada programming support environments (APSEs) as well as language-unique considerations. However, the level of detail of instruction provided to different personnel types will vary, such as in the manner suggested in Table 10.

1.2 CHAPTER SCOPE

In this chapter, the use of SHARP abstracts to teach Ada software is described in two major sections. The discussion addresses 'design methodology' associated with object-oriented techniques and 'Ada language' usage to implement object-oriented designs, but excludes discussion of 'APSEs.'

TABLE 10 LEVEL OF INSTRUCTION REQUIRED IN OBJECT-ORIENTED ADA BY LABOR TYPE

LABOR TYPE	DESIGN METHODOLOGY	ADA PROGRAMMING SUPPORT ENVIRONMENTS (APSEs)	ADA LANGUAGE
Project Managers	M	M	L
System/Test Engineers	H	M	L
Software Engineers	H	H	H
Programmers/QA Personnel	H	H	H

H = High Level of Instruction Required
M = Medium Level of Instruction Required
L = Low Level of Instruction Required

In Section 2, an overview of the use of SHARP for instruction appropriate for project managers, system engineers and software test engineers is provided. Such instruction primarily should introduce the concept of object-oriented software development and the economics of the methodology. This instruction can be accomplished independent of Ada code by using the graphics of SHARP.

In Section 3, an overview of the use of SHARP for instructing software engineers, programmers and software quality assurance personnel is provided. In this section, we address approaches to instruction in learning specific steps in an object-oriented Ada design, the representation of that design with SHARP abstracts, and the transformation of the design to Ada code. The traditional approach to Ada instruction is discussed, whereby Ada is taught in a bottom-up manner (i.e., low level detail up to high level structure). As an alternative and possible improvement over the traditional approach, teaching Ada in a top-down manner is discussed (i.e., high level structure down to low level detail). The latter approach is possible with the abstracts of SHARP, since Ada computer programs can be completely represented by graphics in teaching technical concepts. Then, once the technical concepts have been introduced, their implementation in code can be taught.

2. ADA INSTRUCTION FOR PROJECT MANAGEMENT AND SYSTEM ENGINEERING PERSONNEL

This section addresses the level and depth of Ada-unique instruction appropriate for project management and system engineering personnel. It is pointed out that such personnel should understand essential aspects of an object-oriented Ada computer program, but not in terms of Ada syntax and lexical units. Rather, it is suggested that the important concepts associated with object-oriented Ada efforts must be presented using a level of abstraction higher than Ada code -- a level free of confusing detail.

We feel SHARP abstracts provide a mechanism for teaching Ada concepts that can be understood by management and system engineers, without introducing code detail beyond their scope of comprehension and need.

As we have already stated, experience has proven that software projects not understood tend to go astray. Typically, initial budgets tend to be insufficient and resource allocation inappropriate. Accordingly, it is important that basic concepts of Ada-unique object-oriented software be taught to management and system engineering personnel.

2.1 ADA INSTRUCTION APPLICABLE TO PROJECT MANAGERS

Project managers are responsible for managing a large and complex system acquisition. Government managers direct the preparation of the System/Segment Specification (DI-CMAN-80008) and establish proper funding for the acquisition. They are responsible for approval of software requirements and design specifications, test procedures and ultimate acceptance of the system.

Contractor project managers interface with government managers and direct the development of the system. Contractor project managers, responsible for software development, plan and direct software-related work. The planning activity encompasses:

- Defining goals, budgets, and schedules for the acquisition of a Computer Software Configuration Item (CSCI)
- Identifying needed personnel, development facilities, and other resources
- Investigating and evaluating costs, resources, and the availability of these resources
- Developing plans for using available resources to satisfy CSCI development objectives
- Estimating costs for the resources identified by the plans.

The direction activity encompasses:

- Staffing the CSCI acquisition
- Supervising personnel and appraising performance
- Reviewing documentation
- Evaluating expenditures versus budgets
- Checking accomplishments versus scheduled events
- Checking progress and addressing problems encountered
- Instigating changes to goals, budgets, and schedules to assure that any detected budget overruns and schedule slips are minimized

- Introducing changes to goals, budgets, and schedules to satisfy management or the government

- Informing responsible project directors of any slips in CSCI acquisition schedules and budgets that cannot be corrected.

There are a multitude of issues that software project managers must consider, as discussed in the Program Manager's Guide to Ada.[3] Clearly, project managers must understand the concept of object-oriented Ada software development, and the cost ramifications of this approach, to intelligently address the multitude of Ada-related issues.

2.1.1 Instruction for Project Managers in Object-Oriented Ada-Unique Concepts

Although project managers typically should not be expected to learn Ada lexical units and syntax, they should be very familiar with the material presented in Chapter I. For example, they should be aware of the basic building blocks of an Ada computer program -- the program units called subprograms, tasks and packages. They should be aware of how these program units are used to establish processes. As discussed in Section 3 of Chapter I, SHARP pictographs can be used to represent an Ada computer program at this high level, such as in the manner shown in Item a of Figure 99. As this figure illustrates, the SHARP graphic illustrates concurrently executing process tasks that are declared in the main Ada program to service such things as communication links, terminals, work stations and interfacing microprocessors.

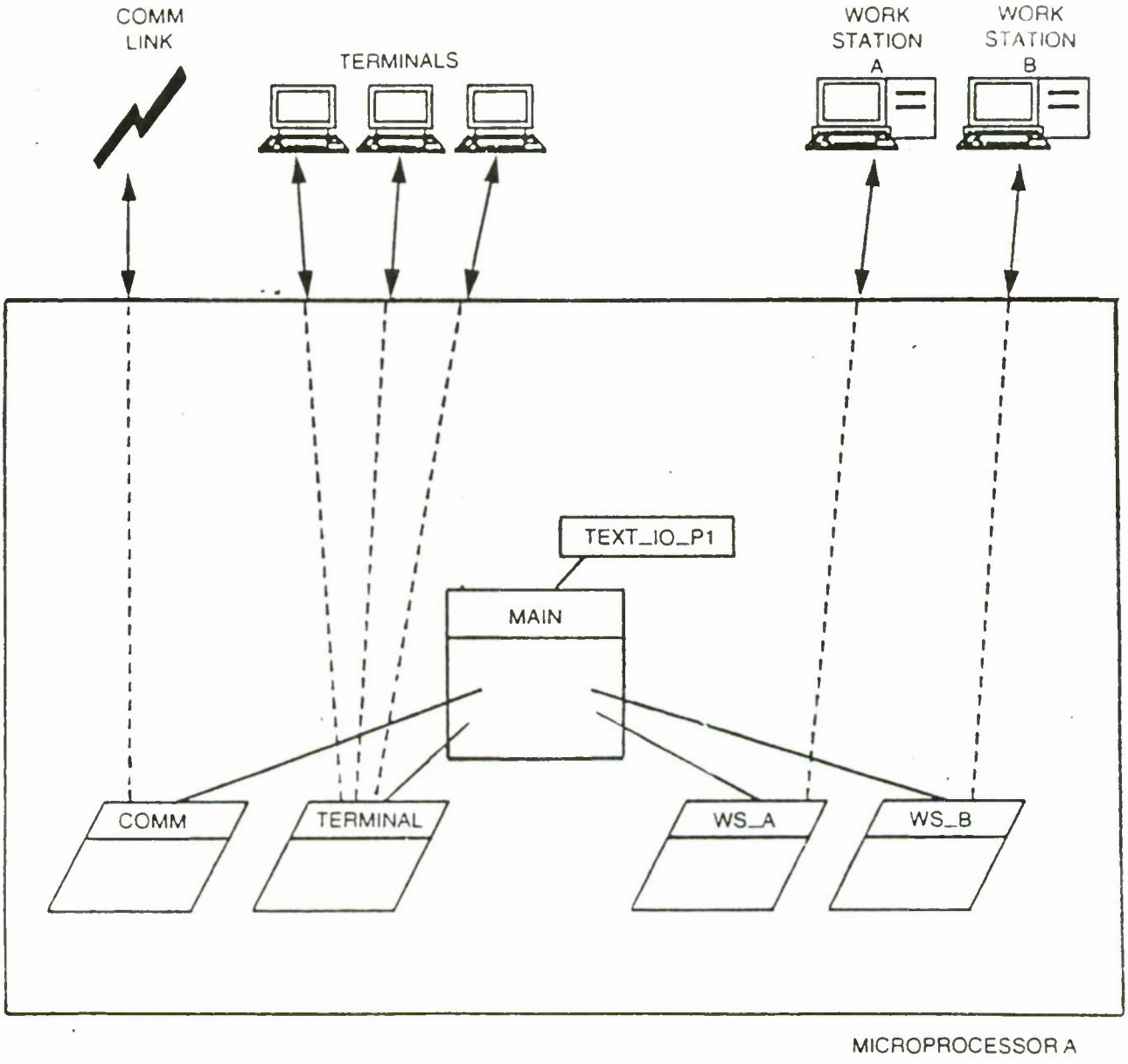
The project managers should also be taught basic concepts for process implementation in an object-oriented manner, at the level of detail shown in Chapter III. As discussed in this chapter, requirements assigned to each process should be distributed among objects. The objects can be implemented using Ada packages or tasks, and should be loosely coupled. The SHARP abstracts shown in Items b and c of Figure 99 can be used to explain these concepts to project managers.

2.1.2 Instruction for Project Managers in Ada-Unique Cost/Schedule Estimation

Project managers should also be taught the cost ramifications of the object-oriented approach to software development versus traditional approaches. It is important that they understand how existing cost models can be used in estimating the cost of object-oriented Ada software, and when they must not be used. This is critical to the project managers establishing meaningful budgets and schedules, and projections for the time and cost at completion.

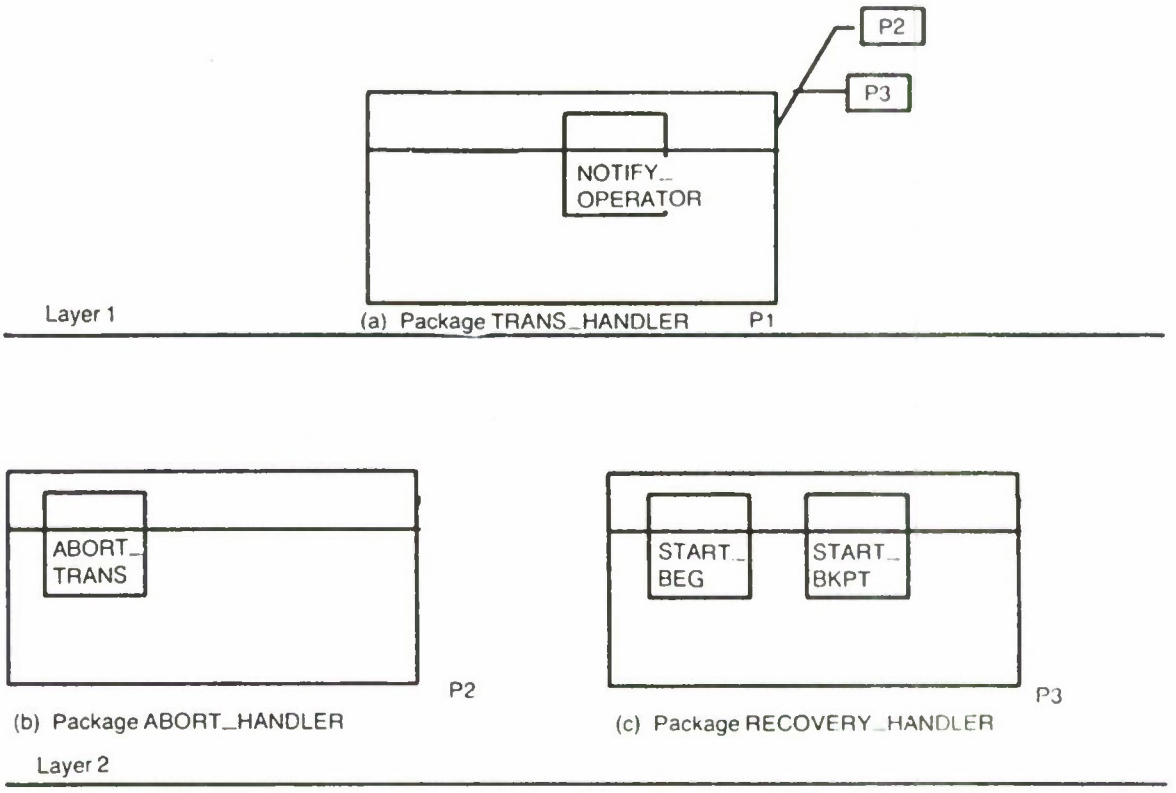
2.2 ADA INSTRUCTION APPLICABLE TO SYSTEM ENGINEERS

System engineers collect, define and evaluate system requirements, many of which must be implemented in software. System engineers in the government interview users, higher headquarters and other agencies authorized to specify system requirements. They document the requirements in a System/-Segment Specification, in accordance with DI-CMAN-80008.



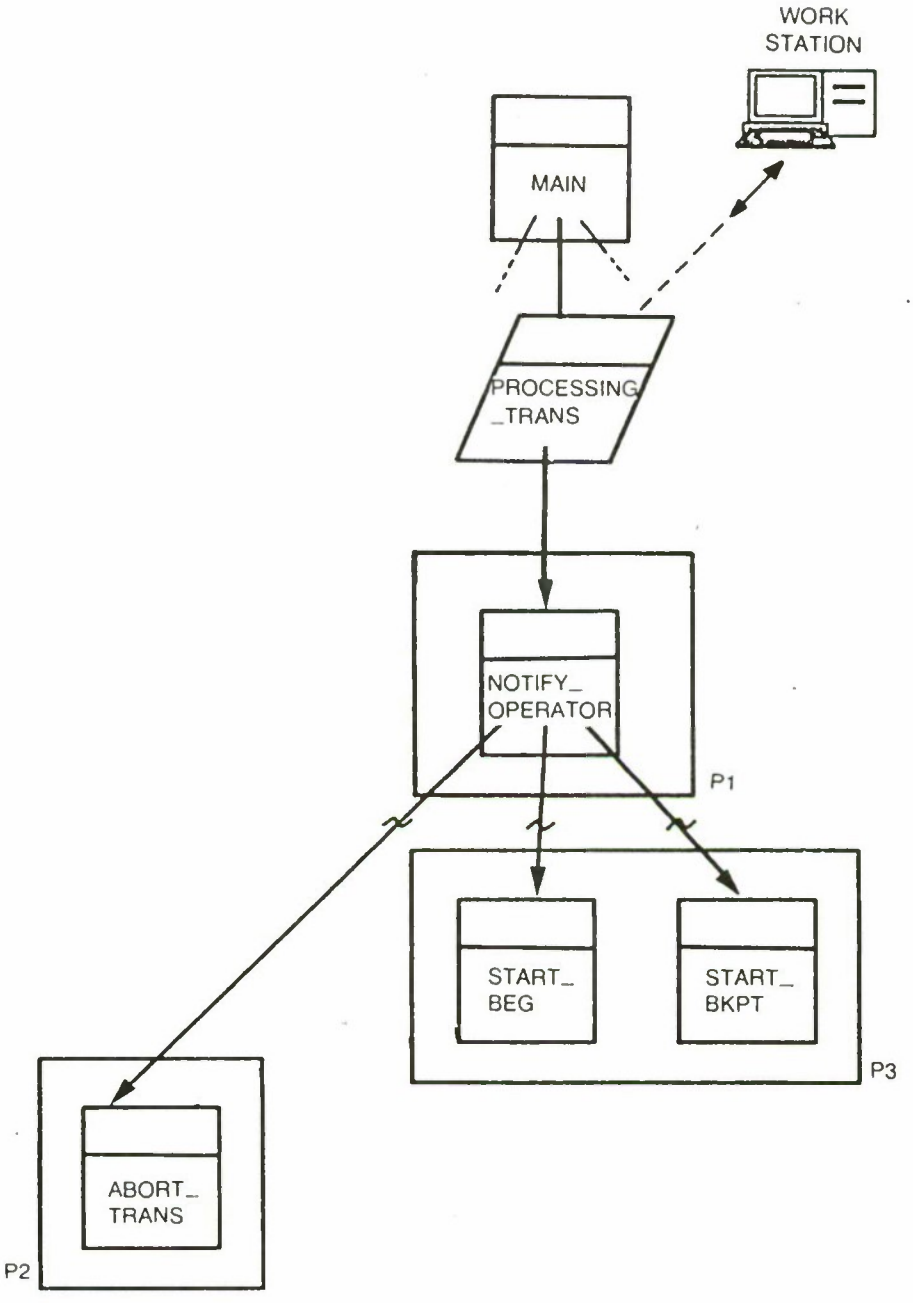
(a) REPRESENTING PROCESSES ESTABLISHED IN THE MAIN PROGRAM

FIGURE 99. SHARP REPRESENTATION OF A COMPUTER PROGRAM AT A HIGH LEVEL



(b) ADA PACKAGES USED TO ENCAPSULATE OBJECT IMPLEMENTATIONS

FIGURE 99. (CONTINUED)



(c) INTERACTION OF OBJECT IMPLEMENTATIONS

FIGURE 99. (CONCLUDED)

Contractor system engineers receive and interpret the System/Segment Specification. They prepare a Software Requirements Specification in accordance with DI-MCCR-80025 and an Interface Requirements Specification in accordance with DI-MCCR-80026. Their definition activities encompass:

- Drafting operational concepts and deployment plans
- Defining operational mode requirements including rules for transition from one operational mode to another
- Defining system inputs and outputs, and defining operations on inputs to produce outputs
- Specifying categories of performance including precision, accuracy, reliability, and maximum allowable processing time (i.e., time to produce an output from inputs)
- Specifying external interface requirements, including specific interface with other systems, information flow and data rates across each, and conditions affecting information flow
- Specifying man-machine interfaces, including display legibility and man-machine dialogue
- Defining design constraints, including required architectural features, memory constraints, algorithms, and function allocations
- Defining the testing requirements against which detailed system test plans and procedures must be written
- Defining cost and schedule constraints, including cost limits and delivery dates.

The evaluation activities encompass:

- Organizing proposed requirements (e.g., classifying collected requirements by source, type, and version)
- Interpreting and restating obscure requirements as necessary to clarify them
- Restating requirements in formal memoranda and working papers for informal validation by representatives of the government
- Reviewing and revising the requirements based on feedback from the government.

The approach to establishing and presenting software requirements by a system engineer should be undertaken with knowledge of how these requirements will be implemented in an object-oriented Ada-unique manner. As is the case for a project manager, system engineers must understand the

concept of object-oriented Ada software development, and the cost ramifications of this approach. In addition, they must understand how to efficiently test an object-oriented Ada computer so that they can draft effective test plans, procedures and reports.

2.2.1 Instruction for System Engineers in Object-Oriented Ada-Unique Concepts

As is the case for project managers, system engineers typically should not be expected to learn Ada lexical units and syntax. However, they must have sufficient knowledge of object-oriented Ada-unique concepts to properly specify requirements in a manner that easily leads to distribution of requirements among processes and objects, and the development of procedures needed to test the implementation of the requirements.

Accordingly, like the project manager, they must be taught the basics of Ada presented in Chapter I and the fundamentals of an object-oriented design demonstrated in Chapter III.

2.2.2 Instruction for System Engineers in Testing Object-Oriented Ada Software

In addition, system engineers responsible for software testing must be taught how to test an object-oriented Ada computer program. DOD-STD-2167 and its companion test-unique data item descriptions (DIDs) specify government requirements for testing deliverable software. Testing of object-oriented Ada software can be envisioned as a mapping of software requirements into test stimuli for a set of test cases. Special purpose test-unique Ada packages are needed to produce the test stimuli and record parameters passed between objects. The recorded data is compared to expected values, either directly or after data reduction calculations.

System engineers should be familiar with issues associated with the test of object-oriented Ada software, such as the material presented in Chapter VI. They must be taught how to nominally test object-oriented software, where stimuli and operating conditions are typical of those to be experienced by the software in an operational "real world" environment. They must be taught how to stress test object-oriented software, where stimuli and operating conditions are extreme, unusual or even erroneous. They must be taught how to test the performance of object-oriented software over long time periods, to see if it fails with time and to identify patterns and biases in results that may set in with time.

As discussed in Chapter VI, SHARP abstracts can be used to explicitly represent test configurations. For example, the SHARP Invocation Diagram shown in Figure 100 represents object implementations encapsulated within Ada packages and special test software, which is represented with shaded pictographs. The test software provides environmental stimuli and records parameters passed between object implementations.

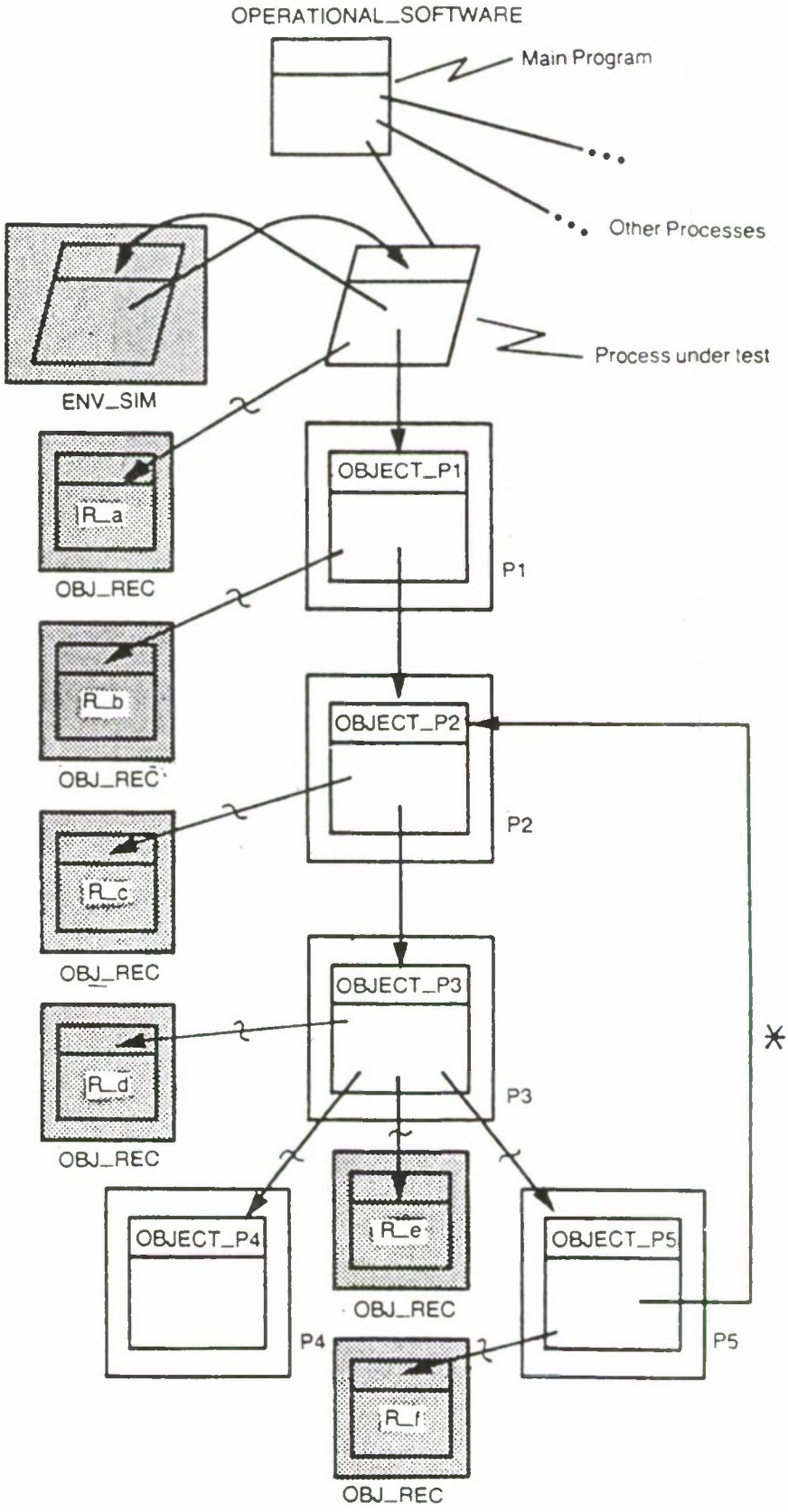


FIGURE 100. SHARP REPRESENTATION OF TEST SOFTWARE

2.2.3 Instruction for System Engineers in Ada-Unique Cost/Schedule Estimation

As is the case for project managers, system engineers should be fully aware of the cost ramifications of object-oriented software, as discussed in Chapter VII. They also must be aware of the cost ramifications of the tests they specify, and the dangers if the tests are not adequate.

3. ADA INSTRUCTION FOR SOFTWARE ENGINEERS AND PROGRAMMERS

Software engineers are responsible for developing an object-oriented Ada-unique design that can be used to map system requirements assigned to software to implementing code. The design is presented in the following documents:

- Software Top-Level Design Document (DI-MCCR-80012)
- Software Detailed Design Document (DI-MCCR-80031)
- Interface Design Document (DI-MCCR-80027)
- Data Base Design Document (DI-MCCR-80028)

Programmers are responsible for mapping the design of an object implementation into Ada source code and rigorously testing the code.

This section addresses the level and depth of Ada instruction appropriate for software engineers and programmers. It is pointed out that such personnel should understand design issues and the associated cost ramifications, as well as the details of coding with the Ada language. With SHARP, Ada code can be taught in the context of the overall design structure. The graphical aids supplied by SHARP provide mental pictures to a student for complex technical concepts. Having a full knowledge of these concepts is a prerequisite to learning the effective use of Ada

3.1 TEACHING ADA IN A BOTTOM-UP MANNER

Often textbooks and seminars on Ada present the Ada language in a bottom-up manner. With this approach, the "nut and bolts" of code are initially described -- the lexical units and syntax of Ada. Then, for example, typing, statements and Ada blocks might be presented. These aspects of Ada then might be used to describe Ada program units and data structures. Finally, such things as exception handling, generics, I/O and low level programming are presented.

Thus, if an instructor presents a course on Ada using the sequence of instruction presented in such textbooks, Ada is used to teach Ada in a bottom up manner. Code is taught in terms of code. When the bottom up approach to teaching Ada is used, we believe the SHARP graphics would help to introduce the technology associated with the use of Ada. For example, we believe the SHARP graphics will be useful in teaching the implementation of data structures with Ada -- especially the visibility of data.

Figure 101 illustrates the visibility of constants and variables found in packages. Variables and constants declared in the specification of a package can be completely visible, private or limited private. As shown in Item a of Figure 101, the geometric figures used to represent types, variables and constants in a SHARP Data Structure Diagram are unshaded when visible and partially shaded when private. In Ada, when a parameter is private, a user is excluded from applying operations on the parameters other than those operations defined within the package specification. The only exception to this rule is assignments and tests for equality and inequality, which can be made.

As shown in Item b of Figure 101, the geometric figures used to represent types, variables and constants in a SHARP Data Structure Diagram are shaded when not visible, having been declared in the body of a package.

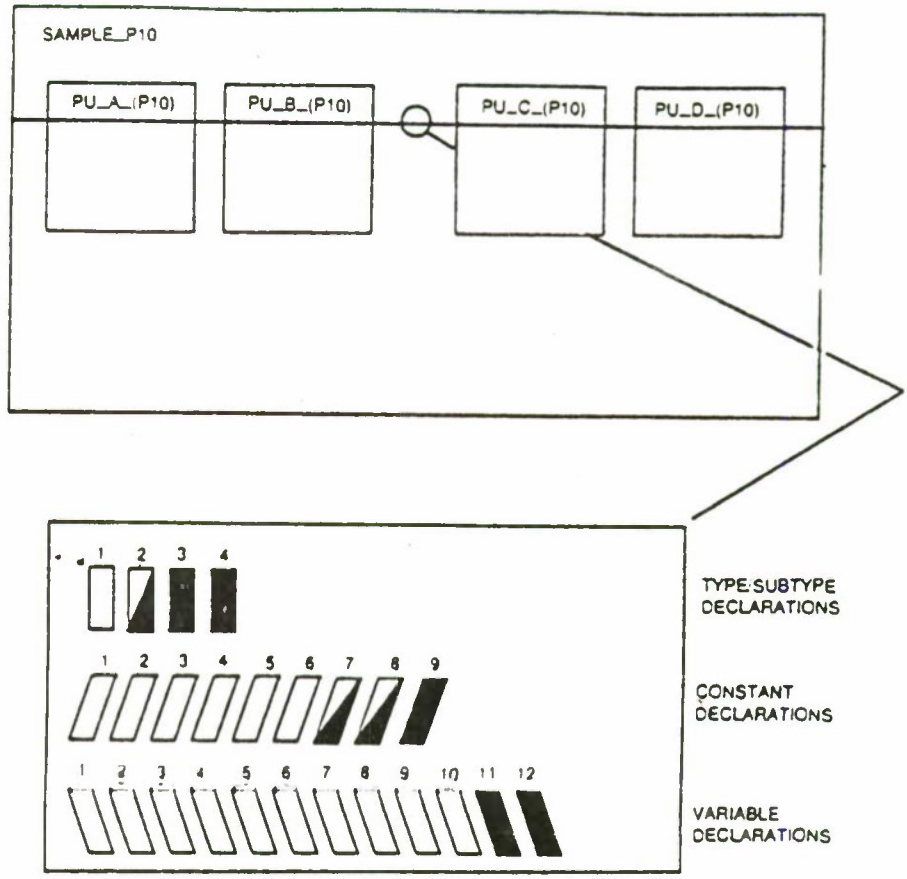
The use of such SHARP graphics will provide a pictorial image in the mind of the student for the data structure implemented with an Ada package. The mental image of such a data structure should be easier to learn and remember than the image supplied by code, especially since the student will not have initially mastered the varied and subtle rules of Ada (e.g., syntax and lexical units applicable to type declarations, constant declarations and variable declarations).

We feel that SHARP graphics will be a very effective aid when teaching Ada in a bottom-up manner. The mental images supplied by the graphics will help the student understand the code being taught.

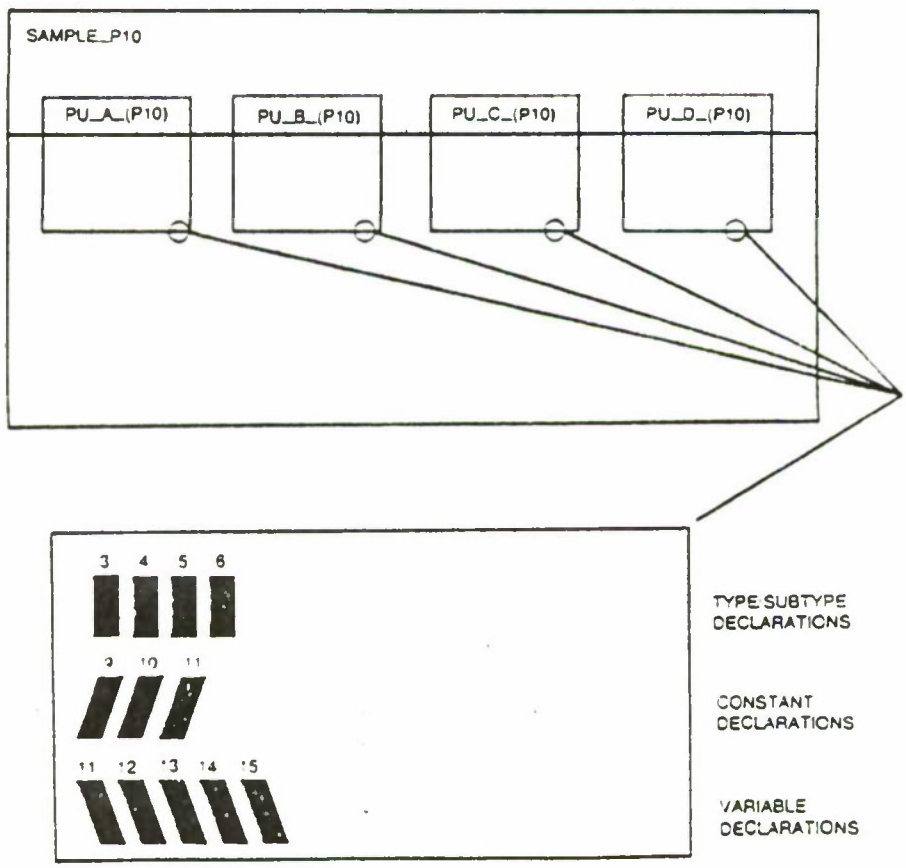
3.2 TEACHING ADA IN A TOP-DOWN MANNER

Teaching Ada in a bottom-up manner is a "bootstrap" operation. Ada code is used to teach other code. We feel there are shortcomings associated with teaching Ada in this manner. For example, if the student has not mastered one aspect of the code (which can happen on a short "shotgun" course in Ada), the lessons dependent upon that unlearned code may also go unlearned. Also, the student is learning Ada independent of the context of a design methodology and its cost ramifications. For example, data structure visibility may be taught independent of adequate understanding of information hiding in object implementations, an important principle in the development of software in an object-oriented manner.

We feel that large and complex Ada software should be built in an object-oriented manner. As discussed in Section 1.2, this approach is needed in order to control complex dependency relationships between variables, types and program units. In addition, the use of this approach can reduce problems introduced by the fact that Ada compilers are slow relative to compilers for most other high order languages, due in part because they make extensive checks of the complex dependencies. Using the object-oriented approach, a large and complex Ada computer program is constructed using a set of loosely coupled objects. The implementation of each object, can be thought of as separate small software acquisition. Being small relative to the overall software product, the object implementation can be compiled and recompiled in a timely manner.



(a) VISIBILITY OF DECLARATIONS MADE IN A PACKAGE'S SPECIFICATION AND BODY



(b) VISIBILITY OF DECLARATIONS MADE IN A PACKAGE'S BODY

FIGURE 101 VISIBILITY OF VARIABLE, CONSTANT AND TYPE DECLARATIONS

However, it is not possible to teach Ada in a top-down manner without an easily understood notation for Ada other than Ada source code itself. SHARP provides such a notation. The remainder of this section describes the scope of a set of lessons that could be used to teach Ada in a top-down manner in the context of an object-oriented design using the graphical notation of SHARP.

3.2.1 Lesson 1 - Process Abstraction

The first lesson describes the basic building blocks of an Ada computer program -- the program units called subprograms, tasks and packages. It introduces the basic pictographs of SHARP used to represent the program units.

Then, the student is taught how the designer of a large and complex computer program typically, as an initial step in the design process, identifies processes needed to establish concurrent processing threads. The implementation of the processes with Ada tasks is taught using the SHARP pictographs, such as those shown in Figure 98.

After having been taught basic technology of process abstraction using SHARP, the student is then ready to learn the Ada source code associated with this technology. To complete the lesson, the student is taught this code. For example, the basic code in Figure 102 could be used to complete this lesson.

3.2.2 Lesson 2 - Process Interaction

The second lesson introduces the task rendezvous, the Ada mechanism for intertask communication and, therefore, communication between processes. Task rendezvous is consummated by the callee (or acceptor task) after being initiated by a caller task. Parameters are passed between the caller and callee during the rendezvous, as represented by SHARP in the manner shown in Figure 103. As the figure shows, the parameters are passed from the caller to the acceptor via the **in**, **out**, or **in out** modes. Entry points in an acceptor task are shown as small parallelograms in the acceptor. Arrows are used to represent each of the three entry modes. Circles on these arrows represent the parameters being transferred.

After having been taught the basic technology of process interaction via task rendezvous using SHARP, the student is then ready to learn the Ada source code associated with this technology. To complete the lesson, the student is taught this code. For example, the basic code in Figure 104 could be used to commence this discussion. Then more advanced code can be presented to explain code-related technical details associated with the implementation of process interaction.

```
with TEXT_IO;
use TEXT_IO;
procedure MAIN is

   task COMM;
   task body COMM is separate;

   task TERMINAL;
   task body TERMINAL is separate;

   task WS_A;
   task body WS_A is separate;

   task WS_B;
   task body WS_B is separate;

end MAIN;
```

FIGURE 102. BASIC ADA CODE USED TO ESTABLISH PROCESSES

3.2.3 Lesson 3 - Object Implementation within Processes

The third lesson introduces the important concept of information hiding, which is fundamental to the implementation of object-oriented designs.

The lesson commences with a discussion of the basic concepts for the establishment of an object-oriented design (e.g., those described in Section 2.2 of Chapter II). The student must be presented the technical reasons for the object-oriented approach to software development (i.e., the control of complex dependencies between variables, types and program units) and the cost ramifications of this approach (i.e., significant cost reduction as compared to traditional approaches for large and complex computer programs). Presumably these facts have already been taught to the students in a preceding course, but should be reviewed as part of Lesson 3.

Next, the student must be taught how information hiding capabilities of Ada packages and tasks are used to encapsulate program units introduced to implement requirements assigned to each object.

The basic pictographs of SHARP can be used to help teach the technical concepts for object implementation, using information hiding capabilities inherent in Ada packages and tasks. For example, Figure 105 shows an Ada package, which contains a visible program unit introduced to facilitate communication with another object implementation, and which contains a local data structure not accessible to other objects. A visible data structure is also provided to account for the declaration of parameters passed to and received from other objects. The processing unique to this object implementation is hidden in the bodies of the communicating program units. Lessons for the implementation of the internal complexities of an object are described in Sections 3.2.7, 3.2.8 and 3.2.9.

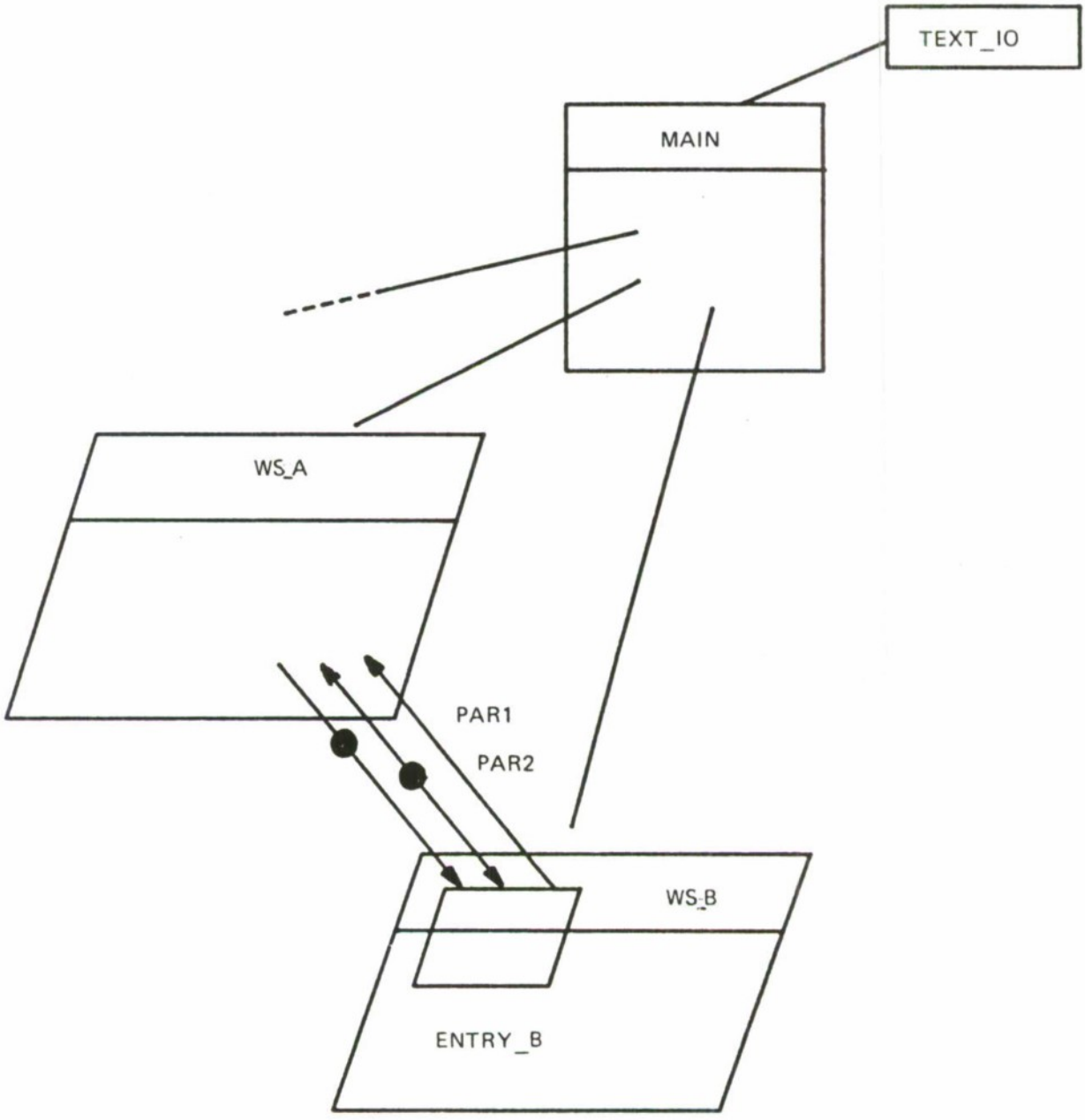


FIGURE 103 SHARP REPRESENTATION OF PROCESS INTERACTION (TASK RENDEZVOUS)

```
with TEXT_IO;
use TEXT_IO;
procedure MAIN is

   task COMM;
   task body COMM is separate;
   task TERMINAL;
   task body TERMINAL is separate;
   task WS_A;
   task body WS_A is separate;
   task WS_B is
        entry ENTRY_B (PAR 1: in INTEGER;
                       PAR 2: in out INTEGER);
   end WS_B;
   task body WS_B is separate;

end MAIN;

-------------------------------------------------------------
   --ESTABLISH 1ST PROCESS WITH TASK COMM
-------------------------------------------------------------

   separate (MAIN)
   task body COMM is
      . . .
   begin
      . . .
   end COMM;

-------------------------------------------------------------
   --ESTABLISH 2ND PROCESS WITH TASK TERMINAL
-------------------------------------------------------------

   separate (MAIN)
   task body TERMINAL is
      . . .
   begin
      . . .
   end TERMINAL;
```

FIGURE 104. BASIC ADA CODE USED TO PASS PARAMETERS BETWEEN PROCESSES

```
------------------------------------------------------------------
     --ESTABLISH 3RD PROCESS WITH TASK WS_A
------------------------------------------------------------------

     separate (MAIN)

     task body WS_A -- caller task
        . . .

     begin
        . . .

        WS_B. ENTRY_B (PAR1, PAR2);
        . . .
     end WS_A;

------------------------------------------------------------------
     --ESTABLISH 4TH PROCESS WITH TASK WS_B
------------------------------------------------------------------

     separate (MAIN)

     task body WS_B is -- Acceptor Task
        . . .
     begin
        . . .
        accept ENTRY_B (PAR1: in INTEGER; PAR2: in out INTEGER) do
        . . .
        end ENTRY_B;

     end WS_B;
```

FIGURE 104. (CONCLUDED)

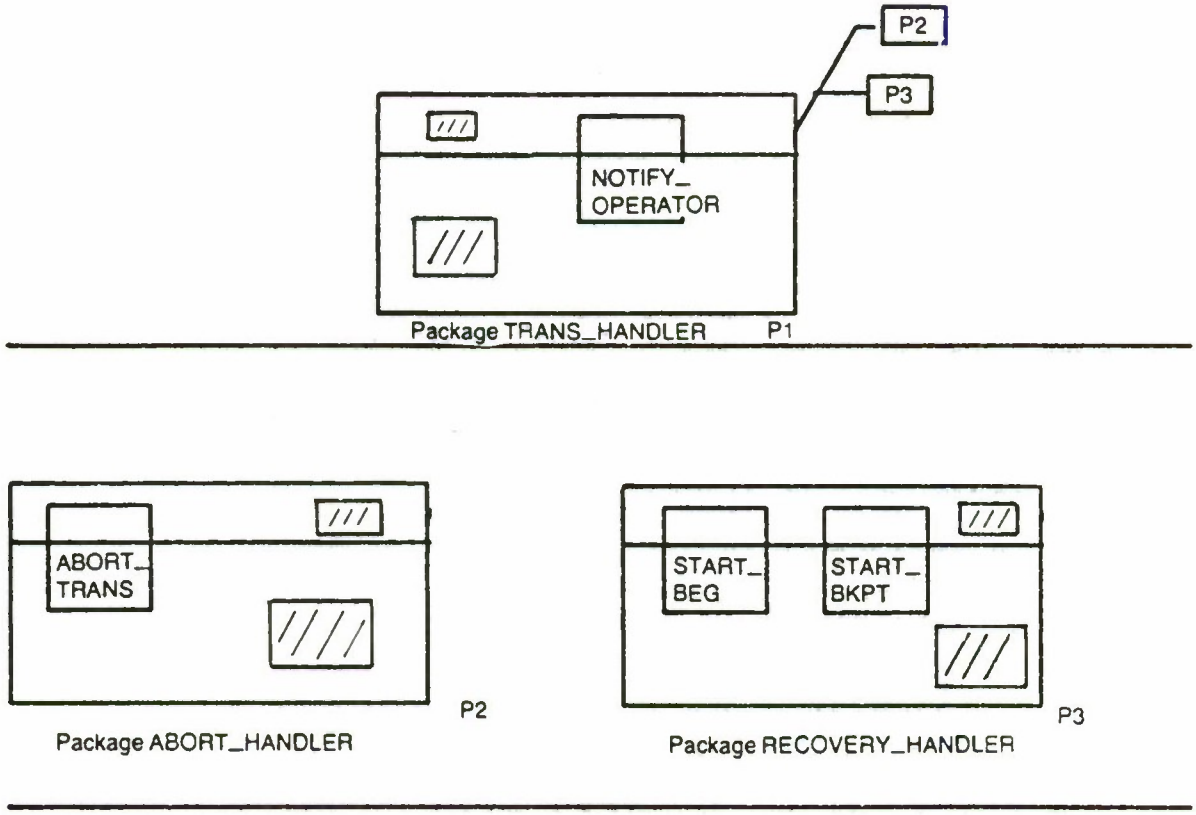


FIGURE 105 OBJECT IMPLEMENTATION IN AN ADA PACKAGE

After having been taught the basic technology for object implementation using SHARP, the student is then ready to learn the Ada source code associated with this technology. To complete Lesson 3, the student is taught the code used for object implementation.

For example, the basic code shown in Figure 106 (which is associated with Figures 98 and 99) could be used to commence this discussion and then more advanced code can be presented to explain code-related technical details associated with object implementation.

```
-------------------------------------------------------------
     --MAIN PROGRAM
-------------------------------------------------------------
     with TEXT_IO;
     use TEXT_IO;
     procedure MAIN is

        task COMM;
        task body COMM is separate;
        task TERMINAL;
        task body TERMINAL is separate;
        task WS_A;
        task body WS_A is separate;
        task WS_B is
             entry ENTRY_B (PAR 1: in INTEGER;
                            PAR 2; in out INTEGER);
        end WS_B;
        task body WS_B is separate;

     end MAIN;
-------------------------------------------------------------
     --ESTABLISH 1ST PROCESS WITH TASK COMM
-------------------------------------------------------------
     separate (MAIN)
     task body COMM is
        package ABORT_HANDLER_P2 is -----------------OBJECT P2
             procedure ABORT_TRANS (   ...   );
             . . .
        end ABORT_HANDLER_P2 is
        package body ABORT_HANDLER_P2 is
             procedure ABORT_TRANS (   ...   ) is separate;
        end ABORT_HANDLER_P2;

        package RECOVERY_HANDLER_P3 is --------------OBJECT P3
             procedure START_BEG (   ...   );
             procedure START_BKPT (  ...   );
             . . .
        end RECOVERY_HANDLER_P3;
        package body RECOVERY_HANDLER_P3 is
             procedure START_BEG (   ...   ) is separate;
             procedure START_BKPT (   ...   ) is separate;
        end RECOVERY_HANDLER_P3;
```

FIGURE 106. BASIC ADA CODE FOR OBJECT IMPLEMENTATIONS

```
        with ABORT_HANDLER_P2-----------------------OBJECT P1
        use ABORT_HANDLER_P2;
        with RECOVERY_HANDLER_P3;
        use RECOVERY_HANDLER_P3;
        package TRANS_HANDLER_P1 is
             procedure NOTIFY_OPERATOR (    ...    );
             . . .
        end TRANS_HANDLER-P1;
        package body TRANS_HANDLER_P1 is
             procedure NOTIFY_OPERATOR (    ...    ) is separate;
             . . .
        end TRANS_HANDLER_P1;
-------------------------------------------------------------
     --OBJECT P2 IMPLEMENTATION WITHIN PROCESS TASK COMM
-------------------------------------------------------------
     separate (MAIN.COMM. ABORT_HANDLER_P2)
     procedure ABORT_TRANS (       ...       ) is
        . . .
     begin
        . . .
     end ABORT_TRANS;
-------------------------------------------------------------
     --OBJECT P3 IMPLEMENTATION WITHIN PROCESS TASK COMM
-------------------------------------------------------------
     separate (MAIN.COMM. RECOVERY_HANDLER_P3)
     procedure START_BEG (       ...      ) is
        . . .
     begin
        . . .
     end START_BEG;
     procedure START_BKPT (  ...  ) is
        . . .
     begin
        . . .
     end START_BKPT;
-------------------------------------------------------------
     --OBJECT P1 IMPLEMENTATION WITHIN PROCESS TASK COMM
-------------------------------------------------------------
     separate (MAIN.COMM. TRANS_HANDLER_P1)
     procedure NOTIFY_OPERATOR (       ...      ) is
        . . .
     begin
        . . .
        if TRANS_ID = ABORT then
             ABORT_TRANS (      ...      )
        elsif TRANS_ID = RESTART then
             START_BEG (     ...     );
        else
             START_BKPT (      ...      );
        end if;
        . . .
     end NOTIFY_OPERATOR;
     begin
        . . .
     end COMM:
```

FIGURE 106. (CONCLUDED)

3.2.4 Lesson 4 - Object Data Structure

An object implementation has a local data structure shared by the program units used to implement the object. The local data structure is not accessible by other object implementations. A small visible data structure is also provided for defined types and variables established to facilitate parameter passing to other object implementations.

The student must be taught the contents of the data structure. SHARP Data Structure Diagrams can be used to help teach the technology for establishing data structures for an object implementation. As shown, declaration of defined types, constants and variables are represented by narrow geometric entities, either upright or slanting to the to the right or left. As illustrated in Figure 107, the visibility of the data structure entities are driven by where they are declared. Shaded data structure entities have been declared within the body of the package and, therefore, are not accessible by program units external to the package. Unshaded or partially shaded entities have been declared within the specification of the package and, therefore, are accessible to external program units. Entities that are partially shaded have been designated as **private** or **limited private**.

When the value of a **private** parameter is passed from one object implementation to another, the receiver object implementation can only use the parameter in assignment statements, statements testing for equality and operations defined within the package specification. If the passed parameter is **limited private**, assignment statements and statements testing for equality are no longer automatically available to the user.

Parameters passed between objects may be declared to be private or limited private in the specification of the package encapsulating the object implementation, especially if they are used in the formulation within the implementation.

The local data structure is established in the body of the encapsulating package. Therefore, entities of this data structure are not accessible by other object implementations. As illustrated in Figure 107, all the entities in this data structure are shaded, signifying the lack of accessibility outside the package.

As described in Section 2.4.4 of Chapter II, the object's local data structure may contain an array or record, and may contain discriminated, access or task types. In Ada, an array is defined to be a collection of entities of the same type. A defined type for an array is signified by the letters 'AR'. A record is defined to be a collection of entities of possibly two or more types, where the entities are determinable at compile time (as opposed to an array entity which can be established dynamically during run time). A defined type for a record is signified by the letter 'R'.

As an alternative to statically allocated data, Ada provides a mechanism for allocating variables dynamically during program execution. Since the storage locations used for dynamic variables are not determined in advance, they cannot be referenced by a name but must instead be referenced

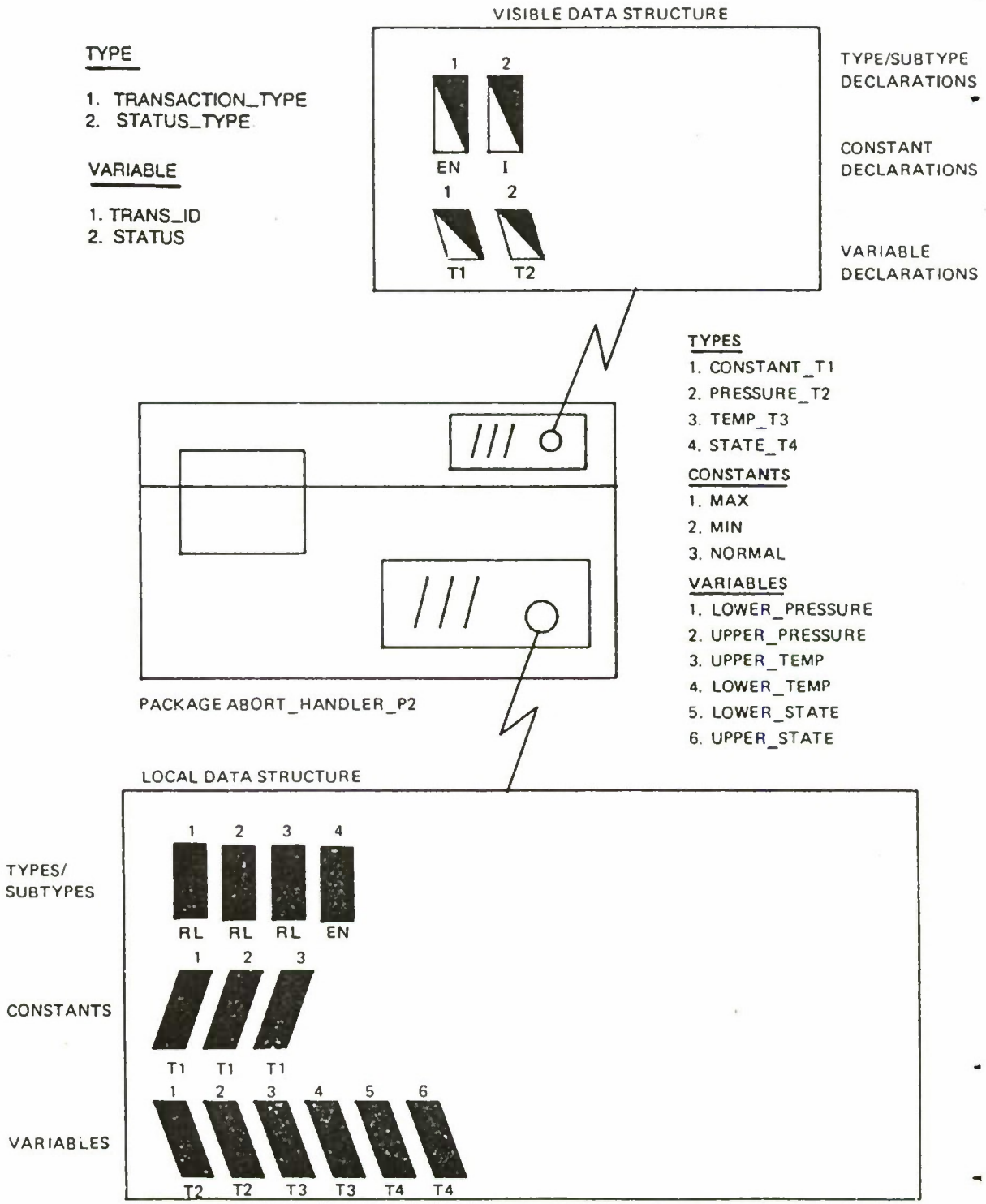


FIGURE 107. DATA STRUCTURE FOR OBJECT IMPLEMENTATIONS

indirectly via a so-called access type. Unknown amounts of data can be handled by dynamically allocating a storage to each new datum when it is received. In this way, complex data structures can be built with components dynamically allocated. An access type is represented by an upright narrow rectangle with the letters "AC" underneath it.

A task type is formed when the keyword task is followed by the keyword type. Elaboration of the corresponding task body defines what a task of that type does. It does not cause a task to be activated. Rather, tasks are activated separately by declaring variables of the task type. A task type is represented by an upright narrow rectangle with the letter "T" underneath it.

In general, the type of a variable or constant can be represented in SHARP Data Structure Diagrams as follows:

- If the type is predefined, then the first letter of the type (e.g., I for INTEGER) is placed under the geometric representation of the variable or constant.

- If the type is defined, the letter "T" followed by the type glossary number is placed under the geometric representation of the variable or constant.

Having been taught the basic technology for object data structures with the aid of SHARP Data Structure Diagrams the student is then ready to learn the Ada source code associated with this technology. To complete Lesson 4, the student is taught the code used to establish a data structure. For example, the basic code shown in Figure 108 (which corresponds to Figure 107) could be used to commence this discussion. Then more advanced code can be presented to explain code-related technical details associated with data structure implementation.

3.2.5 Lesson 5 - Interaction of Object Implementations

Parameters can be passed from one object to another. If object implementations are encapsulated in Ada tasks, the task rendezvous is used to accomplish the object interaction. The teaching of task rendezvous was discussed in Lesson 2 (Section 3.2.2), in the context of process interaction. Object implementations hidden in Ada packages interact using communicating subprograms declared in the packages specification. Item b of Figure 99 uses a SHARP Invocation Diagram to illustrate communication between object implementations, for the set of object implementations shown in Item a of Figure 99.

The specific parameters passed between object implementations can be represented by SHARP Data Flow Diagrams, as shown in Figure 109. As shown, three modes of parameter passing are possible, as is the case with task rendezvous. Specifically, the modes are as follows:

```
------------------------------------------------------------------
     --MAIN PROGRAM
------------------------------------------------------------------
     with TEXT_IO;
     use TEXT_IO;
     procedure MAIN is

        task COMM;
        task body COMM is separate;
        task TERMINAL;
        task body TERMINAL is separate;
        task WS_A;
        task body WS_A is separate;
        task WS_B is
             entry ENTRY_B (PAR1: in INTEGER;
                            PAR2: in out INTEGER);
             end WS_B;

        task body WS_B is separate;

     end MAIN;
------------------------------------------------------------------
     --ESTABLISH 1ST PROCESS WITH TASK COMM
------------------------------------------------------------------
     separate (MAIN)
     task body COMM is

     package ABORT_HANDLER_P2 is --------------------OBJECT P2

        --VISIBLE DATA STRUCTURE
        type TRANSACTION_T1 is private;
        type STATUS_T2 is private;
        TRANS_ID:  TRANSACTION_T1;
        STATUS:  STATUS_T2;
        private
             type TRANSACTION_T1 is (RESTART, BREAKPOINT, ABORT);
             type STATUS_T2 is (RESTART_COMPLETE, NO_RESTART,
                BREAKPOINT_COMPLETE, NO_BREAKPOINT, ABORT_COMPLETE,
                NO_ABORT);
        procedure ABORT_TRANS (   ...   );

        . . .


     end ABORT_HANDLER_P2;
```

FIGURE 108. BASIC ADA CODE FOR THE DATA STRUCTURE OF AN OBJECT IMPLEMENTATION

```
package body ABORT_HANDLER_P2 is

   --LOCAL DATA STRUCTURE
   --types
   type CONSTANT_T1 is INTEGER range 1 ... 100;
   type PRESSURE_T2 is INTEGER range 22 ... 35;
   type TEMP_T3 is INTEGER range 1 ... 120;
   type STATE_T4 is (ON, OFF, PENDING);
   -- constants
   MAX: CONSTANT_T1:= 90;
   MIN: CONSTANT_T1:= 70;
   NORMAL: CONSTANT:= 80;
   --variables
   LOWER_PRESSURE: PRESSURE_T2;
   UPPER_PRESSURE: PRESSURE_T2;
   UPPER_TEMP: TEMP_T3;
   LOWER_TEMP: TEMP_T3;
   LOWER_STATE: STATE_T4;
   UPPER_STATE: STATE_T4;
        . . .
   procedure ABORT_TRANS (    ...    ) is separate;
end ABORT_HANDLER_P2;
        o
        o
        o
```

FIGURE 108. (CONCLUDED)

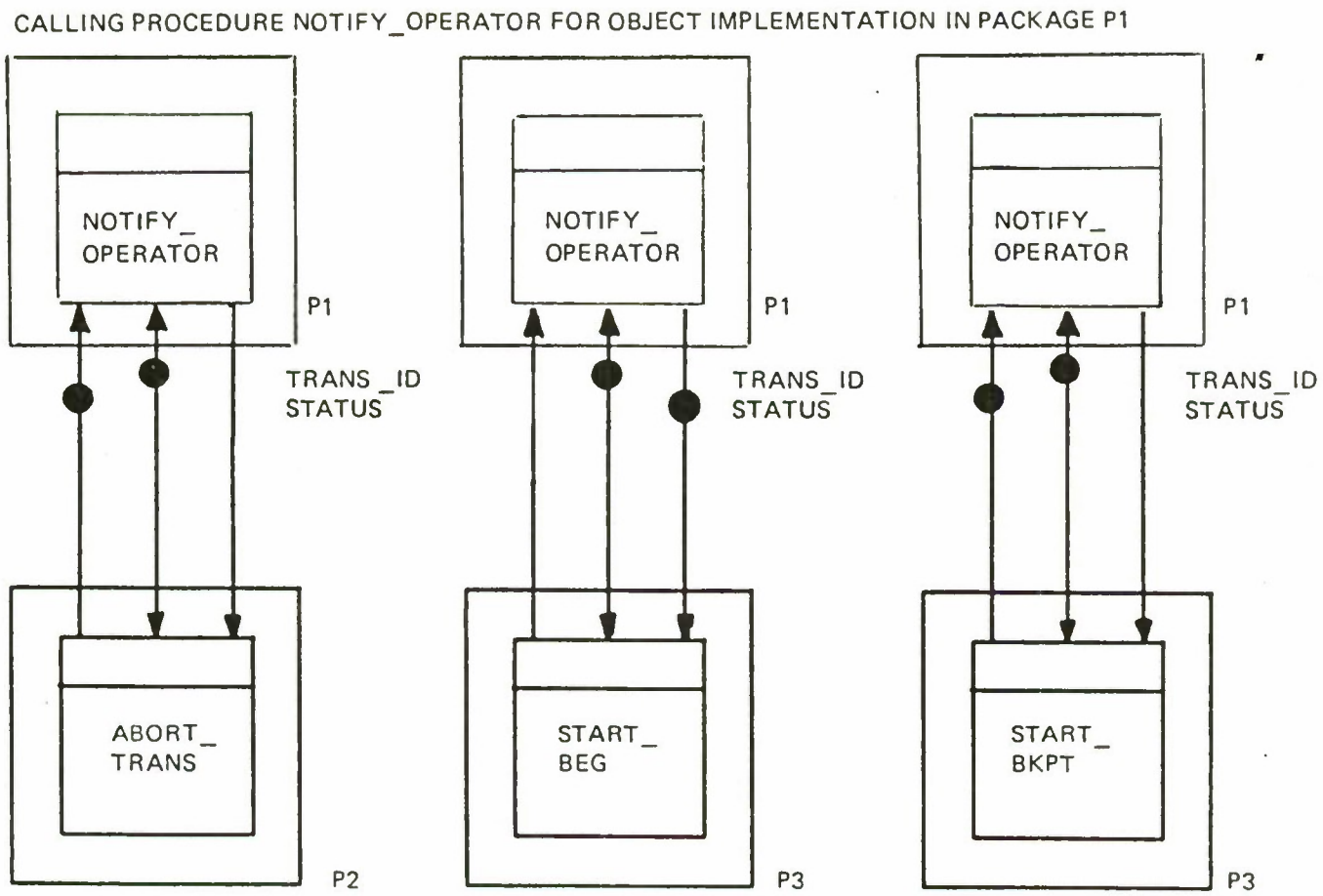


FIGURE 109. DATA FLOW BETWEEN SUBPROGRAMS

- 'in' (i.e., the value of a parameter is received and not modified)
- 'out' (i.e., the value of a parameter is created and exported)
- 'in out' (i.e., the value of a parameter is received, modified and exported).

Having been taught the basic technology for object interaction using SHARP Invocation Diagrams and Data Flow Diagrams, the student is then ready to learn the Ada source code associated with this technology. To complete Lesson 5, the student is taught the code used for the interaction of object implementations. For example, the basic code shown in Figure 110 (which corresponds to Figure 109) could be used to commence this discussion. Then more advanced code can be presented to explain code-related technical details associated with data passage between subprograms. In this example, values for the variable TRANS_ID are passed between the calling subprogram and the called subprograms using the 'in out' mode. Also, values for the variable STATUS are sent from the called subprogram to the caller using the 'out' mode.

```
------------------------------------------------------------------
     --MAIN PROGRAM
------------------------------------------------------------------
     with TEXT_IO;
     use TEXT_IO;
     procedure MAIN is

        task COMM;
        task body COMM is separate;
        task TERMINAL is separate;
        task body TERMINAL is separate;
        task WS_A;
        task body WS_A is separate;
        task WS_B;
        task body WS_B is separate;
             entry ENTRY_B (PAR1: in INTEGER;
                            PAR2: in out INTEGER);
             end WS_B;

        task body WS_B is separate;

      end MAIN;
```

FIGURE 110. BASIC ADA CODE FOR PARAMETER PASSING BETWEEN SUBPROGRAMS

```
------------------------------------------------------------
     --ESTABLISH 1ST PROCESS WITH TASK COMM
------------------------------------------------------------
     separate (MAIN)
     task body COMM is

     package ABORT_HANDLER_P2 is --------OBJECT P2 IMPLEMENTATION

        --VISIBLE DATA STRUCTURE
        type TRANSACTION_T1 is private;
        TRANS_ID:  TRANSACTION_T1;
        STATUS:  STATUS_T2;
        private
             type TRANSACTION_T1 is (RESTART, BREAKPOINT, ABORT);
             type STATUS_T2 is (RESTART_COMPLETE, NO_RESTART,
                BREAKPOINT_COMPLETE, NO_BREAKPOINT, ABORT_COMPLETE,
                NO_ABORT);
        procedure ABORT_TRANS (TRANS_ID: in out TRANSACTION_T1;
                               STATUS: out STATUS_T2);

             . . .

        end ABORT_HANDLER_P2;


        package body ABORT_HANDLER_P2 is

           --LOCAL DATA STRUCTURE
           type CONSTANT_T1 is INTEGER range 1 ... 100;
           type PRESSURE_T2 is INTEGER range 22 ... 35;
           type TEMP_T3 is INTEGER range 1 ... 120;
           type STATE_T4 is (ON, OFF, PENDING);
           -- constant
           MAX: CONSTANT_T1:= 90;
           MIN: CONSTANT_T1:= 70;
           NORMAL: CONSTANT:= 80;
           --variables
           LOWER_PRESSURE: PRESSURE_T2;
           UPPER_PRESSURE: PRESSURE_T2;
           UPPER_TEMP: TEMP_T3;
           LOWER_TEMP: TEMP_T3;
           LOWER_STATE: STATE_T4;
           UPPER_STATE: STATE_T4;
           --
           procedure ABORT_TRANS (TRANS_ID: in out TRANSACTION_T1;
                                  STATUS: out STATUS_T2) is separate;
                . . .
        end ABORT_HANDLER_P2;
```

FIGURE 110. (CONTINUED)

```
package RECOVERY_HANDLER_P3 is ----------OBJECT P3 IMPLEMENTATION
     procedure START_BEG (TRANS_ID: in out TRANSACTION_T1;
                          STATUS: out STATUS_T2) is separate;
     procedure START_BKPT (TRANS_ID: in out TRANSACTION_T1;
                           STATUS: out STATUS_T2) is separate;
          . . .
   end RECOVERY_HANDLER_P3;
   package body RECOVERY_HANDLER_P3 is
     procedure START_BEG (TRANS_ID: in out TRANSACTION_T1;
                          STATUS: out STATUS_T2) is separate;
          . . .
   end RECOVERY_HANDLER_P3;
   --
   with ABORT_HANDLER_P2;
   use ABORT_HANDLER_P2;
   with RECOVERY_HANDLER_P3;
   use RECOVERY_HANDLER_P3;
package TRANS_HANDLER_P1 is--------------OBJECT P1 IMPLEMENTATION
          . . .
     procedure NOTIFY_OPERATOR (STATUS: in STATUS_T2;
                                TRANS_ID: in out TRANSACTION_T1)
                                is separate;
   end TRANS_HANDLER_P1;
   package body TRANS_HANDLER_P1 is
          . . .
     procedure NOTIFY_OPERATOR (STATUS: in STATUS_T2;
                                TRANS_ID: in out TRANSACTION_T1)
                                is separate;
          . . .
   end TRANS_HANDLER_P1;


     o
     o
     o
   separate (MAIN.COMM)
   procedure NOTIFY_OPERATOR is
     . . .
   begin
     . . .

     ABORT_TRANS (TRANS_ID=>TRANSA, STATUS=>STATUSA);
     . . .

     START_BEG (TRANS_ID=>TRANSB, STATUS=>STATUSA);

     . . .


     START_BKPT(TRANS_ID=>TRANSC, STATUS=>STATUSC);

     . . .

   end NOTIFY_OPERATOR;
```

FIGURE 110. (CONCLUDED)

3.2.6 Lesson 6 - Abstraction Internal to an Object Implementation

An object's requirements are typically sufficiently complex that abstraction must be introduced into the design of the object implementation, prior to code implementation. For example, a small and easily comprehended portion of the requirements can be assigned to one level for implementation, while the rest of the requirements will be assigned to other levels. At each of the other levels, the abstraction process can be repeated.

With Ada, the implementation of detail excluded at one level is passed to the bodies of called subprograms. The bodies of the called program units are implemented in the same manner. Therefore, these bodies are also constrained to an easily understood amount of detail, with yet lower detail moved again to other called program units. In this way, a series of nested program units are used to spread implementation detail into levels of abstraction.

This abstraction process can be clearly represented by SHARP Hierarchy Diagrams and SHARP Invocation diagrams. For example, the Hierarchy diagram shown in Figure 111 represents nested program units assigned to levels. The subject program unit (i.e., a subprogram visible in an Ada package used to encapsulate an object implementation) is assigned to Level 1. Program units directly nested within the subject program unit are assigned to Level 2. In general, a program unit directly nested within a program unit at Level n is assigned to Level n+1.

The related Invocation Diagram is shown in Figure 112. It provides information relevant to the sequence in which program units will execute. For example, a call to a subprogram may be dependent upon certain conditions having been met. As shown in Figure 112, the existence of conditional calls is indicated by a tilde on the arrow representing the potential program unit call.

Having been taught the basic technology for the internal abstraction of an object implementation using SHARP Hierarchy and Invocation Diagrams, the student is then ready to learn the Ada source code associated with this technology. To complete Lesson 6, the student is taught this code. For example, the basic code shown in Figure 113 (which corresponds to Figures 111 and 112) could be used to commence this discussion. Then more advanced code can be presented to explain code-related technical details associated with abstraction internal to an object.

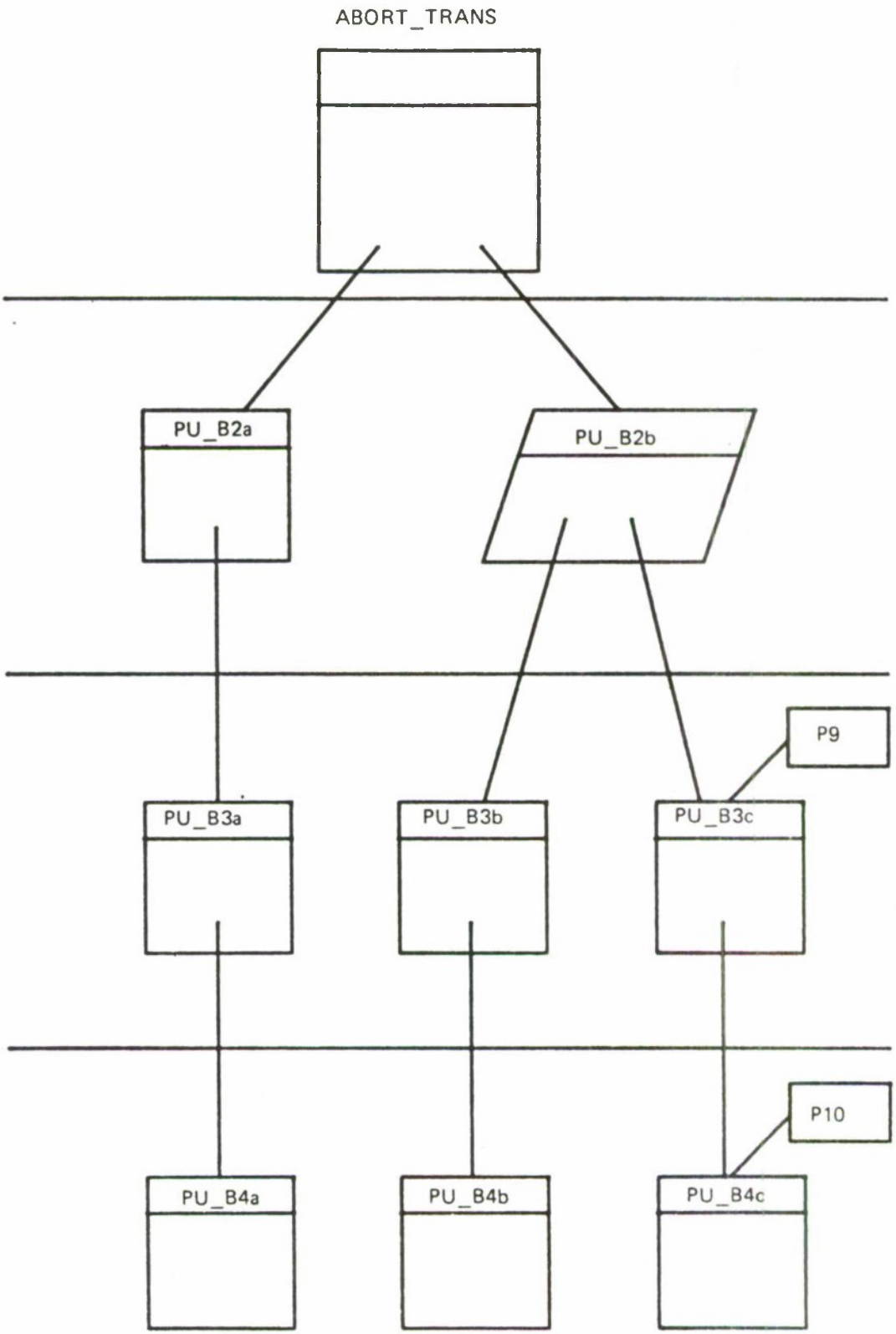


FIGURE 111. HIERARCHY OF PROGRAM UNITS INTERNAL TO AN OBJECT IMPLEMENTATION

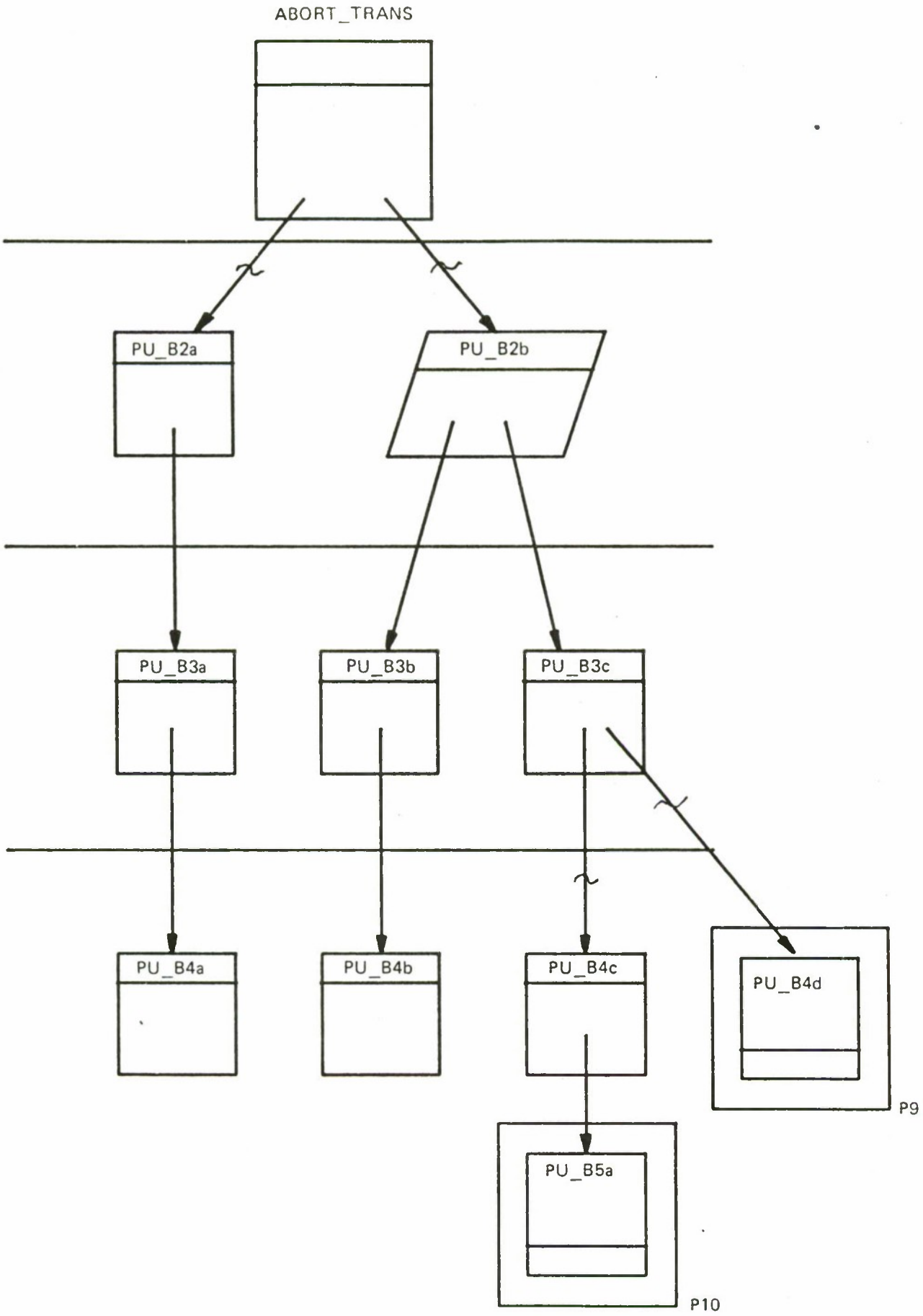


FIGURE 112. INVOCATION OF PROGRAM UNITS INTERNAL TO AN OBJECT IMPLEMENTATION

```
                    o
                    o
                    o

-------------------------------------------------------------------
     OBJECT P2 IMPLEMENTATION
-------------------------------------------------------------------
     separate (MAIN.COMM.ABORT_HANDLER)
     procedure  ABORT_TRANS (TRANS_ID: in out TRANSACTION_T1;
                             STATUS: out STATUS_T2) is
        procedure PU_B2a is separate;
        task PU_B2b is
             entry C;
             entry D;
        end task PU_B2b;
        task body PU_B2b is separate

     begin
             . . .
     end ABORT_TRANS;

-------------------------------------------------------------------
     --OBJECT P2, Level 2, Unit a
-------------------------------------------------------------------
     separate (MAIN.COMM.ABORT_HANDLER.ABORT_TRANS)
     procedure PU_B2 is

        procedure PU_B3a is separate;

     begin -- PU_B2a

        . . .

     end PU_B2a;

-------------------------------------------------------------------
     --OBJECT P2, Level 2, Unit b
-------------------------------------------------------------------
     separate (MAIN.COMM.ABORT_HANDLER.ABORT_TRANS)
     task body PU_B2b is

        procedure PU_B3b is separate;
        procedure PU_B3c is separate;

     begin -- PU_B2b

             . . .

     end PU_B2b;
```

FIGURE 113. BASIC ADA CODE FOR ABSTRACTION WITH PROGRAM UNITS INTERNAL TO AN OBJECT IMPLEMENTATION

```
------------------------------------------------------------------
     --OBJECT P2, Level 3, Unit a
------------------------------------------------------------------
     separate (MAIN.COMM.ABORT_HANDLER.ABORT_TRANS.PU_B2a)
     procedure PU_B3a is

        procedure PU_B4a is separate;

     begin -- PU_B3a

        . . .

     end PU_B3a;
------------------------------------------------------------------
     --OBJECT P2, Level 3, Unit b
------------------------------------------------------------------
     separate (MAIN.COMM.ABORT_HANDLER 2.ABORT_TRANS.PU_B2b)
     procedure PU_B3b is

        procedure PU_B4b is separate;

     begin -- PU_B3b

        . . .

     end PU_B3b;
------------------------------------------------------------------
     --OBJECT P2, Level 3, Unit c
------------------------------------------------------------------
     with P9;
     use P9;
     separate (MAIN.COMM.ABORT_HANDLER.ABORT_TRANS.PU_B2b)
     procedure PU_B3c is

        procedure PU_B4c is separate;

     begin -- PU_B3c

        . . .

     end PU_B3c;
------------------------------------------------------------------
     --OBJECT P2, Level 4, Unit a
------------------------------------------------------------------
     separate (MAIN.COMM.ABORT_HANDLER 2.ABORT_TRANS.PU_B3a)
     procedure PU)_B4a is

     begin -- PU_B4a

        . . .

     end PU_B4a;
------------------------------------------------------------------
```

FIGURE 113. (CONTINUED)

```
------------------------------------------------------------------------
     --OBJECT P2, Level 4, Unit b
------------------------------------------------------------------------
     separate (MAIN.COMM.ABORT_HANDLER.ABORT_TRANS.PU_B3b)
     procedure PU_B4b is

     begin -- PU_B4b

        . . .

     end PU_B4b;
------------------------------------------------------------------------
     --OBJECT P2, Level 4, Unit c
------------------------------------------------------------------------
     with P10;
     use P10;
     separate (MAIN.COMM.ABORT_HANDLER.ABORT_TRANS.PU_B3c)
     procedure PU_B4c is

     begin -- PU_B4c

        . . .

     end PU_B4c;

------------------------------------------------------------------------
```

FIGURE 113. (CONCLUDED)

3.2.7 Lesson 7 - Implementing Processing Internal to Program Units

To establish the design for requirements abstracted to levels of program units, SHARP utilizes annotated pseudo code to represent the bodies of the program units. SHARP criteria include general standards for the pseudo code and its annotation. The standards require the pseudo code to account for the following:

- Logic and decisions
- Algorithms
- Program unit calls and I/O
- Generic instantiation
- Exception handling

The standards require that these entities must be presented using certain Ada key words and annotation, as described in Section 2.5 of Chapter II. The annotated pseudo code can be used to help teach Ada detail, as we reach the bottom of the top-down description of Ada.

For example, consider Ada statements used for logic and decisions. The Ada **if** and **case** statements are used to provide conditional control (i.e., the selection of one of a number of alternate actions).

The **if** statement selects a course of action depending upon the truth value of one or more conditions. In Ada, there are three basic forms of the if statement:

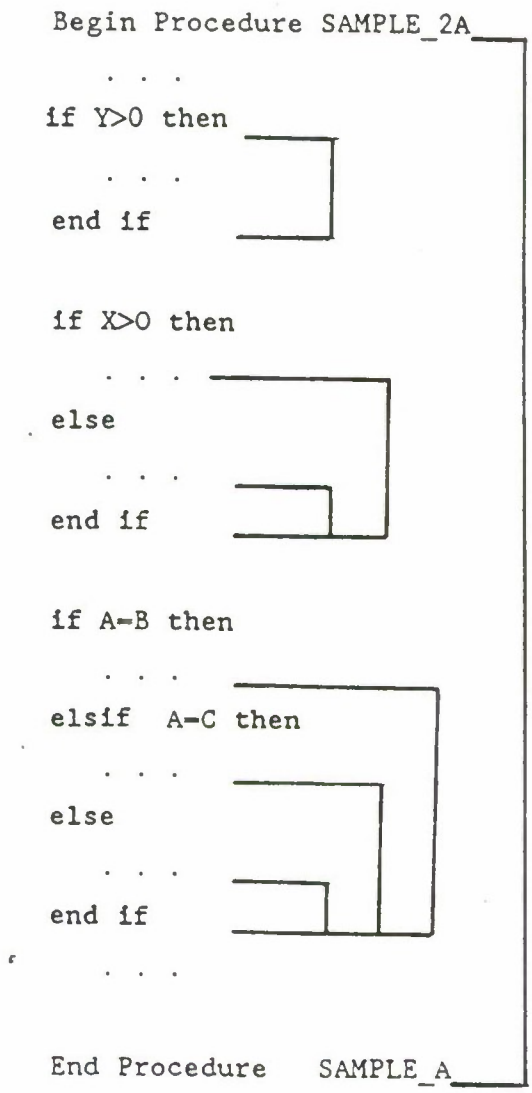
- **if**-then
- **if**-then-else
- **if**-then-elsif

In each case , the **if** statement is terminated with an **end if** clause. SHARP can be used to help teach the **if** statements, using annotated pseudo code (e.g., as shown) in item a of Figure 114.

The **case** statement provides for the selection one of a set of multiple alternative actions, as a function of the value of an expression. SHARP can be used to help teach the **case** statement using annotated pseudo code (e.g., as shown in item b of Figure 114).

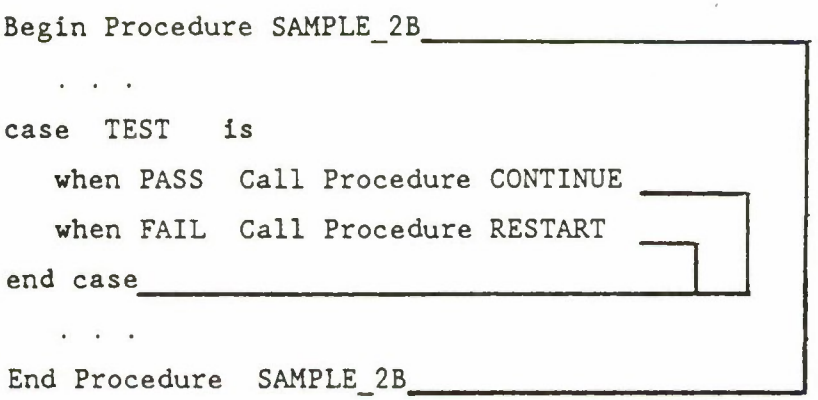
Repetitive execution of action is accomplished in Ada using the **loop** statement. The basic loop is accomplished using a **loop** and **end loop** statement. To leave a loop, an **exit** statement is used. SHARP can be used to represent the loop statement with annotated pseudo code (e.g., as illustrated in item c of Figure 114).

To repeat a loop for a specific number of times, the basic loop can be preceded by a **for** iteration clause. Also, another form of iteration can be accomplished with the **while** statement, whereby a sequence of statements is repeated as long as some condition is true. SHARP can be used to help teach the **for** and **while** statements with annotated pseudo code (e.g., as shown in item d of Figure 114).

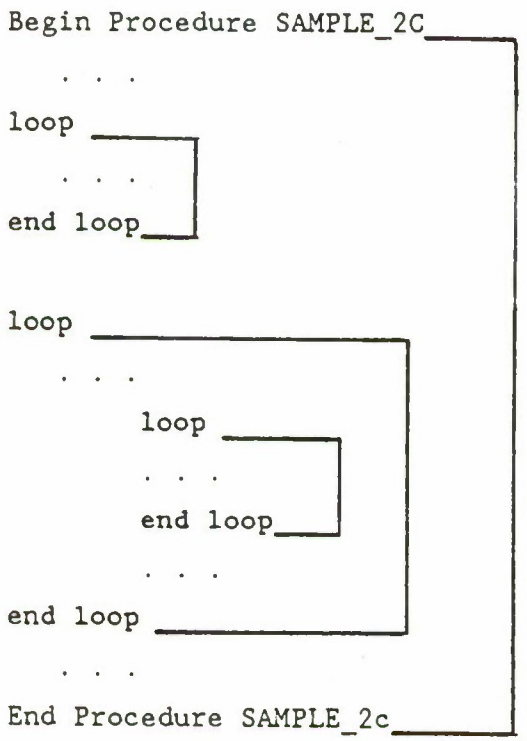


(a) USE OF "IF" STATEMENTS

FIGURE 114. ANNOTATED PSEUDO CODE FOR REPRESENTING LOGIC AND DECISIONS WITHIN THE BODIES OF PROGRAM UNITS

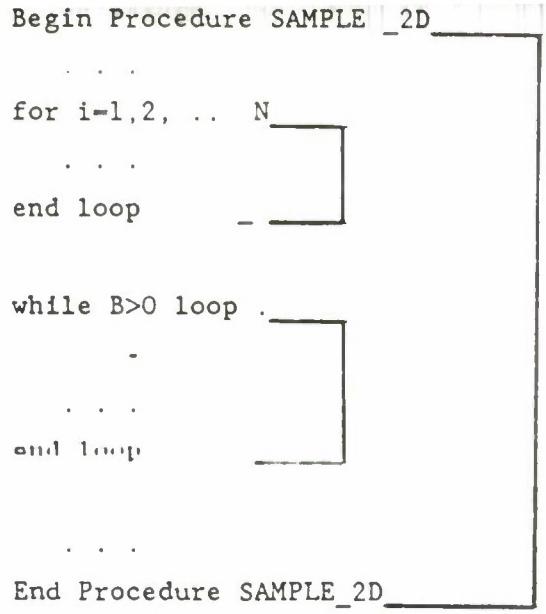


(b) USE OF "CASE" STATEMENT

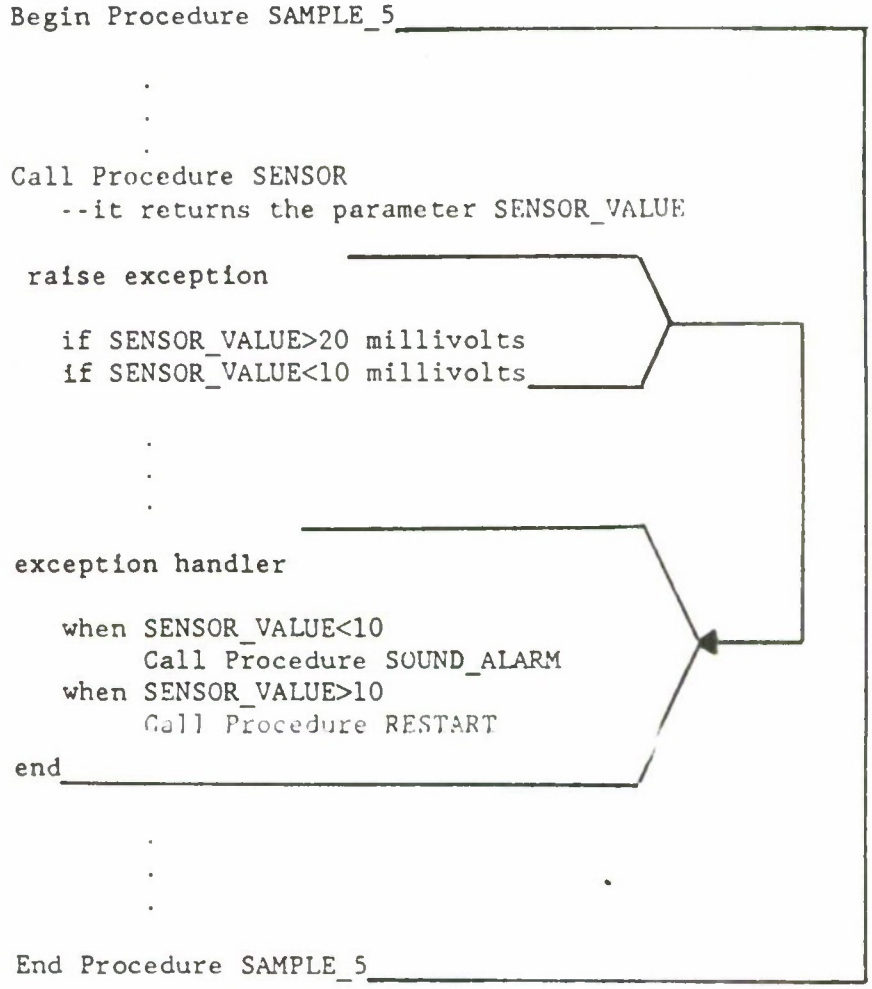


(c) USE OF "LOOP" STATEMENT

FIGURE 114. (CONTINUED)



(d) USE OF THE "FOR" AND "WHILE" STATEMENTS



(e) REPRESENTING EXCEPTIONS

FIGURE 114. (CONCLUDED)

As another example, consider exception handling with Ada. Ada provides an explicit mechanism for detecting and responding to an anomaly. The anomaly, for example, could be associated with erroneous input data or overflow conditions. SHARP criteria require that the design of a program unit's body must specify the detection of the anomaly and the course of action taken after the occurrence of the anomaly.

In Ada, the detection of the anomaly causes normal program execution to be suspended and control transferred to an exception handler. Once the exception handler has completed its processing, control transfers to code following the exception handler code.

Figure 114 provides an example of annotated SHARP pseudo code used to specify the detection of an anomaly and action taken in an exception handler. SHARP criteria requires that (a) the pseudo code for the anomaly detection must be introduced by the key words 'raise exception' and must be bracketed as shown in the figure, (b) the pseudo code for the action taken upon occurrence of the anomaly must be introduced by the key words 'exception handler,' must be concluded with the key word 'end,' and must be bracketed as shown in the figure, and (c) an arrow must point from the bracket enclosing pseudo code for the anomaly detection to the bracket enclosing pseudo code for the exception handler.

Having been taught the basic technology for entities implemented in the body of a program unit through the use of annotated SHARP pseudo code, the student is then ready to learn the Ada source code associated with this technology. To complete Lesson 7, the student is taught this code. For example, the code shown in Figure 116, which implements the SHARP pseudo code shown in Figure 115, can be used as the basis for this lesson. Additional code can be presented when needed to explain details associated with entities implemented in the bodies of program units.

3.2.8 Lesson 8 - Use of Existing Ada Packages and Packages of Common Program Units

The basic SHARP pictograph for a package can be used to represent existing Ada packages and Ada packages established to encapsulate common program units (i.e., program units accessed by two or more other program units). For example, in Ada, the predefined package TEXT_IO is used to facilitate input and output. (The use of TEXT_IO is explained in Chapter 15 of An Introduction to Ada. [2])

Also, it is expected that software contractors will utilize existing in-house packages in the implementation of large and complex Ada computer programs. It is anticipated that in time contractors will build a library of packages containing such things as hardware drivers, communication protocols, low level I/O, mathematical functions and special purpose application routines (e.g., Fast Fourier Transform).

In addition, packages will be developed to house program units common within an object implementation, or in certain cases, common to more than one object implementation.

With SHARP, such packages are represented in the manner shown in Figure 117. These diagrams identify visible program units declared in a package's specification but the diagrams do not identify program units declared in the package's body. Rather, SHARP uses its hierarchy and invocation diagram to identify these program units, as described in Section 3.2.6 in the context of abstraction internal to an object implementation.

As shown in Figure 117, SHARP abstracts represent a subject package's access to existing packages, or packages containing common program units, using the Ada 'with' clause. Each of the accessed packages is represented by a small rectangle encapsulating its name, and a straight line drawn to the specification or body (as appropriate) of the accessing package.

Having been taught the basic technology for accessing existing Ada packages and packages of common program units using SHARP, the student is then ready to learn Ada source code for this technology. To complete Lesson 8, the student is taught this code. For example, the code shown in Figure 118 can be used as the basis for this lesson.

3.2.9 Lesson 9 - Ada at the Bottom and Course Completion

After completing the first eight lessons, the student has been presented a wide range of Ada design and implementation factors in a top-down manner. As the student learned the technology of an object-oriented implementation using SHARP, the related Ada syntax and other code detail was introduced on an as needed basis. The final lesson summarizes the low level detail that has been introduced and introduces detail not yet mentioned. This final lesson can act as a bottom-up summary of what has been taught top-down.

To complete the Ada course, it is important to summarize important concepts associated with the object-oriented approach to Ada software implementation and the cost ramifications of this approach. The need for comprehensive testing of individual objects must be stressed and basic steps in completing such testing introduced. The object-oriented Ada software development will be especially cost effective if individual object implementations are relatively error free so that object integration proceeds smoothly, as described in Chapter VI.

4. CHAPTER SUMMARY

This chapter describes the application of SHARP in teaching object-oriented Ada technology to project managers and system engineers. It describes how software engineers and programmers can be taught Ada in a top-down manner using SHARP.

It is important that appropriate Ada instruction is given to project managers and system engineers, as well as software engineers and programmers. Of course, the level of detail and emphasis of this instruction must vary in depth depending upon the target audience. However, the gap in knowledge between contractor team members must be kept small so that effective communication can take place during system acquisition.

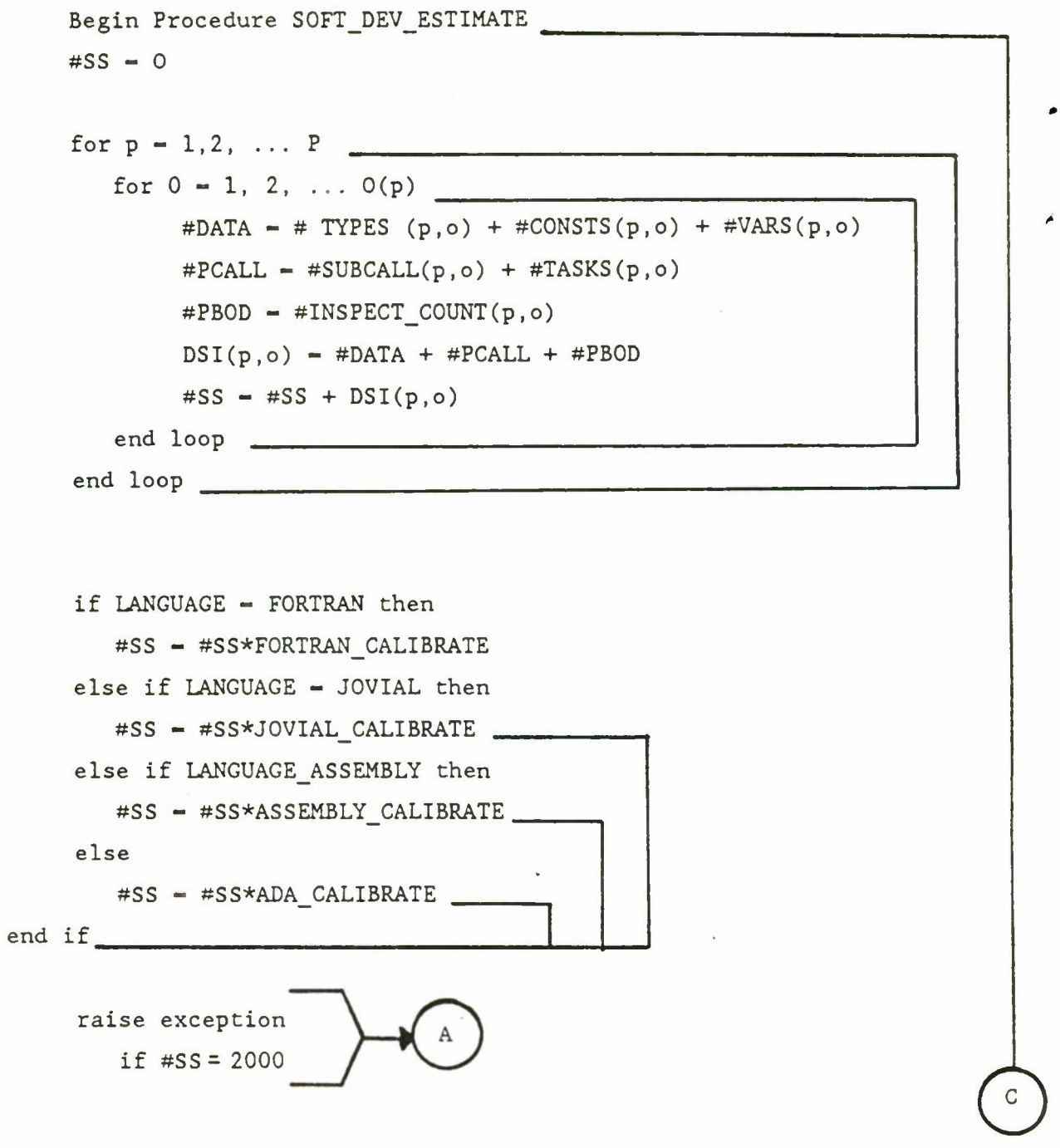


FIGURE 115. EXAMPLE OF ANNOTATED PSEUDO CODE

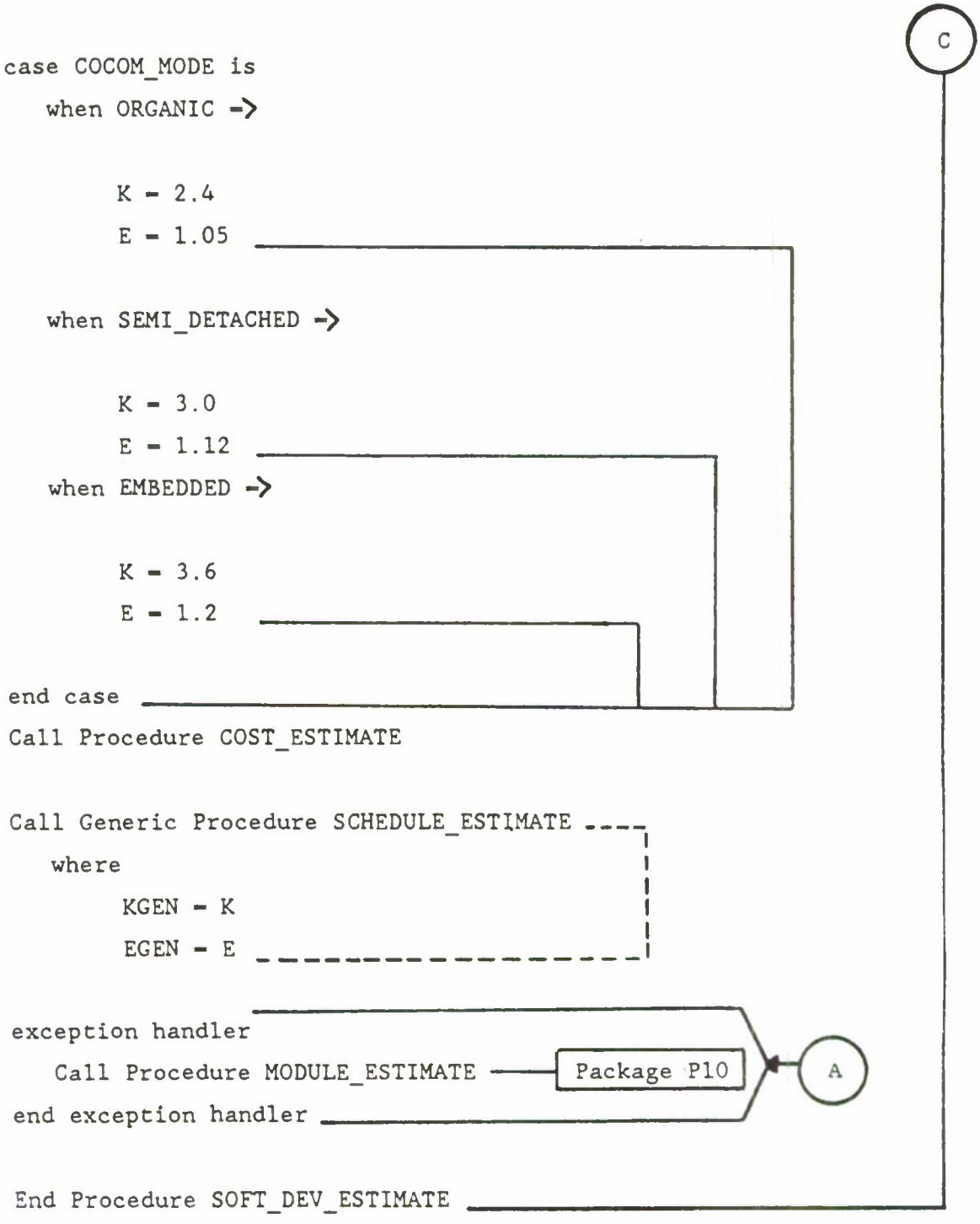


FIGURE 115. (CONCLUDED)

```
with P10;
use P10;
procedure SOFT_DEV_ESTIMATE(    ...    ) is
   TOO_SMALL:exception -- declare an exception
   procedure  ESTABLISH_TIME is new SCHEDULE_ESTIMATE
                  (KGEN = K, EGEN = E);--generic instantiation
begin
NO_SS := 0
for P in 1 .. CAP_P
   for O in 1 .. CAP_O

   NO_DATA:= NO_TYPES(P,O) + NO_CONSTS(P,O) + NO_VARS(P,O);
   NO_PCALL:= NO_SUBCALL(P,O) + NO_TASKS(P,O);
   NO_PBOD:= NO_INSPECT_COUNT(P,O);
   DSI(P,O):= NO_DATA + NO_PCALL + NO_PBOD;
   NO_SS:= NO_SS + DSI(P,O);

  end loop;
end loop;

if LANGUAGE = FORTRAN then
   NO_SS:= NO_SS*FORTRAN_CALIBRATE
elsif  LANGUAGE = JOVIAL then
   NO_SS:= NO_SS*JOVIAL_CALIBRATE
elsif  LANGUAGE = ASSEMBLY then
   NO_SS:= NO_SS*ASSEMBLY_CALIBRATE
else
   NO_SS:= NO_SS*ADA_CALIBRATE
end if;
```

FIGURE 116. EXAMPLE OF ADA CODE OF A PROGRAM UNIT'S BODY

```
if NO_SS > TWO_THOUSAND then --raise the exception
   raise TOO_SMALL;
end if;

case COCOMO_MODE is
   when ORGANIC =>

        K:= 2.4    ;
        E:= 1.05   ;
   when SEMI_DETACHED =>
        K:= 3.0    ;
        E:= 1.12   ;
   when EMBEDDED =>

        K:= 3.6    ;
        E:= 1.2    ;
end case;

COST_ESTIMATE (    ...    );
SCHEDULE_ESTIMATE (    ...    );  -- execution of generic subprogram


exception   -- handle the exception
   when  TOO_SMALL =>
           MODULE_ESTIMATE;
end;
```

FIGURE 116. (CONCLUDED)

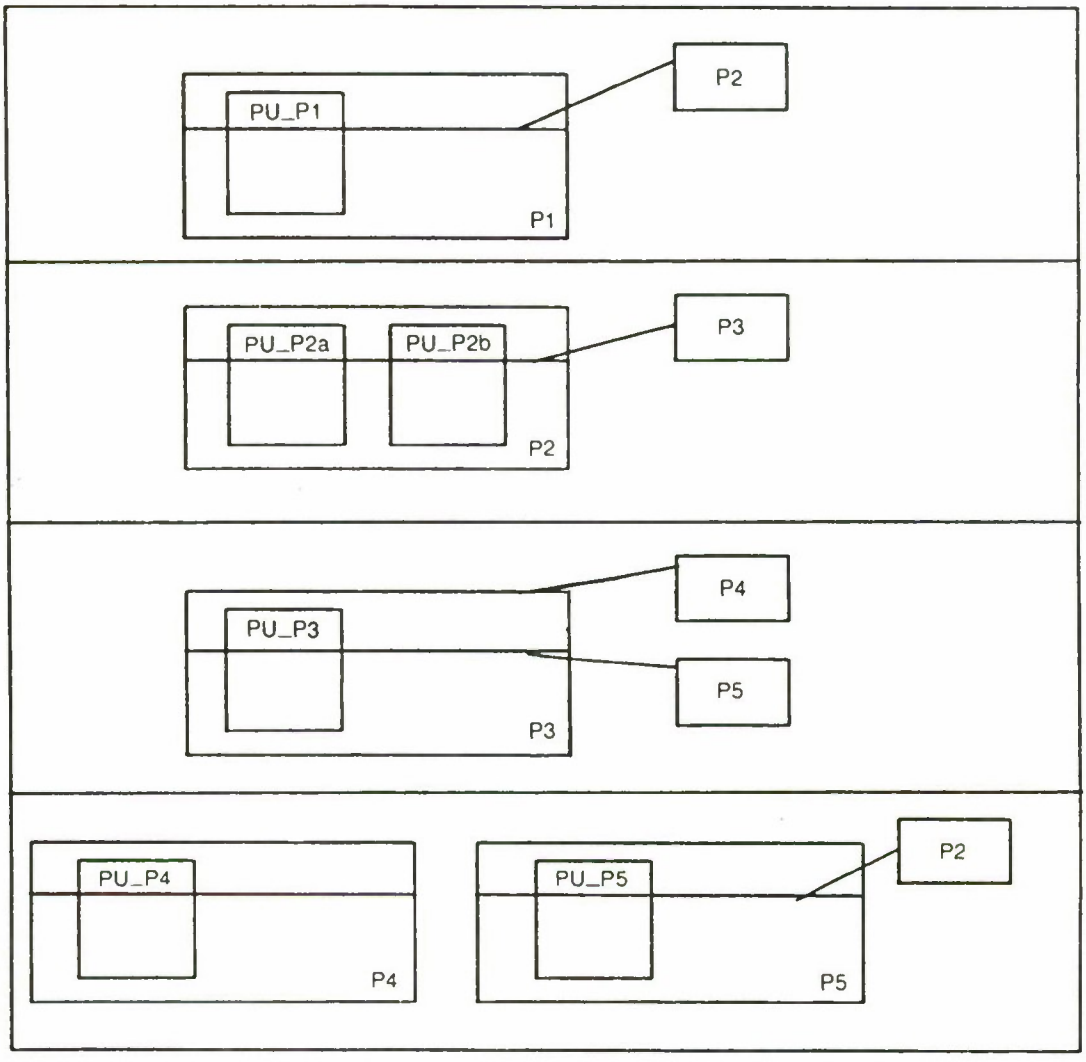


FIGURE 117. LAYERS OF ADA PACKAGES

```
package P1 is
        . . .
        procedure PU_P1 (    ...    );
        . . .
end P1;
with P2;
use P2;

package body P1 is
        . . .
end P1;
```

```
package P2 is
        . . .
        procedure PU_P2a (    ...    );
        procedure PU_P2b (    ...    );
        . . .
end P2;
with P3;
use P3;
package body P2 is
        . . .
end P2;
```

FIGURE 118. BASIC ADA CODE FOR ACCESSING PROGRAM UNITS WITHIN ADA PACKAGES

```
with P4;
use P4;
package P3 is
   . . .
   procedure PU_P3 (    ...    );
   . . .
end P3;
with P5;
use P5;
package body P3 is
   . . .
end P3;
-------------------------------------------------------------
package P4 is
   . . .
   procedure PU_P4 (    ...    );
   . . .
end P4;
package body P4 is
   . . .
end P4;
-------------------------------------------------------------
package P5 is
   . . .
   procedure PU_P5 (    ...    );
   . . .
end P5;
with P2;
use P2;
package body P5 is
   . . .
end P5;
```

FIGURE 118. (CONCLUDED)

All members of the development team must be familiar with the concept of object-oriented design with Ada, and the cost ramifications of this approach. Only when complex dependency relationships between types, variables and program units are controlled, can large and complex software systems be developed in a cost effective manner. The old traditional approaches have proven to be expensive when used to implement large and complex computer programs.

When teaching Ada to programmers, the source code can be taught in the context of an object-oriented design in a top-down manner. This approach provides a meaningful context for the many diverse Ada capabilities and provides a framework for the programmer to understand the cost ramifications of software implementation using Ada.

PART THREE: AUTOMATION OF SHARP

CHAPTER IX

PHASED DEVELOPMENT OF AUTOSHARP

The graphics presented in this document have been, in part, prepared at Arthur D. Little using an IBM dual drive personnel computer and the software package FREELANCE written by Graphic Communications, Inc. The command language of this software package is menu driven and relatively straightforward to learn and use. With this computerized drawing capability, complicated diagrams presented in this report (e.g., the intermediate level hierarchy and invocation diagrams) took about 15 minutes each to prepare. The less complicated diagrams (e.g., low level subprogram data flow and task rendezvous diagrams) took about 5 minutes each to prepare.

With the automation of SHARP, efficient interfaces between users of SHARP and FREELANCE, or some other graphics software package, will be developed.

I. INTRODUCTION

1.1 BACKGROUND

An automated SHARP capability is needed in the short-term, as the DoD software engineering community "gears up" for initial applications of Ada. For example, a SHARP software system for teaching Ada would help students attempting to grasp such complex notions as design abstraction and information hiding, in the context of object-oriented design, as discussed in Chapter VIII.

Furthermore, in the short-term software engineers will have to establish cost estimates for the development of Ada software. Existing cost models can be used to project Ada costs, based upon a projection of the number of source statements. An automated SHARP system for establishing the graphical representation of an Ada design provides a framework for projecting the number of instructions required to implement a large and complex Ada computer program, in the manner discussed in Chapter VII. Such projections can be made automatically, as described in subsequent paragraphs of this chapter.

As described in Chapters III and IV, an automated SHARP system would also be invaluable in the long-term, when major defense system contractors develop large and complex Ada computer programs. Such a graphical system should prove to be helpful to both contractor personnel and government personnel, especially in the preparation of graphics for design reviews and design documentation. With a SHARP software design system, the SHARP Ada abstracts could be generated in a timely manner, and easily iterated to update or optimize an Ada design.

Furthermore, as suggested in the next section, a SHARP software system should prove to be invaluable to Air Force personnel responsible for the maintenance of a large Ada computer program.

1.1.1 Computer Aided Design

Computer Aided Design (CAD) tools have existed for years to support the designers of electrical, mechanical and civil engineering systems. In the last two years, the development of such CAD systems for the design of Ada software has been initiated. For example, CAEDE (Carleton Embedded System Design Environment) is being developed by R.J.A. Buhr at Carleton University, Ottawa, Canada; and PAMELA (Process Abstraction Method for Embedded Large Applications) is being developed and implemented in the automated tool AdaGRAPH by George Cherry at The Analytic Sciences Corporation (TASC).

Among other things, the following factors should be considered when developing a CAD system for Ada.

a. The design of a large and complex computer program to be implemented in Ada should be represented in a comprehensive but abstracted manner.

b. Process synchronization and the implementation of applications software should be kept simple to permit straightforward testing not necessitating elaborate and costly proofs of temporal properties.

c. The number of Ada tasks used in the implementation of applications software may have to be limited due to testing difficulties and the limitations of hardware architectures currently in existence. Existing hardware may not be able to support a large number of interacting tasks in a timely manner.

d. Dependency relationships between types, variables and program units will have to be controlled to facilitate cost effective software development and to save money during software maintenance.

We feel that SHARP complies with item a. We also feel that because of items b and c, extensive use of Ada tasks is not practical in the "real" world. Furthermore, we feel that because of item d defense contractors will have to take advantage of design techniques referred to as "object-oriented." In contrast with CAEDE and PAMELA, SHARP has been developed with emphasis on these factors.

SHARP can be used to represent low level design detail in a manner similar to both CAEDE and PAMELA. However, it can also be used to directly represent higher levels of an object-oriented design. We feel this level of design is very important. As demonstrated by the example presented in Chapter IV, large and complex computer programs can be composed with layers of objects. Objects can be implemented with a source statement count that

allows compilation in a timely manner. By stubbing interacting object implementations, a given implementation can be relatively quickly compiled. The development of the such object implementations will also be relatively straightforward, as will be their integration if they are sufficiently decoupled.

1.1.2 Knowledge-Aided Design (KAD) and Maintenance

An Ada KAD system can represent the design of an Ada computer program using graphics, which represent the design in a comprehensive and abstracted manner. With it, software designers will be able to relatively rapidly generate abstracted design representations. The abstracts can be reviewed and the design representation iterated in order to, in some sense, optimize the design. Knowledge built into the KAD system will help guide inexperienced designers lacking extensive knowledge of Ada and object-oriented techniques. In this way, typical inefficiencies in design development and representation will decrease.

Upon system turnover to users, graphical abstracts can be used to support software maintenance. The maintainer would be able to selectively produce abstracts that, in a systematic manner, zero in at the touch of a terminal key on parts of the program he must modify. The abstracts would make the complexities of the design readily apparent, as opposed to culling thousands of statements in a source code listing. The exclusive use of source code to maintain a large and complex computer program has proven to be very expensive, as we have already indicated.

1.1.3 Automatic Programming and Cost Estimation

In addition to being used to establish design abstracts, a design knowledge base established by a user of a KAD system could also be mapped into Ada source code. The code would encompass aspects of the design directly accounted for in the design abstracts. This code could, in turn, be expanded and refined by a programmer using a syntax directed editor. In the field of artificial intelligence, this capability is referred to as automatic programming and knowledge engineering.

Knowledge Engineering utilizes expert knowledge and heuristics to generate a computer program in a specific high order language. This is accomplished by introducing the knowledge in the form of rule-type data structures, which can be added to or removed from the knowledge base.

Automatic Programming systems are said to be knowledge-based when they encompass knowledge or expertise for program synthesis, including programming knowledge. Programming knowledge includes both programming language knowledge and the semantics of the high order language in which the computer program will be written. It may include general programming knowledge about such general computational mechanisms such as initialization, loops, sorting, searching, linked lists and hashing. It also may include planning, optimization, and high-level programming techniques.

1.1.4 Size Metric Derivation

The design knowledge base could also be mapped into a size metric. The size metric, along with user inputs on the attributes of the software acquisition, could be used as inputs to a cost estimation algorithm that projects the cost to build a large and complex computer program. In this way, the cost estimation problem is merged with the automated design process so that meaningful estimates can be generated in a timely manner.

1.2 CHAPTER SCOPE

In this chapter, Section 2 provides an overview of AdaGRAPH and CAEDE, two Ada-unique computer-aided programming systems. Section 3 suggests a phased development of an Automated SHARP system, which we refer to as AUTOSHARP. The system encompasses capabilities for KAD, Ada-unique automatic programming and Ada-unique cost estimation.

2 EXISTING CAD SYSTEMS

As we have indicated, automated systems that support the design and implementation of Ada computer programs have been extensively investigated. As examples of such systems, let us consider AdaGRAPH (PAMELA) and CAEDE.

2.1 AdaGRAPH (PAMELA)

AdaGRAPH is an automated system that supports the development of large software systems in Ada. It automates the capabilities of PAMELA, which (as we have indicated) is the requirements abstraction technique being marketed by TASC. As such, AdaGRAPH acts as the syntax-directed graphical editor and compiler of PAMELA.

PAMELA is an Ada-specific method for (a) transforming a software requirements specification into a design and (b) transforming the design into Ada source statements. It supports a user in undertaking the following:

a. A 'Specification' step, where software requirements are transformed into a hierarchical process graph. The user develops a graph of external entities and the software interfaces with them. This graph is then expanded in an abstracted manner to show the major events associated with each entity. The results can be iterated by the development team.

b. An 'Architectural Design' step, where program units are identified to implement the events established in the first step.

c. A 'coding' step, where "skeleton" code is automatically generated for program unit calls. Then using a syntax directed editor, a programmer is able to establish the detailed code needed to implement each of the called program unit.

PAMELA functions on a semantic level but is not knowledgeable about the heuristics of object-oriented design. When representing the design of an Ada computer program, PAMELA is limited to only a hierarchy diagram of program units and does not graphically distinguish between the different kinds of Ada program units. As such, it does not provide a comprehensive graphical representation of an Ada design, whether being developed using traditional or object-oriented techniques. PAMELA also appears to be preoccupied with extensive use of tasking (which we feel is not practical, as discussed in Section 1.1.1) and does not provide an explicit mechanism for representing object-oriented designs for Ada (which we feel will be necessary for pragmatic reasons, as discussed throughout this document).

However, it does appear to provide a viable graphical mechanism for systematically presenting the requirements for a software package in an abstracted manner. As such, it could be used at the requirements analysis level, prior to comprehensively establishing an Ada design (e.g., using a tool like AUTOSHARP).

2.2 CAEDE

CAEDE is an automated system that supports establishing the structural design of Ada programs using icons. CAEDE allows the user to enter structural and temporal design information using a graphics interface. It serves as a basis for design analysis and skeleton code generation. It facilitates, under control of the interface, the entrance of program "strips" to fill in the functional gaps in the skeleton code. The iconic information is converted automatically into a Prolog design data base of facts and rules, for off-line temporal assessment.

CAEDE represents design level abstractions modeled on selected features of Ada as graphical icons. It supports a user in undertaking the following four steps:

a. A 'Structure of the Level' step, where a schematic diagram of the program components and their interactions is produced.

b. A 'Temporal Behavior' step, where temporal behavior is described for tasks across interfaces, blocking and unblocking of tasks at the interfaces, and the enabling and disabling of blocking conditions at the interfaces.

c. A 'Specification of Internal Temporal Characteristics' step, where the designer specifies the characteristics that are required to achieve the temporal interface behavior.

d. A 'Program Strip Definition' step, where sequential program fragments are provided to fill in the gaps in the skeleton code produced by the previous steps. They are entered into the skeleton program using the iconic interface.

CAEDE uses Prolog rules to generate Ada code skeletons from the Prolog facts in the design data base. Using Prolog for compiler implementation is advantageous in that Prolog has the ability to perform translations given only the translation rules. The system contains language syntax rules in order to generate a legal program satisfying the syntax rules. Construction of Ada code under these rules proceeds until a decision must be made - then the system consults the Prolog facts in the data base, resolves issues, and produces the skeleton code.

CAEDE can be considered a prototype of a commercial CAD/automatic programming system. Its method of nesting program units using icons is limited (i.e., its graphical notation would be difficult to apply in large and complex software systems). Like PAMELA, it is heavily oriented towards extensive use of Ada tasks, which we feel is not practical.

3. PHASES OF DEVELOPMENT

SHARP provides a set of pictorial abstracts that can be used to represent the design of an Ada computer program in a comprehensive manner. SHARP can be used to represent all extremes of design with respect to both size or design methodology. The abstracts of SHARP can be generated as a product of a knowledge based intelligent design system for Ada, capable of design, automatic programming and cost estimation. AUTOSHARP Version I provides an Ada-unique capability for knowledge aided design (KAD). AUTOSHARP Version II provides an Ada-unique automatic programming capability and AUTOSHARP Version III provides an Ada-unique cost estimation capability.

3.1 AUTOSHARP VERSION I (KAD)

AUTOSHARP Version I is a KAD system used to establish the SHARP graphical representation of the design of a computer program to be implemented using Ada. Version I interfaces a user (a software designer) with software graphical packages, which can be used to create pictographic abstracts of large and complex computer programs to be implemented in Ada.

3.1.1 Version I Description

The core of AUTOSHARP Version I is a knowledge based system envisioned as follows:

	AUTOSHARP
Description Method	Natural Language, Description by Method Example, and Graphical Description
Target Language	Ada
Problem Domain	Efficient Object-Oriented Design and Traditional Design Abstraction
System Approach	Knowledge Engineering

As shown in Figure 119, AUTOSHARP Version I will consist of (a) classic primary elements -- a User Interface, an Inference Engine, and a System Knowledge Base, and (b) task specific elements -- a Help capability, a User Interface Enhancer, a Library of Reusable Software, a Graphics Mapper and a Graphics Generator.

3.1.1a Primary Elements

The 'User Interface' will receive and manipulate information establishing the design of the computer program. The user will enter data using menus, tables and natural language interface.

The 'Inference Engine' will perform reasoning using domain specific information. It will review the design as a function of the problem domain to identify any potential design deficiencies, based upon rules formulated within the Knowledge Base . The Inference Engine will detect missing information and will bring such problems to the attention of the user.
The 'Knowledge Base' will encompass facts, definitions and rules applicable to the consistency and completeness of the overall design.

3.1.1b Task Specific Elements

The 'Help Capability' provides a menu of selectable guidance to the designer, relevant to the use of AUTOSHARP and both object-oriented design (OOD) techniques and traditional techniques. For example, OOD help includes tutorial information on object selection, operations unique to an object implementation, local data structures and passing parameters between object implementations.

The 'Library of Reusable Software' contains design information for existing object implementations (e.g., a Fast Fourier Transform) and components used to implement objects (e.g., math functions and routines for such things as stacks, queues and trees).

The user 'Interface Enhancer' provides capability for enhancement to the interface, allowing a diverse number of variations for information input (e.g., a Natural Language (English) capability, Touch-screen capability, Digitizing Pad capability, and Ada Command capability).

The 'Graphics Mapper' produces SHARP pictographic commands as a function of the user generated knowledge. The Graphics Generator receives the commands from the Graphics Mapper and produces SHARP pictographs on a display and/or printer.

3.1.2 Version I Operation

With AUTOSHARP, an operator can describe the design of a computer program. The description, in English, could be:

> "Show MAIN as the main program with three tasks declared to represent processes. The tasks shall be named COMM_LINK, WORK_STATION and BUILT_IN_TEST".

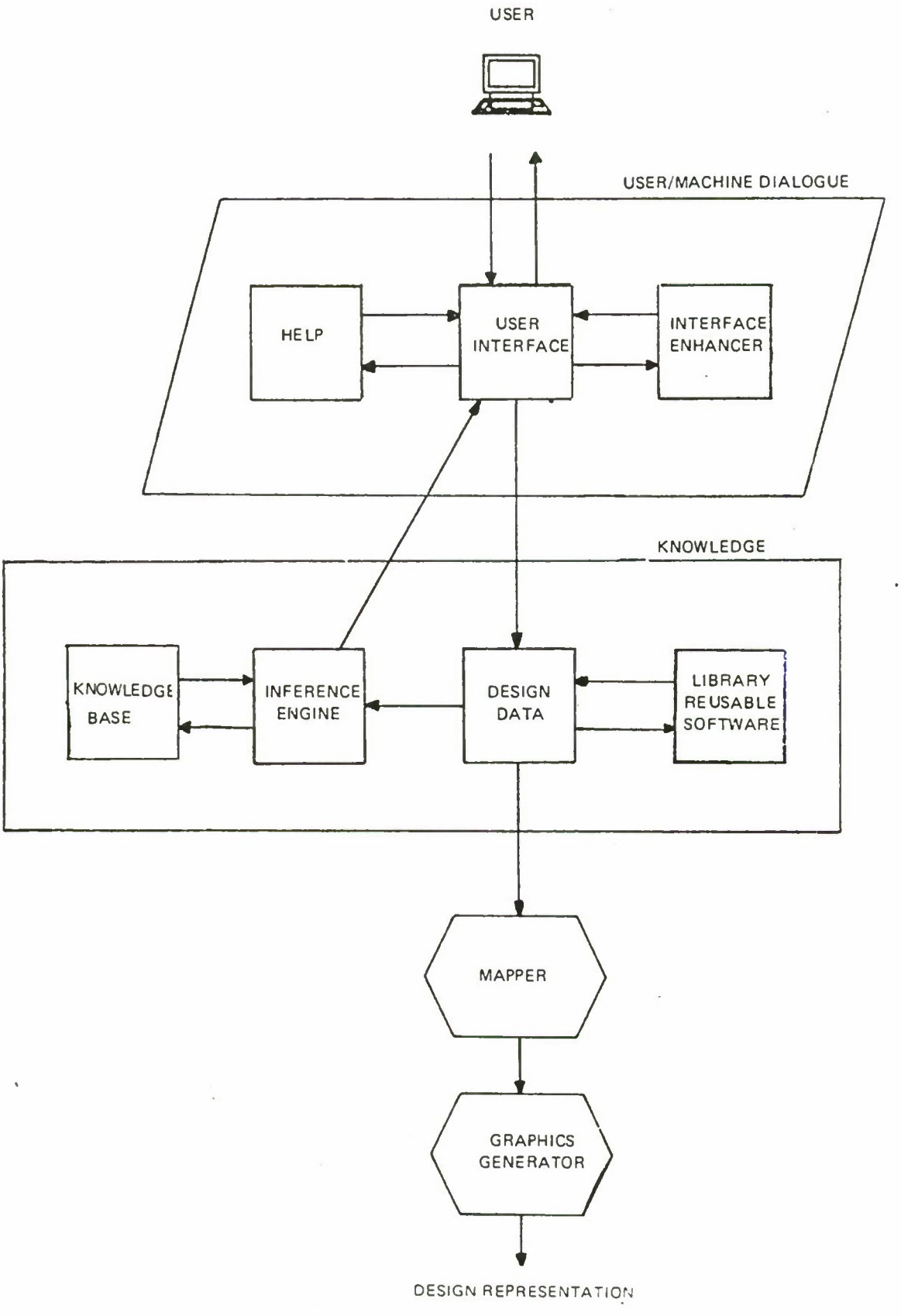


FIGURE 119. ELEMENTS OF AUTOSHARP

The User Interface would accept this data, and the Knowledge Engineering segments would, through the use of facts, rules, and heuristics, yield a legal and correct Ada conceptual construct which would become input to the Graphics Mapper. The Mapper would generate output for the Graphics Generator yielding the SHARP abstract shown in Figure 120.

3.1.3 Scope of the AUTOSHARP Knowledge Base

It is envisioned that the Knowledge Base of AUTOSHARP will consist of rules concerning the semantics, grammar, and features of the Ada language. In addition, the knowledge base must include knowledge of the specifics of SHARP and SHARP's own internal rules. Knowledge about the rules themselves, metaknowledge or heuristics, must also be included.

The Knowledge Base is envisioned as having rules similar to these following English-like examples:

```
"If this is the main program
   then information is needed
        on the number of packages that are to be 'withed',
        on whether or not to 'with' TEXT I/O,
        on whether or not to 'with' SEQUENTIAL I/O,
        on whether or not to 'with' DIRECT I/O,
        on the names of developed packages to be 'withed',
        on the names of subprograms to be declared,
        and on the names of tasks to be declared."

"If this is a process
   then information is needed
        on the number of objects,
        on the alpha-numeric identifier of the first object,
        on whether or not this object is a package or task,
        on the names of other program units in the package with which
          this unit will communicate,
        on the name of an interacting object,
        on the name of the second object,
        on whether or not this object is a package or task,
        on the names of other program units in the package with which
          this unit will communicate,
        on the name of another interacting object,
          . . ."
                and so on.

"If this is an object
   then information is needed
        on the object name,
        on the applicable process number,
        on the visible data structure type definitions,
        on the local data structure constant definitions,
        on the names of communicating program units,
          . . ."
                and so on.
```

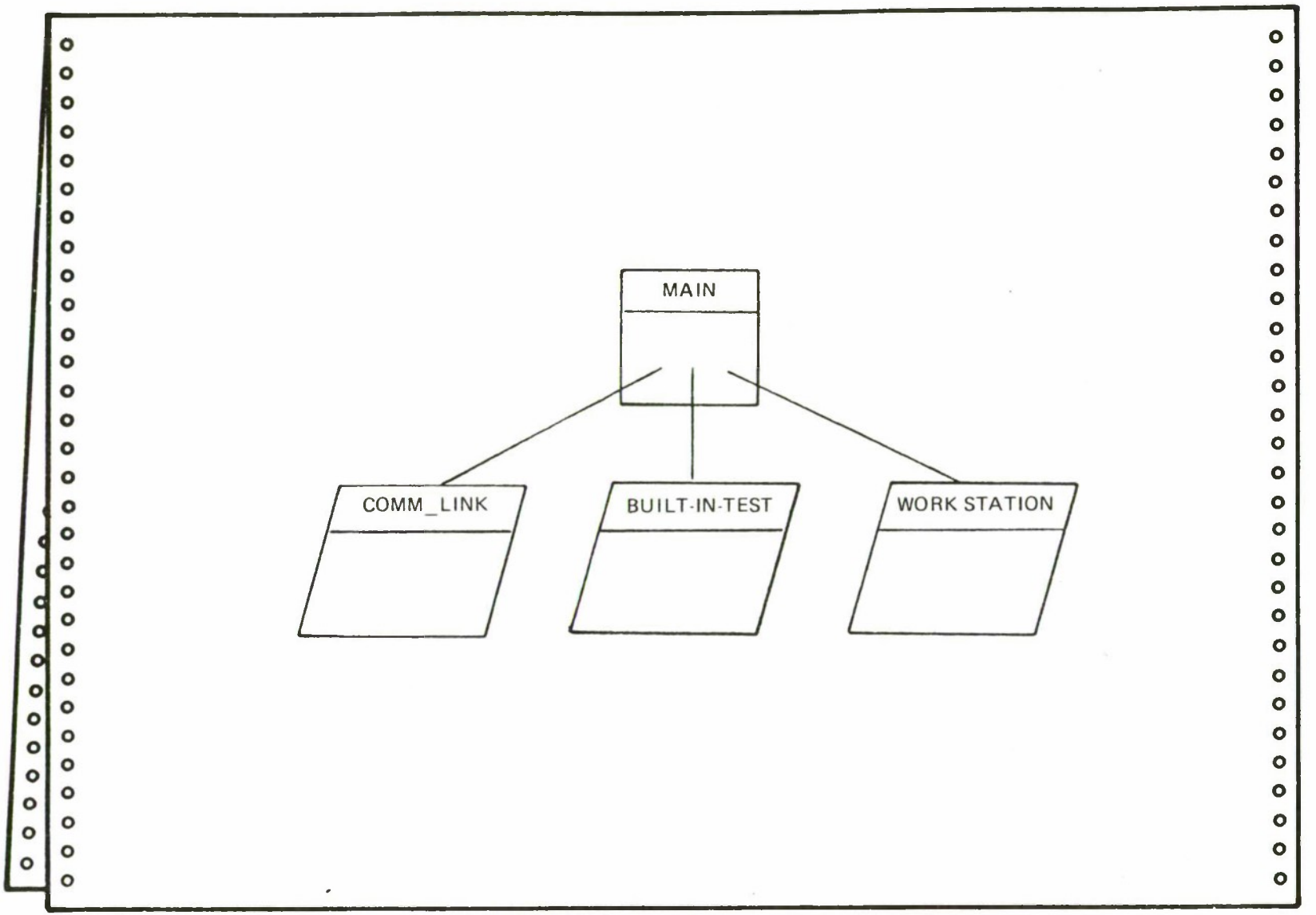


FIGURE 120. SHARP ABSTRACT FOR SAMPLE OPERATOR INPUT

By codifying these rules into the knowledge base, the design problem can be described very precisely with no ambiguity. Since the system is interactive with the user, any partial information can be elaborated via a question and answer dialogue with the user. In this way, AUTOSHARP can further specify the precise nature of the design.

3.1.4 Mapping the Output

As shown on Figure 121, SHARP abstracts will be available to the user after the output from the Knowledge Base is processed through the Graphics Mapper and the Graphics Generator. The Mapper will contain the facility to translate the resultant design structure provided by the Knowledge Base into a form suitable for input into the Graphics Generator. This segment will, most likely, translate design structure commands into the appropriate geometric form and, in order to view various size segments of the design, manipulate the display into a suitable scale. The Graphics Generator therefore need only be a relatively precise drafting/text generation software package.

3.2 AUTOSHARP VERSION II (CODING CAPABILITY)

The second phase of the implementation of AUTOSHARP adds capabilities to generate Ada source code listings that correspond to the design graphics generated with Version I of AUTOSHARP. As shown by bold face in Figure 122, the core of AUTOSHARP Version II will add two elements to Version I -- an Ada "Skeleton" Code Generator and an Ada Syntax Directed Editor.

The Ada Code Generator uses the output of the knowledge base to provide the programmer with existing code from the library of reusable software and "skeleton" code of legal, correct Ada for the rest of the design established with AUTOSHARP Version I. This code implements to the extent possible the design represented by high, intermediate and low level SHARP abstracts, and flags gaps where they exist.

The Ada Syntax Directed Editor can then be used by a programmer to refine and complete the code. This editor, knowing the "rules" of the language, provides the capability to produce the detailed Ada source code listings in a computer-aided manner. It is used to integrate the detailed code into the "skeleton" code produced automatically by the system. As an example, suppose the system outputs the following "skeleton" code --

```
task body ALERTER is
        . . .
   select
        ALARM.POST_ALARM (. . .);
   else
        . . . **Alternative Required**
   end select;
        . . .
   end ALERTER;
```

where **Alternative Required** is a message from the system alerting the designer that further information must be provided for legal coding. The designer could then "flesh out" the code as follows:

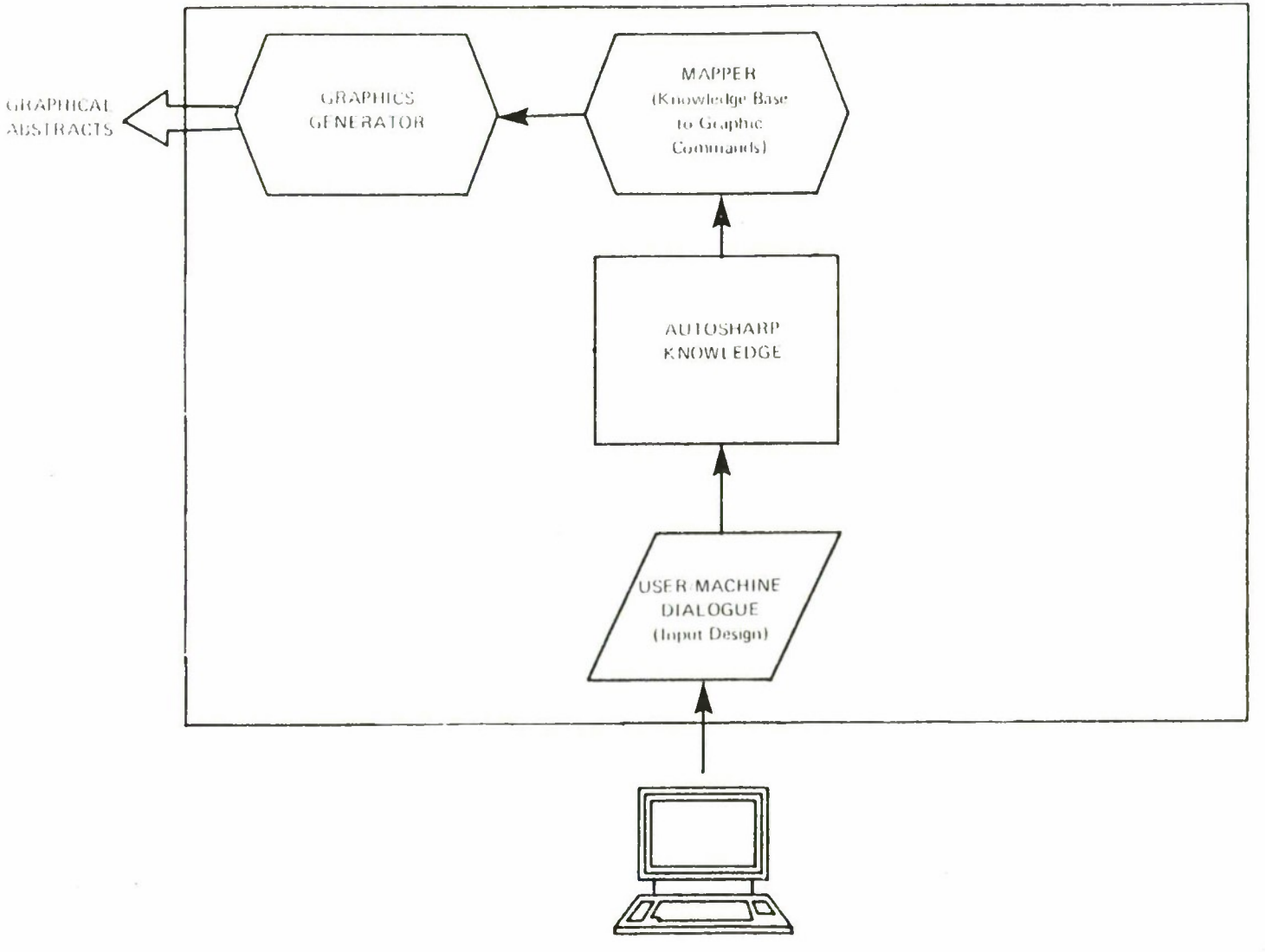


FIGURE 121. AUTOSHARP VERSION I

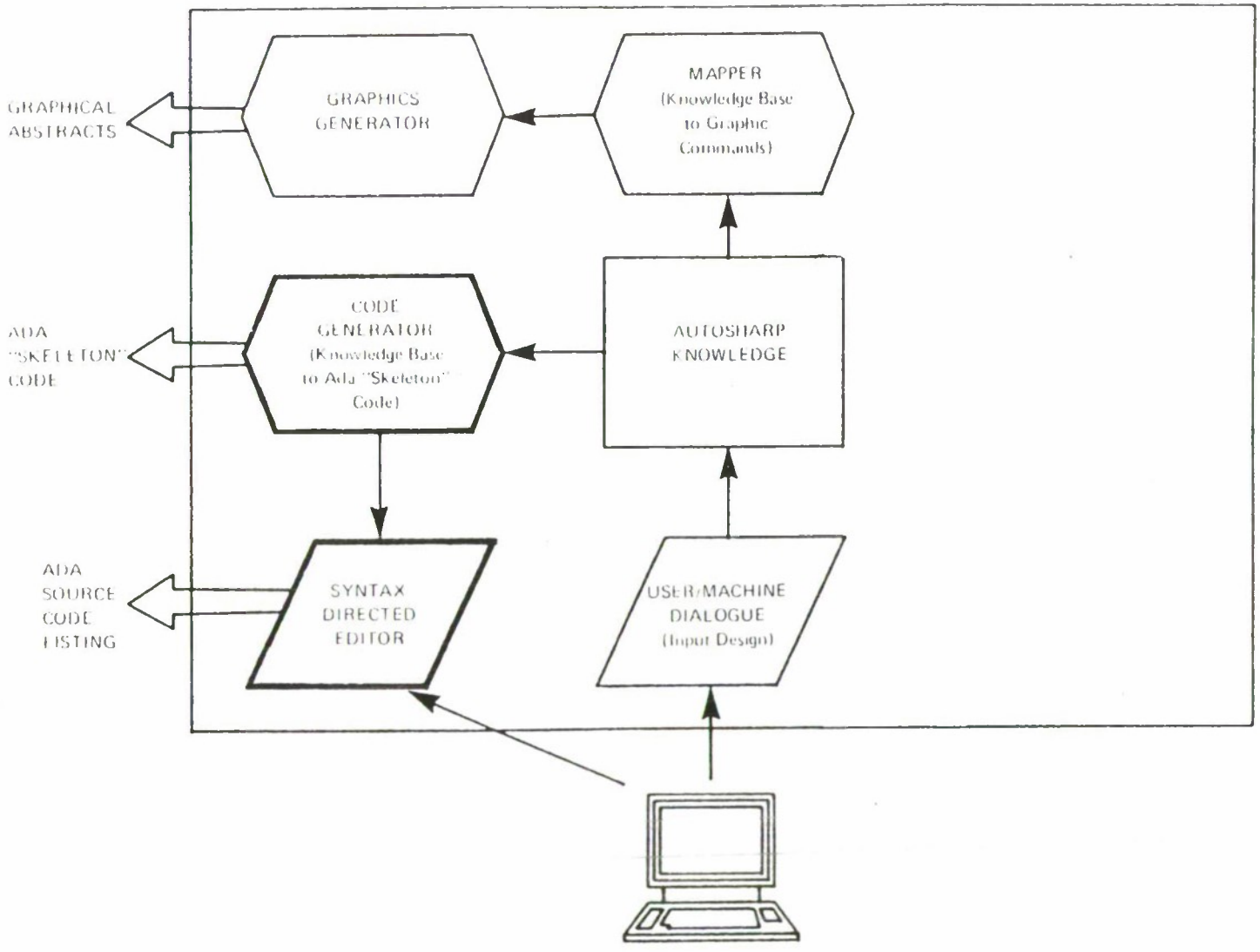


FIGURE 122. AUTOSHARP VERSION II

```
task body ALERTER is
        . . .
   select
        ALARM.POST_ALARM (. . .);
   else
        accept STOP
   end select;
        . . .
   end ALERTER;
```

At this point, the AUTOSHARP system is capable of the following:

Interfacing with the user
"Translating" user requests into a manipulatable form
Making inferences about Ada program design through use of the Knowledge Base
Producing output to the Mapper
Producing Pictographs from the Mapper via the Graphics Generator
Producing Ada "Skeleton" Code
Producing Ada Source Code Listings

The utility of such a system is apparent in that the KAD/automatic programming capabilities of AUTOSHARP would increase the productivity of software designers and programmers. Therefore, a significant increase in the productivity of software design and source code generation would yield significant savings to the overall software development effort. These gains are in addition to the gains that can already be realized through the use of object-oriented design techniques implemented with Ada, as quantified in the manner described in Chapter VII and represented (for a representative example) in Figure i of the Executive Summary.

3.3 AUTOSHARP VERSION III (COST ESTIMATION)

A basic task for a manager of a software acquisition is the calculation of accurate cost estimates. Such estimates are needed by the government to establish meaningful budgets, provide the basis to assess contractor bids and monitor the progress of software work from a cost point of view.

Most existing software cost models operate on a projection of the software size to establish their estimates. It is not possible to make accurate cost estimates without accurate size estimates. The absence of a credible size metric reminds one of the Vermont farmer who attempts to weigh his pigs, before going to market. He carefully balances his scale using rocks and then guesses at the weight of the rocks.

To avoid "guessing at the weight of the rocks", AUTOSHARP introduces segments needed to interlock the design process with the cost estimation process. Specifically, as shown by bold face in Figure 123, AUTOSHARP Version III will add elements to Version II -- an Estimation Interface, Knowledge Base Accessor and Cost Estimate Calculator.

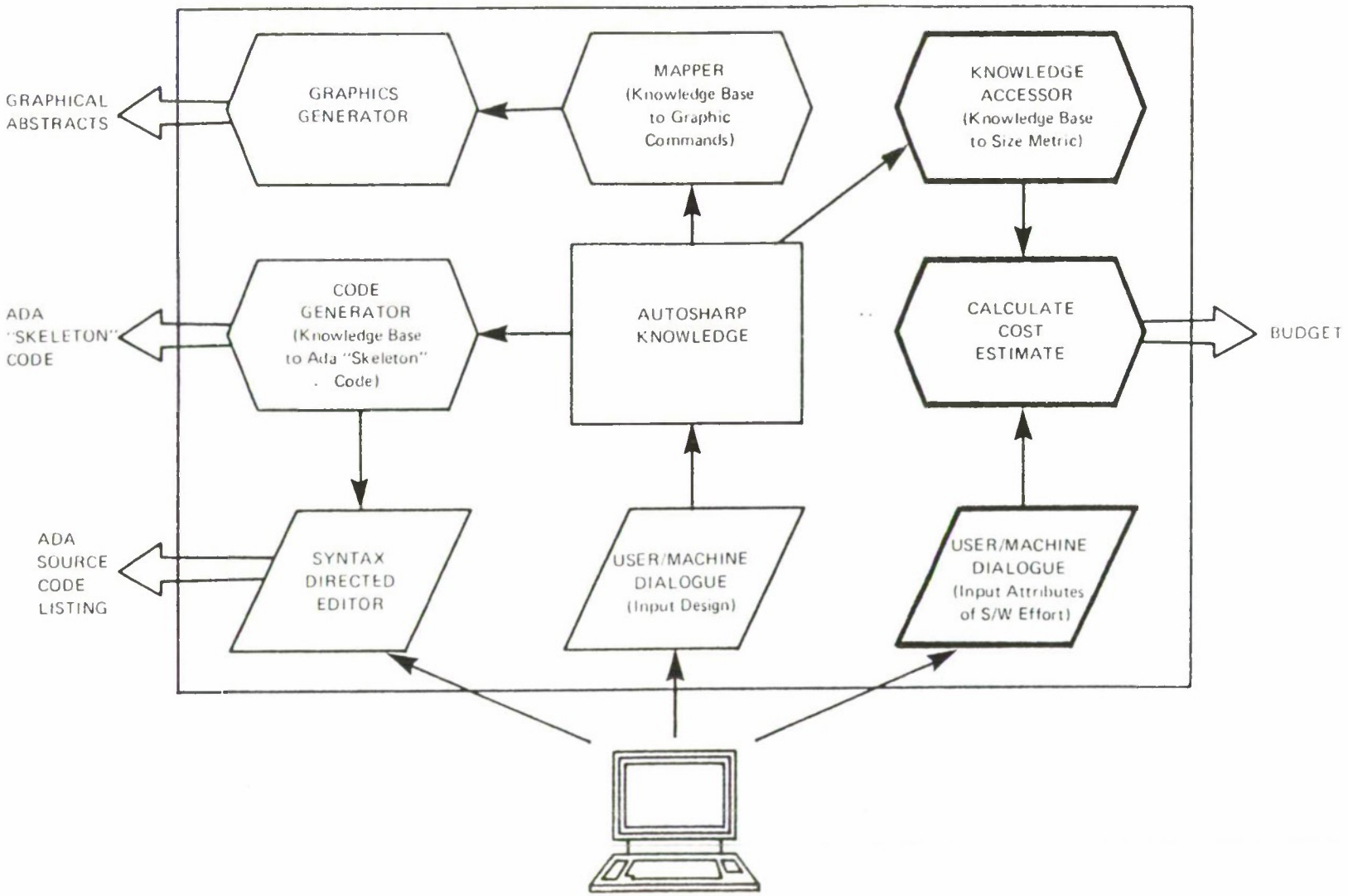


FIGURE 123. AUTOSHARP VERSION III

The Estimation Interface communicates with an estimator to establish data describing the software acquisition (e.g., metrics describing product attributes, the computer used, and development personnel).

The Knowledge Base Accessor will communicate with the Knowledge Base in order to retrieve essential design data to accurately estimate the size metric for the subject computer program.

Cost Estimate Calculator is specifically designed to process user and knowledge base information using applicable algorithms for cost estimation (e.g., COCOMO) customized to the design methodology to be applied. Analogous to a spreadsheet, changes in the knowledge base's data used to derive the program size and complexity can be communicated directly into the cost model. Rather than relying on the traditional educated guess regarding size and design variations, such information can be directly extrapolated. In this way, the cost ramifications of design variations can be assessed. The establishment of algorithms needed to make such an extrapolation are provided in Chapter VII.

3.4 AUTOSHARP METAKNOWLEDGE

An inherent advantage to introducing rule based data within the knowledge base is that the system can contain rules about the rules. Thus the system itself is able to infer, within the accuracy of its original data, answers to questions which are not typically answerable by a system. Consider asking your word processor how many paragraphs are now in this document. Because of the conceptual structures used to create representations of knowledge, knowledge of the program itself is not transparent to the system. As an example, AUTOSHARP will compare a user's design to that of a KAD model of the design. Possible deficiencies in the user's design can be detected and brought to the user's attention. Since the system can "see" its own design and code, a query concerning the number of pictographs and lines of code can be raised.

Additionally, key words can be counted or otherwise analyzed (depending on the usual restraints -- how much memory is available, how sophisticated the cost estimation search is allowed to be, etc.).

4 BENEFITS OF AUTOSHARP

The design of large, complex computer programs is a costly and challenging process. Furthermore, a design will increase or decrease subsequent software work during test, system integration and maintenance. Designers of Ada unique software must typically establish a set of packages to encompass all its major parts, must carefully design each package and its interface to eliminate complex dependencies, and must design each package in an abstracted manner. AUTOSHARP will help designers establish correct designs, will help programmers establish error-free code and will help managers by providing accurate reliable costing data. These capabilities are all critical to lower costs in the development of software using Ada.

Through the use of a knowledge-base and KAD techniques inherent in AUTOSHARP, a designer can create a program design graphically. This design will be consistent with the rules of Ada, and the resulting design graphics will violate neither any tenets of the design philosophy nor any graphical rules.

AUTOSHARP will produce "skeleton" code in accordance to a set of rules on the grammar, syntax, structure, and typing of Ada. This "skeleton" code will be correct when produced. This production of code will increase the amount of viable coding being done by a programmer and, as a side effect, constrain the universe of testing.

Inasmuch as the Knowledge Base contains rules and rules about rules, inaccurate designs will be questioned by the system and designers constrained to designs within the parameters of proper object-oriented code. This will, in the long run, create programs that are technically superior and cost effective. Should large numbers or programs be created under a single set of well-defined, well-constrained, and well-founded rules, the style of such programs will be consistent and therefore familiar to the professionals maintaining them. Because of this, they will be less difficult to maintain. Since software maintenance costs are very high, this capability is significant.

Another beneficial capability of AUTOSHARP is the cost estimator. Given accurate models, the system will have the capability to query itself and provide data on the variables effecting the design cost. This information should be significantly more readily available than it is now. Cost estimation using AUTOSHARP could be such a facile task that metrics and estimates could be generated essentially at the touch of a terminal key, as opposed to several months of analytical effort. The estimates also could be readily generated so that "what if" and design tradeoff studies could be performed. More, importantly the accuracy of the estimates would be superior to those produced in the past at great cost, since the estimates would be driven by up-to-date and accurate projections of the software size.

APPENDIX A

CRITERIA FOR SHARP

A. GENERAL

1. Graphical representation of a computer program to be implemented in Ada shall meet the criteria set forth in Paragraph B below.

B. CRITERIA

1. Pictographs for Program Units

a. A subprogram must be represented by a square.

b. A task must be represented by a parallelogram.

c. A package must be represented by a rectangle.

d. Each pictograph must be divided by a horizontal line into a narrow part representing its specification and a wide part representing its body. For a generic program unit, the dividing line is dashed.

e. The specifications of Ada subprograms and tasks contained within a package, and directly accessed by program units external to the package, must be shown within the package's specification. The bodies of these subprograms and tasks must be shown within the package's body.

2. Hierarchy Diagram for the Main Program of an Ada Computer Program

a. The Ada main program must be represented by the pictograph for a subprogram.

b. A straight line must be drawn from the body of the Ada main program to each pictograph used to represent an Ada task introduced to implement a process.

c. A straight line must be drawn from the body of the Ada main program to each pictograph used to represent an Ada subprogram declared in the Ada main program.

d. A small rectangle must be introduced to represent each Ada "with" clause applicable to the Ada main program. A straight line must be drawn from each rectangle to the specification of the Ada main program.

e. A dotted line must be drawn from the specification of each Ada task (representing a process) to a geometric representation of an external interface with the task, when appropriate.

f. The name of each Ada subprogram, task and package must be clearly shown within, or adjacent to the corresponding pictograph.

3. Ada Package Catalog

a. The names of Ada packages used in the implementation of a large Ada computer program must be shown in a set of cells within a rectangle. Each cell must contain the name of one package.

b. The name of a generic package must be encapsulated by a cell drawn with dashed lines within the package's cell.

4. Ada Package Content Diagram

a. Visible program units declared in the specification of a package must be provided in the pictograph for a package.

b. Hidden program units, other than a nested package, do not have to be shown.

c. A nested package must be shown under the subject package, with a straight line drawn from the body of the subject package to the specification of the nested package.

d. A straight line must be drawn to the package's specification or body, as appropriate, from a small rectangle to indicate a package "with" clause.

e. The name of a package, nested packages and packages accessed through the "with" clause must be clearly shown within, or adjacent to the corresponding pictograph.

5. Hierarchy Diagram

a. The subject program unit must be assigned to Level 1.

b. Program units nested within the subject program unit must be assigned to Level 2.

c. Program units nested within a program unit at Level n must be assigned to Level n+1.

d. A straight line must be drawn from the body of a program unit at Level n to the specification of a nested program unit at Level n+1. In this way, all levels of abstraction must be established.

e. A straight line must be drawn from a small rectangle indicating a package "with" clause to the program unit to which the clause applies.

f. The name of each subprogram and task, and each package accessed through the "with" clause, must be clearly shown within, or adjacent to, the pictograph that represents it.

6. Invocation Diagram

a. A straight arrow must be drawn to the specification of a subprogram from the body of the program unit that called it.

b. A curved arrow must be drawn to the specification of an acceptor task from the body of a calling task to represent task rendezvous.

c. A subprogram or task must be bounded by a square or rectangle if it is contained within a package other than the package containing the calling program unit. The bounding square or rectangle must be dashed if the subject program unit is generic.

d. Any subprogram or task, which depends upon some transient condition in order to be accessed (for example, 'select,' 'accept,' 'if,' or 'case' statements), must be pictorially represented as if it were accessed. The fact that such access is conditional must be expressed by a tilde (i.e., a short undulating line) placed on the arrow indicating accession.

e. Except in recursive cases and loops, if a subprogram or task (i.e., any program unit) is called n times during the execution of a program, the program unit must be shown n times in an invocation diagram.

f. Recursive calls must be represented by arrows with heads pointing to both of two program units or by "feedback loops," with an asterisk provided adjacent to the doubled headed arrow or "feedback loop."

g. The names of each program unit must be clearly shown within, or adjacent to, the pictograph that represents it.

7. Task Rendezvous Diagram

a. A task entry point must be represented by a small parallelogram that overlaps the task's specification and body.

b. A call to an Ada task must be represented by three arrows from the body of the calling task to an entry point in the acceptor task. One arrow must have its head pointing to the entry point, to represent the "in" mode of parameter passing. A second arrow must have its head pointing to the body of the calling task, to represent the "out" mode of parameter passing. The third arrow must have two heads, thus pointing to both the entry point and the body of the calling task, to represent the "in out" mode of parameter passing. If one or more of the modes are not applicable, the arrows can be omitted.

c. A conditional call must be indicated by tilde across the three arrows running from the body of the calling program unit to the entry point. The letter "T" is shown adjacent to the tilde a time conditional call is to be made.

d. Conditional acceptance of a call must be indicated by a tilde within the affected entry point, or the letter "T" within the entry point if a time conditional acceptance applies. Acceptance on a fixed order basis must be represented by numbers within the affected entry points, where "1" indicates first. Acceptance on a first arrival basis must be represented by a line connecting the affected entry points.

e. Parameters received by the accepting task must be represented by shaded circles on the line representing the "in" mode; parameters received by the calling task must be represented by shaded circles on the line representing the "out" mode; and parameters transferred in the "in out" mode must be represented by shaded circles on the line representing the "in out" mode.

f. The names of each task must be clearly shown within, or adjacent to the pictograph. Entry point names and names for passed parameters must be clearly shown adjacent to the corresponding pictograph.

8. Subprogram Data Flow Diagram

a. A call from one subprogram to another must be represented by three arrows drawn from the body of the calling subprogram to the specification of the called subprogram. One arrow must point to the specification of the called subprogram to represent the "in" mode of parameter passing; one arrow must point to the body of the calling subprogram to represent the "out" mode of parameter passing; and the third arrow must have two heads, thus pointing to both the called subprogram and the body of the calling subprogram, to represent the "in out" mode of parameter passing. If one or more of the modes are not applicable, the arrows can be omitted.

b. A conditional call must be indicated by a tilde across the three arrows running from the body of the calling program unit to the specification of the called program unit.

c. Parameters received by the called subprogram must be represented by shaded circles on the line representing the "in" mode; parameters received by the calling subprogram must be represented by shaded circles on the line representing the "out" mode; and parameters transferred in the "in out" mode must be represented by shaded circles on the line representing the "in out" mode.

d. Names for subprograms and passed parameters must be clearly shown.

9. Generic Program Units

a. In the pictograph for a generic program unit, the horizontal line drawn to divide a pictograph into a narrow part (representing a program unit specification) and from a wide part (representing its body) must be dashed.

b. In a data flow and task rendezvous diagram, circles representing generic parameters to be passed must not be shaded.

c. The small rectangle used to represent the Ada "with" clause must be dashed if the relevant program unit is generic.

10. Data Structure Diagram

a. Type declarations must be represented by a series of upright narrow rectangles, side-by-side.

b. Constant declarations must be represented by a series right-slanted rectangles, side-by-side.

c. Variable declarations must be represented by a series of left-slanted rectangles, side-by-side.

d. The geometric representation of visible declarations must not be shaded; the geometric representation of hidden declarations must be shaded.

e. The geometric representation of private declarations must be half shaded and half unshaded.

f. The name of each type, constant, and variable must be clearly shown in a glossary. The glossary entries must be referenced by numbers placed directly over the geometric figures representing each type, constant and variable.

g. Directly under the geometric representation of a type, the letters "AR" must appear if the type is an array type; the letter "R" must appear if a record type; the letter "D" must appear if a discriminated type; the letters "AC" must appear if an access type; the letter "T" must appear if a task type; the letter "I" mustr appear if an integer type; the lettrs "RL" must appear if a real type; and the letters "EN" if an enumeration type.

h. Directly under the geometric representation of each constant and variable, the type of the variable or constant must be represented by (1) the first letter of a predefined type (e.g., I for INTEGER) or (2) the letter "T" followed by the type glossary number of a defined type, as appropriate.

11. Annotated Pseudo Code

Annotated pseudo code must be used to represent design requirements to be implemented in the body of a subject program unit. The annotated pseudo code must represent the design using text subject to the following use of key words and annotation.

a. Pseudo code for logic and decisions must be introduced and terminated by Ada control statements. These key words include 'if-then,' 'if-then-else,' if-then-elsif,' and 'end if.'

b. Pseudo code for loops must be introduced and terminated by Ada loop statements. These key words include 'loop,' 'end loop,' 'exit,' 'while,' and 'for.' Nested brackets (closing on the right side) must be used to identify the start and stop of program units, loops and conditional clauses.

c. A called program encapsulated in an external package must be signified by a small rectangle containing the name of the external package to the right of the call, with a line drawn from the small rectangle to the pseudo code for the call.

d. Pseudo code for generic instantiation must be bracketed (closing on the right side) with a dashed line.

e. Pseudo code for exception detection must be introduced by the key words 'raise exception' and must be bracketed (closing on the right), where the right side of the bracket is diamond shaped.

f. Pseudo code for exception handling must be introduced by the key words 'exception handler' and must be bracketed (closing on the right), where the right side of the bracket is diamond shaped. A line must be drawn from the 'raise exception' bracket to the 'exception handler' bracket.

APPENDIX B

SAMPLE ABSTRACTED ADA "SKELETON" CODE LISTING

```
with GLOBAL_TYPES;
use GLOBAL_TYPES;
procedure MAIN is

     task COMM_A;
     task body COMM_A is separate;

     task COMM_B;
     task body COMM_B is separate;

     procedure INITIALIZE is separate;
     procedure RESTART is separate;

end MAIN;

-------------------------------------------------------------
-- MAIN, Level 1, Unit a
-------------------------------------------------------------separate
(MAIN)
procedure INITIALIZE is

     procedure PU_M2a is separate;

begin -- INITIALIZE

     . . .

end INITIALIZE;

-------------------------------------------------------------
-- MAIN, Level 1, Unit b
-------------------------------------------------------------
separate (MAIN)
procedure RESTART is

     procedure PU_M2b is separate;
     procedure PU_M2c is separate;

begin -- RESTART

     . . .

end RESTART;

-------------------------------------------------------------
```

```
-- MAIN, Level 2, Unit a
-------------------------------------------------------------
separate (MAIN.INITIALIZE)
procedure PU_M2a is

     procedure PU_M3a is separate;

begin -- PU_M2a

     . . .

end PU_M2a;

-------------------------------------------------------------
-- MAIN, Level 2, Unit b
-------------------------------------------------------------
separate (MAIN.RESTART)
procedure PU_M2b is

begin -- PU_M2b

     . . .

end PU_M2b;

-------------------------------------------------------------
-- MAIN, Level 2, Unit c
-------------------------------------------------------------
separate (MAIN.RESTART)
procedure PU_M2c is

     procedure PU_M3b is separate;
     procedure PU_M3c is separate;

begin -- PU_M2c

     . . .

end PU_M2c;

-------------------------------------------------------------
```

```
-- MAIN, Level 3, Unit a
-------------------------------------------------------
with P3;
use P3;
separate (MAIN.INITIALIZE.PU_M2a)
procedure PU_M3a is

     procedure PU_M4a is separate;

begin -- PU_M3a

     . . .

end PU_M3a;

-------------------------------------------------------
-- MAIN, Level 3, Unit b
-------------------------------------------------------
separate (MAIN.RESTART.PU_M2c)
procedure PU_M3b is

begin -- PU_M3b

     . . .

end PU_M3b;

-------------------------------------------------------
-- MAIN, Level 3, Unit c
-------------------------------------------------------
separate (MAIN.RESTART.PU_M2a)
procedure PU_M3c is

begin -- PU_M3c

     . . .

end PU_M3c;

-------------------------------------------------------
--MAIN, Level 4, Unit a
-------------------------------------------------------
with P6;
use P6;
separate (MAIN.INITIALIZE.PU_M2a.PU_M3a)
procedure PU_M4a is

begin -- PU_M4a
     . . .

end PU_M4a;
-------------------------------------------------------
```

```
-- THREAD A, Level 1, Unit a
-------------------------------------------------------
separate (MAIN)
task body COMM_A is

     procedure PU_A2a is separate;
     task PU_A2b is
          entry A;
          entry B;
     end task PU_A2b;
     task body PU_A2b is separate;

begin -- COMM_A

     . . .

end COMM_A;

-------------------------------------------------------
-- THREAD A, Level 2, Unit b
-------------------------------------------------------
separate (MAIN.COMM_A)
task body PU_A2b is

     procedure PU_A3b is separate;
     procedure PU_A3c is separate;

begin -- PU_A2b

     . . .

end PU_A2b;

-------------------------------------------------------
-- THREAD A, Level 3, Unit a
-------------------------------------------------------
with P7;
use P7;
separate (MAIN.COMM_A.PUA2a)
procedure PU_A3a is

     procedure PU_A4a is separate;
     procedure PU_A4b is separate;

begin -- PU_A3a

     . . .

end PU_A3a;

-------------------------------------------------------
```

```
-- THREAD A, Level 3, Unit b
-------------------------------------------------------------
separate (MAIN.COMM_A.PU_a2B)
procedure PU_A3b is

begin -- PU_A3b

     . . .

end PU_A3b;

-------------------------------------------------------------
-- THREAD A, Level 3, Unit c
-------------------------------------------------------------
with P9;
use P9;
separate (MAIN.COMM_A.PU_A2b)
procedure PU_A3c is

begin -- PU_A3c

     . . .

end PU_A3c;

-------------------------------------------------------------
-- THREAD B, Level 1, Unit a
-------------------------------------------------------------
separate (MAIN)
task body COMM_B is

     procedure PU_B2a is separate;
     task PU_B2b is
          entry C;
          entry D;
     end task PU_B2b;
     task body PU_B2b is separate

begin -- COMM_b

     . . .

end COMM_B;

-------------------------------------------------------------
```

```
-- THREAD B, Level 2, Unit a
-------------------------------------------------------
separate (MAIN.COMM_B)
procedure PU_B2a is

     procedure PU_B3a is separate;

begin -- PU_B2a

     . . .

end PU_B2a;

-------------------------------------------------------
-- THREAD B, Level 2, Unit b
-------------------------------------------------------
separate (MAIN.COMM_B)
task body PU_B2b is

     procedure PU_B3b is separate;
     procedure PU_B3c is separate;

begin -- PU_B2b

     . . .

end PU_B2b;

-------------------------------------------------------
-- THREAD B, Level 3, Unit a
-------------------------------------------------------
separate (MAIN.COMM_B.PU_B2a)
procedure PU_B3a is

     procedure PU_B4a is separate;

begin -- PU_B3a

     . . .

end PU_B3a;

-------------------------------------------------------
```

```
-- THREAD B, Level 3, Unit a
-------------------------------------------------------------
separate (MAIN.COMM_B. PU_B2a)
procedure PU_B3a is

     procedure PU_B4a is separate;

begin -- PU_B3a

     . . .

end PU_B3a;

-------------------------------------------------------------
-- THREAD B, Level 3, Unit b
-------------------------------------------------------------
with P9;
use P9;
separate (MAIN.COMM_B.PU_B2b)
procedure PU_B3c is

     procedure PU_B4c is separate;

begin -- PU_B3c

     . . .

end PU_B3c;

-------------------------------------------------------------
-- THREAD B, Level 4, Unit a
-------------------------------------------------------------
separate (MAIN.COMM_B.PU_B2a. PU_B3a)
procedure PU_B4a is

begin -- PU_B4a

     . . .

end PU_B4a;

-------------------------------------------------------------
```

```
-- THREAD B, Level 4, Unit b
-------------------------------------------------------
separate (MAIN.COMM_B. PU_B2b. PU_B3b)
procedure PU_B4b is

begin -- PU_B4b

     . . .

end PU_B4b;

-------------------------------------------------------
-- THREAD B, Level 4, Unit c
-------------------------------------------------------
with P10;
use P10;
separate (MAIN.COMM_B.PU_B2b. PU_B3c)
procedure PU_B4c is

begin -- PU_B4c

     . . .

end PU_B4c;

-------------------------------------------------------
-- PACKAGE P3
-------------------------------------------------------
package P3 is

     . . .

end P3;
package body P3 is

     . . .

end P3;

-------------------------------------------------------
```

```
-- PACKAGE P6
-------------------------------------------------------------
package P6 is

     . . .

end P6;
package body P6 is

     . . .

end P6;

-------------------------------------------------------------
-- PACKAGE P7
-------------------------------------------------------------
package P7 is

     . . .

end P7;
package body P7 is

     . . .

end P7;

-------------------------------------------------------------
-- PACKAGE P8
-------------------------------------------------------------
package P8 is

     . . .

end P8;
package body P8 is

     . . .

end P8;

-------------------------------------------------------------
```

```
-- PACKAGE P9
-------------------------------------------------------------
package P9 is

     . . .

end P9;
package body P9 is

     . . .

end P9;

-------------------------------------------------------------
-- PACKAGE P10
-------------------------------------------------------------
package P10 is

     . . .

end P10;
package body P10 is

     . . .

end P10;

-------------------------------------------------------------
```

APPENDIX C

REQUIREMENTS FOR A HYPOTHETICAL SPACE STATION COMPUTER PROGRAM

1.0 INTRODUCTION

This specification presents requirements for a computer program to operate in a hypothetical space station. The computer program shall provide capabilities for:

a. Experiment data collection and processing

b. Environmental sensor monitoring and alarm generation for out of bound environmental readings

c. Space station solar panel orientation correction by command from ground control stations

d. Built-in test of the computer hardware

1.1 EXPERIMENT DATA COLLECTION AND PROCESSING

1.1.1 Data Collection and Storage

The program shall collect data for three experiments and assemble data samples. A data base shall be established to store experiment data samples in local memory for immediate processing. Provisions shall be for archiving an experiment's data samples in a mass memory medium or when the allocated local memory is full.

1.1.2 Operator Interface

A work station interface shall be provided for operator input. The operator shall be able to enter commands for viewing available data samples or initiating statistical processing of a data sample.

1.1.3 Data Processing

The operator shall be able to initiate the calculation of a data sample's mean, standard deviation, and statistical distributions (i.e., normal or Poisson). The results of the calculations shall be displayed on the work station.

1.2 ENVIRONMENTAL SENSOR MONITORING

1.2.1 Sensor Interface

The program shall interface with sensor hardware to take power, temperature and pressure readings for the space station.

1.2.2 Sensor Checking

A range of suitable sensor readings shall be established by the computer program (default boundaries) or shall be provided by operator chosen input. Sensor readings shall be compared with the range of suitable values and an alarm generated if a sensor reading is out of bounds.

1.2.3 Operator Interface

An interface with a work station shall be provided for operator setting of sensor value ranges and for display of alarm messages.

1.2.4 Alarm Report

An alarm report shall be generated at the work station operator console if sensor checking makes a sensor out of bounds determination. The alarm message shall also be recorded in mass storage.

1.3 SOLAR PANEL ORIENTATION CORRECTION

1.3.1 Update Orientation

The program shall receive an angle pair from earth ground stations giving the correction for the solar panel orientation. The new solar panel orientation shall be calculated from the old orientation. The new orientation shall be transmitted to earth.

1.3.2 Solar Panel Arm Motions

The program shall calculate the needed solar panel mounting arm motions needed to orient the solar panel, given the old orientation and the new orientation. The program shall generate control information to two solar panel mounting panel motors to reorient the solar panel.

1.4 COMPUTER PROCESSOR BUILT-IN TEST

A test suite of Ada instructions shall be run to monitor the fitness of the computer processor. The test suite shall exercise the computer processor instruction code and memory use. Unexpected results shall be reported at the sensor monitoring work station.

APPENDIX D

ACCOUNTING FOR CHARACTERISTICS OF A SOFTWARE ACQUISITION USING COCOMO

1 INTRODUCTION

COCOMO utilizes coefficients to account for characteristics of a software acquisition. Specifically, the estimation algorithms of Intermediate and Detailed COCOMO contain 15 coefficients. Each coefficient is a function of a different attribute of the software acquisition being estimated. As described in Section 2.3.1 of Chapter VII, coefficients account for the characteristics of the software product to be developed, the computer to be used, the personnel to do the work, and other project considerations.

2 VALUES OF COCOMO ATTRIBUTE COEFFICIENTS USED IN INTERMEDIATE COCOMO

2.1 Estimating Development Costs and Schedule

When estimating development costs and schedule duration, COCOMO Attribute Coefficients are given by the relationship:

$$C_n = f(A_n)$$

where

A_1 = required software reliability
A_2 = data base size
A_3 = product complexity
A_4 = execution time constraints
A_5 = main storage constraint
A_6 = virtual machine volatility
A_7 = computer turnaround time
A_8 = analyst capability
A_9 = applications experience
A_{10} = programmer capability
A_{11} = virtual machine experience
A_{12} = programming language experience
A_{13} = use of modern programming practices
A_{14} = level of tool support
A_{15} = schedule constraint

The set of values possible for each attribute, the meaning of each value, and the corresponding coefficient value is given in Table D-1.

2.2 Estimating Maintenance Costs

When estimating maintenance costs and schedule duration, the coefficient values are selected from Table D-1, except for the coefficients C_1, C_{13} and C_{15}. These coefficients have unique values for maintenance. The unique values are given in Table D-2.

TABLE D-1

Possible Ratings for COCOMO Attribute Inputs
Numerical Representations of Related Coefficients *

Coeff.	Input Attribute	Permissible Ratings	Coeff. Values	Meaning	
C_1	Required Software Reliability	Very High	1.40	Risk to human life	If software fails
		High	1.15	High financial loss	
		Nominal	1.00	Moderate recoverable loss	
		Low	0.88	Easily recoverable loss	
		Very Low	0.75	Slight inconvenience	
C_2	Data Base Size	Very High	1.16	$D/DSI \geq 1000$	D = data base size in bytes
		High	1.08	$100 \leq D/DSI < 1000$	
		Nominal	1.00	$10 \leq D/DSI < 100$	DSI = delivered source instructions
		Low	0.94	$D/DSI < 10$	
C_3	Product Complexity	Extra High	1.65	Multiple resource scheduling, dynamic priorities	
		Very High	1.30	Reentrant, recursive code. Fixed priority interrupt handling	
		High	1.15	Highly nested, compound predicates, intermodule control	
		Nominal	1.00	Simple nesting, some intermodule control, decision tables	
		Low	0.85	Straight forward nesting, simple predicates	
		Very Low	0.70	Straight-line code with a little nesting, simple predicates	
C_4	Execution Time Constraint	Extra High	1.66	95%	Use of available execution time
		Very High	1.30	85%	
		High	1.11	70%	
		Nominal	1.00	50% or less	
C_5	Main Storage Constraint	Extra High	1.56	95%	Use of available storage
		Very High	1.21	85%	
		High	1.06	70%	
		Nominal	1.00	50% or less	
C_6	Virtual Machine Volatility	Very High	1.30	2 days to 2 weeks	Time period between changes
		High	1.15	1 week to 2 months	
		Nominal	1.00	2 weeks to 6 months	
		Low	0.87	1 month to 12 months	
C_7	Computer Turnaround Time	Very High	1.15	12 or more	Average response time (hours)
		High	1.07	4 to 12	
		Nominal	1.00	less than 4	
		Low	0.87	interactive	
C_8	Analyst Capability	Very High	1.46	90th	Percentile
		High	1.19	75th	
		Nominal	1.00	55th	
		Low	0.86	35th	
		Very Low	0.71	15th	
C9	Applications Experience	Very High	0.82	12 years or more	Average experience
		High	0.91	6 years	
		Nominal	1.00	3 years	
		Low	1.14	1 year	
		Very Low	1.29	4 months or less	

TABLE D-1 (CONCLUDED)

Coeff.	Input Attribute	Permissible Ratings	Coeff. Value	Meaning	
C_{10}	Programmer Capability	Very High	0.70	90th	Percentile
		High	0.86	75th	
		Nominal	1.00	55th	
		Low	1.17	35th	
		Very Low	1.42	15th	
C_{11}	Virtual Machine Experience	High	0.90	3 years or more	Average experience
		Nominal	1.00	1 year	
		Low	1.10	4 months	
		Very Low	1.21	1 month or less	
C_{12}	Programming Language Experience	High	0.95	3 years or more	Average experience
		Nominal	1.00	1 year	
		Low	1.07	4 months	
		Very Low	1.14	1 month or less	
C_{13}	Use of Modern Programming Practices	Very High	0.82	Heavy and efficient	
		High	0.91	Above average	
		Nominal	1.00	Average	
		Low	1.10	Beginning	
		Very Low	1.24	None	
C_{14}	Level of Tool Support	Very High	0.83	Advanced maxicomputer tools	
		High	0.91	Strong maxicomputer tools	
		Nominal	1.00	Strong minicomputer tools	
		Low	1.10	Basic minicomputer tools	
		Very Low	1.24	Basic microprocessor tools	
C_{15}	Schedule Constraint	Very High	1.10	160% or more of nominal	
		High	1.04	130% of nominal	
		Nominal	1.00	Nominal	
		Low	1.08	85% of nominal	
		Very Low	1.23	75% of nominal	

* The numerical values shown are for software development work. When estimating software maintenance costs, the same coefficients are applicable with the following exceptions:

C_1 = 1.10 (very high) or 0.98 (high) or 1.00 (nominal) or 1.15 (low) or 1.35 (very low)

C_{15} = 1.00

C_{13} varies as follows:

DSI/1000	(very high)	(high)	(nominal)	(low)	(very low)
2	0.81	0.90	1.00	1.12	1.25
8	0.77	0.88	1.00	1.14	1.30
32	0.74	0.86	1.00	1.16	1.35
128	0.72	0.85	1.00	1.18	1.40
512	0.70	0.84	1.00	1.20	1.45

TABLE D-2 POSSIBLE RATINGS FOR COCOMO ATTRIBUTE INPUT AND RELATED COCOMO COEFFICIENTS UNIQUE TO MAINTENANCE

Coefficient	Input Attribute	Permissible Rating	Coefficient Value	Product Size
C_1	Required Software Reliability	Very High	1.10	
		High	0.98	
		Nominal	1.00	N/A
		Low	1.15	
		Very Low	1.35	
C_{13}	Use of Modern Programming Practices	Very High	0.81	2,000
		High	0.90	
		Nominal	1.00	
		Low	1.12	
		Very Low	1.25	
		Very High	0.77	8,000
		High	0.88	
		Nominal	1.14	
		Low	1.14	
		Very Low	1.30	
		Very High	0.74	32,000
		High	0.86	
		Nominal	1.00	
		Low	1.16	
		Very Low	1.35	
C_{13}	Use of Modern Programming Practices	Very High	0.72	128,000
		High	0.85	
		Nominal	1.00	
		Low	1.18	
		Very Low	1.40	
		Very High	0.70	512,000
		High	0.84	
		Nominal	1.00	
		Low	1.20	
		Very Low	1.45	
C_{15}	Schedule Constraint	Very High	1.00	N/A
		High	1.00	
		Nominal	1.00	
		Low	1.00	
		Very Low	1.00	

3 VALUES OF COCOMO ATTRIBUTE COEFFICIENTS USED IN DETAILED COCOMO

3.1 Coefficients Variable at the Module Level

COCOMO coefficients for Detailed COCOMO vary by phase. Four of them also typically vary from module to module. Specifically, C_3 as a function of product complexity, C_{10} as a function of programming capability, C_{11} as a function of virtual machine experience and C_{12} as a function of programming language experience vary by both phase and module. The estimator selects values for them for each specific module from the values given in Table D-3.

3.2 Coefficients Fixed at the Module Level

With detailed COCOMO certain coefficients vary by phase but are relatively fixed from one module to another. These coefficients are shown in Table D-4.

TABLE D-3 DETAILED COCOMO, ATTRIBUTE COEFFICIENTS VARIABLE BY MODULE AND PHASE

	Permissible	Phase Number (p)			
C_3	Very Low	.70	.70	.70	.70
	Low	.85	.85	.85	.85
	Nominal	1.00	1.00	1.00	1.00
	High	1.15	1.15	1.15	1.15
	Very High	1.30	1.30	1.30	1.30
	Extra High	1.65	1.65	1.65	1.65
C_{10}	Very Low	1.50	1.50	1.50	1.50
	Low	1.00	1.20	1.20	1.20
	Nominal	1.00	1.00	1.00	1.00
	High	1.00	.83	.83	.83
	Very High	1.00	.65	.65	.65
C_{11}	Very Low	1.10	1.10	1.30	1.30
	Low	1.05	1.05	1.15	1.15
	Nominal	1.00	1.00	1.00	1.00
	High	.90	.90	.90	.90
C_{12}	Very Low	1.02	1.10	1.20	1.20
	Low	1.00	1.05	1.10	1.10
	Nominal	1.00	1.05	1.10	1.10
	High	1.00	.98	.92	.92

p = 1 for the product design phase

p = 2 for the detailed design phase

p = 3 for the code and unit test phase

p = 4 for the integration and test phase

TABLE D-4 DETAILED COCOMO, ATTRIBUTE COEFFICIENTS FIXED BY MODULE AND VARIABLE BY PHASE

Coefficient	Permissible Rating	Phase Number (p) 1&2	3	4	5
C_1	Very Low	.80	.80	.80	.80
	Low	.90	.90	.90	.90
	Nominal	1.00	1.00	1.00	1.00
	High	1.10	1.10	1.10	1.30
	Very High	1.30	1.30	1.30	1.70
C_2	Low	.95	.95	.95	.90
	Nominal	1.00	1.00	1.00	1.00
	High	1.10	1.05	1.05	1.15
	Very High	1.20	1.10	1.10	1.30
C_4	Nominal	1.00	1.00	1.00	1.00
	High	1.10	1.10	1.10	1.15
	Very High	1.30	1.25	1.25	1.40
	Extra High	1.65	1.55	1.55	1.95
C_5	Nominal	1.00	1.00	1.00	1.00
	High	1.05	1.05	1.05	1.10
	Very High	1.20	1.15	1.15	1.35
	Extra High	1.55	1.45	1.45	1.85
C_6	Low	.95	.90	.85	.80
	Nominal	1.00	1.00	1.00	1.00
	High	1.10	1.12	1.15	1.20
	Very High	1.20	1.25	1.30	1.40
C_7	Low	.98	.95	.70	.90
	Nominal	1.00	1.00	1.00	1.00
	High	1.00	1.00	1.10	1.15
	Very High	1.02	1.05	1.20	1.30
C_8	Very Low	1.80	1.35	1.35	1.50
	Low	1.35	1.15	1.15	1.20
	Nominal	1.00	1.00	1.00	1.00
	High	.75	.90	.90	.85
	Very High	.55	.75	.75	.70
C_9	Very Low	1.40	1.30	1.25	1.25
	Low	1.20	1.15	1.10	1.10
	Nominal	1.00	1.00	1.00	1.00
	High	.87	.90	.92	.92
	Very High	.75	.80	.85	.85

TABLE D-4 DETAILED COCOMO, ATTRIBUTE COEFFICIENTS FIXED BY MODULE AND VARIABLE BY PHASE (continued)

Coefficient	Permissible Rating	Phase Number (p) 1&2	3	4	5
C_{13}	Very Low	1.05	1.10	1.25	1.50
	Low	1.00	1.05	1.10	1.20
	Nominal	1.00	1.00	1.00	1.00
	High	1.00	.95	.90	.83
	Very High	1.00	.90	.80	.65
C_{14}	Very Low	1.02	1.05	1.35	1.45
	Low	1.00	1.02	1.15	1.20
	Nominal	1.00	1.00	1.00	1.00
	High	.90	.95	.90	.85
	Very High	.95	.90	.80	.70
C_{15}	Very Low	1.10	1.25	1.25	1.25
	Low	1.00	1.15	1.15	1.10
	Nominal	1.00	1.00	1.00	1.00
	High	1.10	1.10	1.00	1.00
	Very High	1.15	1.15	1.05	1.05

REFERENCES

1. Buhr, R. J. A., System Design With Ada, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 02632 (1985).

2. Young, S. J., An Introduction to Ada, John Wiley & Sons, New York, N.Y. 1983

3. R. G. Howe, et al, Program Manager's Guide to Ada, ESD-TR-85-159, The MITRE Corporation, Bedford, Massachusetts (1985). AD B092503L

4. Booch, G., "Object-Oriented Development," IEEE Transactions on Software Engineering, Volume SE-12, No. 2 (1986).

5. Barnes, J. G., Programming in Ada, Addison-Wesley Publishing Company, London (1982).

6. Booch, G., Software Engineering With Ada, Benjamin/Cummings Publishing Company, Inc., Menlo Park, California (1983).

7. EVB Software Engineering, An Object Oriented Design Handbook for Ada Software, EVB Software Engineering, Inc., Rockville, Maryland (1985).

8. Boehm, B. W., Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1981).

9. G. A. Miller, "The Magical Number Seven, Plus or Minus Two", The Physchological Review, Volume 63, No. 2, March 1956.

10. Clapp, J. A., A Study of Computer Resource Utilization in ESD Weapon System Acquisitions, AD A125641, The MITRE Corporation (1982).

11. Najberg, A. C. and Healy, R. D., "The Impact of Ada on Software Development Costs", The Analytical Sciences Corporation, Reading, Massachusetts (1984).

12. "Software Productivity", Volume 1, Number 1, Software Productivity Research, Inc., (1986).

13. John H. James, "Validation of Some Software Cost Estimation Models", The MITRE Corporation (1984).