505195DA

AD-A286 826

AMSAA

SOFTWARE ENGINEERING IN Ada

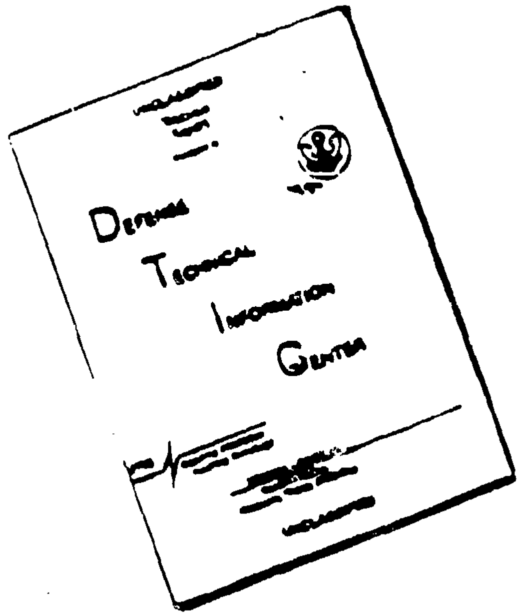
Presented by: Capt David Vega
3390th Technical Training Group
Keesler Air Force Base, MS

Sponsored by: Ada Joint Program Office (OSD)

Organized by: Herbert E. Cohen
US Army Materiel Systems
Analysis Activity
Aberdeen Proving Ground,
Maryland

U. S. ARMY MATERIEL SYSTEMS ANALYSIS ACTIVITY
ABERDEEN PROVING GROUND, MARYLAND

DISCLAIMER NOTICE



Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION AMSAA 6b. OFFICE SYMBOL (If applicable) AMXSY-MP	7a. NAME OF MONITORING ORGANIZATION Same as 6
6c. ADDRESS (City, State, and ZIP Code) Aberdeen Proving Ground, MD 21005-5071	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Ada Joint Program Office (OSD) 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) The Pentagon Washington, D.C. 20301-3081	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Software Engineering in Ada (u).

12. PERSONAL AUTHOR(S)

Cohen (Organizer)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM ______ TO ______	88/03/22	331

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Fundementals in Ada, types, control structures, sub programs, packages, exceptions, generics, tasks, program design

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

Provides a detailed course in Software Engineering in Ada.

95-01574

DTIC QUALITY INSPECTED 3

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Herbert F. Cohen	22b. TELEPHONE (Include Area Code) (301) 278-2785/6577 22c. OFFICE SYMBOL AMSAA (AMXSY-MP)

DD Form 1473, JUN 86 — Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

ACKNOWLEDGEMENTS

I would like to take this opportunity in behalf of the Ada Joint Program Office (OSD) and the US Army Materiel Systems Analysis Activity (AMSAA) to express my deep appreciation to CPT David Vega of the 3390th Technical Training Group, Keesler Air Force Base, Mississippi for an outstanding lecture series in software engineering. Mr. Lou Puckett of the 3300th Technical Training Wing and CPT William Frey of the 3390th Technical Training Group at Kessler AFB provided invaluable assistance in coordinating this program for which I am also deeply indebted. The producer/director of the video production, Mr. Jim Blum of Det2, 1365 AV at Keesler AFB, did an outstanding professional job.

The road to final production of these tapes was long and hard but it could not have been achieved without the support of two distinguished officers from the Ada Joint Program Office (OSD). My very sincere appreciation is extended to LTC(P) David Taylor and MAJ Allen Kopp (AF) of the AJPO (OSD) for their support and wish them the very best in their future assignments.

Herbert E. Cohen
US Army Materiel Systems Analysis
Activity
Aberdeen Proving Ground, MD

REQUEST FOR VIDEO TAPES AND TEXT

1. DOD and other government agencies may obtain copies of tapes and text through the nearest local Training and Audio Visual Support Center.

2. Reference tapes by SAV PIN# 505195.

3. Army/Navy and other government agencies should request tapes by writing:

Department of the Army
US Army Visual Information Center
Joint Visual Information Activity
ATTN: ASNV-OJVP-CM
Tobyhanna Army Depot, PA 18466-5102

PHONE: (717) 590-7063
[illegible]

4. Air Force activities can request tapes by writing:

AFCVIL
1352nd AVS/DOSQ
Bldg #248
Norton AFB, CA 92409-5996

5. Tapes will be in standard DOD 3/4 inch video cassette; however, 1/2 inch VHS formats may also be available on request.

6. The general public can obtain tapes at minimal cost, in any of the formats specified above, by writing to:

National Audio Visual Center
GSA
ATTN: Order Section
Washington, DC 20409

7. For additional information, contact:

Ada Joint Program Office
Rm 3E114
The Pentagon
Washington, DC 20301-3081

PHONE: (202) 694-0210
AUTOVON 224-0210

or

Director
US Army Materiel Systems Analysis Activity
ATTN: AMXSY-MP (Herbert Cohen)
Aberdeen Proving Ground, MD 21005-5071

PHONE: (301) 278-2785/6577
AUTOVON 298-2785/6577

LECTURER

CPT David Vega

3390th Technical Training Group
Keesler Air Force Base, MS

TEXT

Text No. 1. Fundamentals of Ada Programming/Software Engineering (Note-tasking Guide) - 90P-890

Text No. 2. Fundamentals of Ada Programming/Software Engineering (Study Guide/Workbook) - 90P-893

Text No. 3. Object Oriented Design - 90P-886

HANDOUT 90P-890

E3OAR4924 004
E4OST4924 020

Technical Training

Fundamentals of Ada Programming/Software Engineering

[illegible]

October 1987

USAF TECHNICAL TRAINING SCHOOL
3390 Technical Training Group
Keesler Air Force Base, Mississippi 39534-5000

Authorized for ATC Course Use
DO NOT USE ON THE JOB

3390 Technical Training Group
Keesler AFB, Mississippi 39534-5000

HO 90P-890
E3OAR4924 004
E4OST4924 020
December 1987

NOTE-TAKING GUIDE

Philosophy

The philosophy of the Wing emerges from a deep concern for individual Air Force men and women and the need to provide highly trained and motivated personnel to sustain the mission of the Air Force. We believe the abilities, worth, self-respect, and dignity of each student must be fully recognized. We believe each must be provided the opportunity to pursue and master an occupational specialty to the full extent of the individual's capabilities and aspirations for the immediate and continuing benefit of the individual, the Air Force, DOD, and the country. To these ends, we provide opportunities for individual development of initial technical proficiencies, on-the-job training in challenging job assignments, and follow-on growth as supervisors. In support of this individual development, and to facilitate maximum growth of its students, the Wing encourages and supports the professional development of its faculty and administrators, and actively promotes innovation through research and the sharing of concepts and material with other educational institutions.

Contents

FUNDAMENTALS OF Ada SYSTEMS

- Software Engineering
- Ada Language Features F-1
- Program Library
- Simple Control Structures
- Simple Input/Output
- Host Computer Operations

SOFTWARE ENGINEERING

THE CRITICALITY OF SOFTWARE

- Hardware is no longer the dominant factor in the hardware/software relationship
  - — Cost
  - — Technology F-2
- The demand for software is rising exponentially
- The cost of software is rising exponentially
- Software maintenance is the dominant software activity
- Systems are getting more complex
- Life and property are dependent on software

CHARACTERISTICS OF [illegible]D SOFTWARE

- Expensive
- Incorrect
- Unreliable F-3
- Difficult to predict
- Unmaintainable
- Not reusable

Student Notes:

F 4

FACTORS AFFECTING DOD SOFTWARE

- Ignorance of life cycle implications
- Lack of standards
- Lack of methodologies
- Inadequate support tools
- Management
- Software professionals

F-5

CHARACTERISTICS OF DOD SOFTWARE REQUIREMENTS

- Large
- Complex
- Long lived
- High reliability
- Time constraints
- Size constraints

F-6

THE FUNDAMENTAL PROBLEM

- Our inability to manage the COMPLEXITY of our software systems (G. Booch)
- Lack of a disciplined, engineering approach

Student Notes:

SOFTWARE ENGINEERING

THE ESTABLISHMENT AND APPLICATION OF SOUND ENGINEERING =>

- Environments
- Tools
- Methodologies
- Models
- Principles
- Concepts

F-7

SOFTWARE ENGINEERING

COMBINED WITH -->

- Standards
- Guidelines
- Practices

F-8

SOFTWARE ENGINEERING

TO SUPPORT COMPUTING WHICH IS =>

- Understandable
- Efficient
- Reliable and safe
- Modifiable
- Correct

F-9

THROUGHOUT THE LIFE CYCLE OF A SYSTEM

(C. McKay, 1985)

Student Notes:

SOFTWARE ENGINEERING

F 10

- Purposes
- Concepts
- Mechanisms
- Notation
- Usage

SEI, Sep 1986

PURPOSES

11

- Create software systems according to good engineering [illegible]
- Manage elements within the software life cycle

CONCEPTS

F-12

- Derive the architecture of software systems
- Specify modules of the system

MECHANISMS

- Tools for:
  - Writing operating systems
  - Tuning software
  - Prototyping
- Techniques for:
  - Managing projects
  - Systems analysis
  - Systems design
- Standards for:
  - Coding
  - Metrics
  - Human and machine interfacing

F-13

NOTATION

- Languages for writing linguistic models
- Documentation

F-14

USAGE

- Embedded systems
- Data processing
- Control
- Expert systems
- Research and development
- Decision support
- Information management

F-15

Student Notes:

CONTENT AREAS

F-16

- Communication skills
- Software development and evolution processes
- Problem analysis and specification
- System design
- Data Engineering
- Software generation
- System quality
- Project management
- Software engineering projects

SEI, June 1986

PROGRAMMING LANGUAGES AND SOFTWARE ENGINEERING

F-17

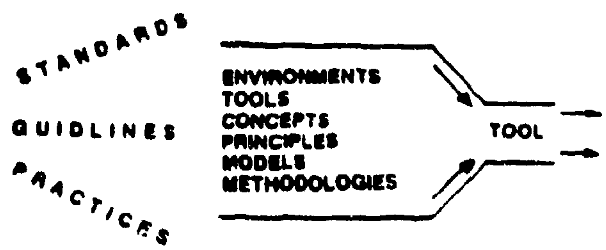
- A programming language is a software engineering tool
- A programming language EXPRESSES and EXECUTES design methodologies
- The quality of a programming language for software engineering is determined by how well it supports a design methodology and its underlying models, principles, and concepts

TRADITIONAL PROGRAMMING LANGUAGES AND SOFTWARE ENGINEERING

F-18

Programming Languages

- Were not engineered
- Have lacked the ability to express good software engineering
- Have failed to constrain software engineering



Student Notes:

Ada
AND
SOFTWARE ENGINEERING

Ada
- Was itself "engineered" to support software engineering
- Embodies the same concepts, principles, and models to support methodologies F-19
- Is the best tool (programming language) for software engineering currently available

	ENVIRONMENTS	
STANDARDS	TOOLS	T
	CONCEPTS	O
GUIDLINES	PRINCIPLES	
	MODELS	O
PRACTICES	METHODOLOGIES	L

PRINCIPLES OF SOFTWARE ENGINEERING

- Abstraction
- Modularity
- Localization F-20
- Information hiding
- Completeness
- Confirmability
- Uniformity

(Ross, Goodenough, Irvine, 1975)

ABSTRACTION

- The process of separating out the important parts of something while ignoring the inessential details
- Separates the "what" from the "how" F-21
- Reduces the level of complexity
- There are levels of abstraction within a system

Student Notes:

MODULARITY

F-22

- Purposeful structuring of a system into parts which work together
- Each part performs some smaller task of the overall system
- Can concentrate and develop parts independently as long as interfaces are defined and shared
- Can develop hierarchies of management and implementation

LOCALIZATION

- Putting things that logically belong together in the same physical place

F-23

INFORMATION HIDING

- Puts a wall around localized details
- Prevents reliance upon details and causes focus of attention to interfaces and logical properties

COMPLETENESS

- Ensuring all important parts are present
- Nothing left out

F-24

CONFIRMABILITY

- Developing parts that can be effectively tested

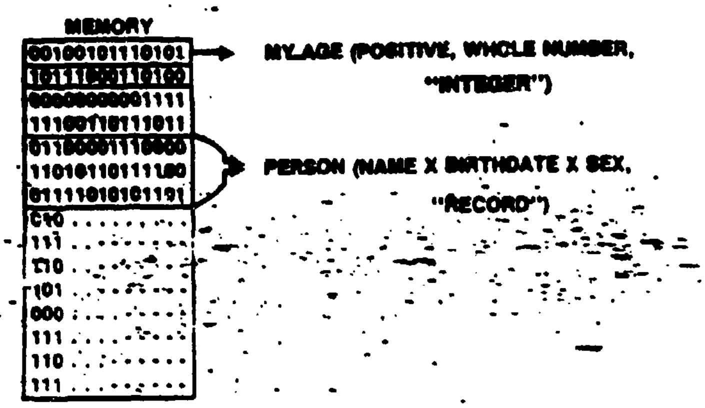
UNIFORMITY

- No unnecessary differences across a system

Ada LANGUAGE FEATURES

DATA TYPING

- **The imposition of structure on data values manipulated by a programming language**



F-25

- A data type defines a set of values that objects of the type may assume and the set of operations that may manipulate them.

TYPE	VALUES	OPERATIONS
AGE_TYPE	Positive, Exact Numbers	+, −, /, * ...
PERSON_TYPE	Names x Birthdates x Sex	Examine Name of Person, Examine Sex, Assignment ...

F-26

DESIRABLE REASONS TO TYPE DATA

- Factorization of Properties, Maintainability
- Reliability
- Abstraction, Information Hiding

(Rationale for the Design of the Ada Programming Language)

F-27

Student Notes:

STRONG TYPING

F-28

- Ada is a strongly typed language
  - All objects must be declared to be of a particular type
  - Different types may not be implicitly mixed
  - Operations on a type must preserve that type (remain within set of values)

MY_AGE + PERSON -- ILLEGAL

TYPE DECLARATION

F-29

- Creates a type name
- Specifies the set of values and set of operations for the type

type TYPE_NAME is ["set of values and operations"]

TYPE DECLARATION

F-30

TYPE	VALUES	OPERATIONS
AGE_TYPE	0, 1, 2...130	Those applicable to integer values
MONEY_TYPE	Real values between 0.0 and 100.0	Those applicable to real values

MAX_AGE : constant := 130

type AGE_TYPE is range 0 .. MAXIMUM_AGE

Student Notes:

OBJECT DECLARATION

- An instance of a given type
- A name for a storage location whose structure is that defined for the type

F-31

```
MY_AGE    : AGE_TYPE;
YOUR_AGE  : AGE_TYPE;
NO_MONEY  : constant MONEY_TYPE := 0.0;
```

```
-- A simple program that adds three
-- ages together

procedure ADD_AGES_TOGETHER is

   MAX_AGE     : constant := 130;
   type AGE_TYPE is range 0 .. MAX_AGE;
   JOHNS_AGE   : AGE_TYPE := 10;
   MARYS_AGE   : AGE_TYPE := 40;
   JANS_AGE    : AGE_TYPE := 20;
   TOTAL       : AGE_TYPE := 0;

begin

   TOTAL := JOHNS_AGE + MARYS_AGE + JANS_AGE;
end  ADD_AGES_TOGETHER;
```

F-32

CLASSES OF Ada TYPES

- Scalar
  - Discrete
    - Integer Types
    - Enumerated Types
  - Real
    - Fixed Point
    - Floating Point

F-33

Student Notes:

F-34

- Composite
  - Array
  - Record
- Access
- Private
  - Private
  - Limited
- Task

F-35

SYSTEMS ENGINEERING

- Analyze problem
- Break into solvable parts
- Implement parts
- Test parts
- Integrate parts to form total system
- Test total system

F-36

REQUIREMENTS FOR EFFECTIVE SYSTEMS ENGINEERING

- Ability to express architecture
- Ability to define and enforce interfaces
- Ability to create independent components
- Ability to separate architectural issues from implementation issues

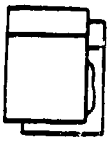
Student Notes:

PROGRAM UNITS

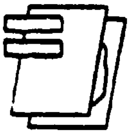
- Components of Ada which together form a working Ada software system
- Express the architecture of a system
- Define and enforce interfaces

F-37

PROGRAM UNITS



SUBPROGRAMS — Working components that perform some action



TASKS — Performs actions in parallel with other program units

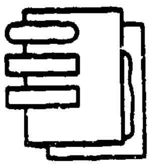
PACKAGES — A mechanism for collecting entities together into logical units

F-38

PROGRAM UNITS

- Consist of two parts: specification and body

SPECIFICATION: Defines the interface between the program unit and other program units (the WHAT)



BODY: Defines the implementation of the program unit (the HOW)

F-39

Student Notes:

F-40

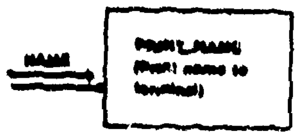
PROGRAM UNITS

- The specification of the program unit is the only means of connecting program units
- The interface is enforced
- The body of a program unit is not accessible to other program units
- There is a clear distinction between architecture and implementation

F-41

ABSTRACT ACTIONS

- Perform some discrete activity



PRINT_NAME (NAME);

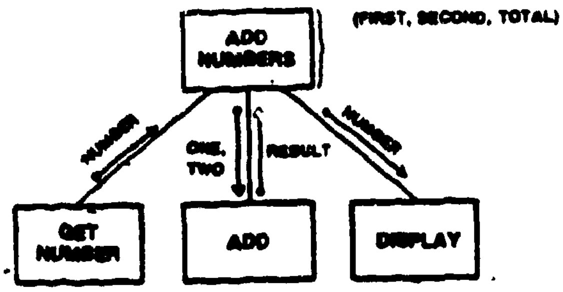
ADD (COUNT_1, COUNT_2, RESULT);

F-42

DISCRETE COMPONENTS

- Allow a system to be composed of black boxes
- Provide clear, understandable functions
- Black boxes can be more effectively validated and verified
- Prevalent across engineering disciplines

Student Notes:




```
ADD_NUMBERS

GET_NUMBER (FIRST)
GET_NUMBER (SECOND)
ADD (FIRST, SECOND, TOTAL)
DISPLAY (TOTAL)
```

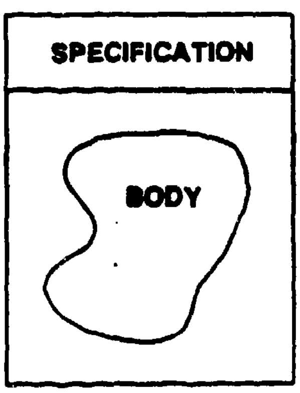
F-43

SUBPROGRAMS

- A program unit that performs a particular action
  - — Procedures
  - — Functions
- Contains an interface (parameter part) mechanism to pass data to and from the subprogram
- The basic discrete component which acts like a black box
- Gives ability to express abstract actions

F-44

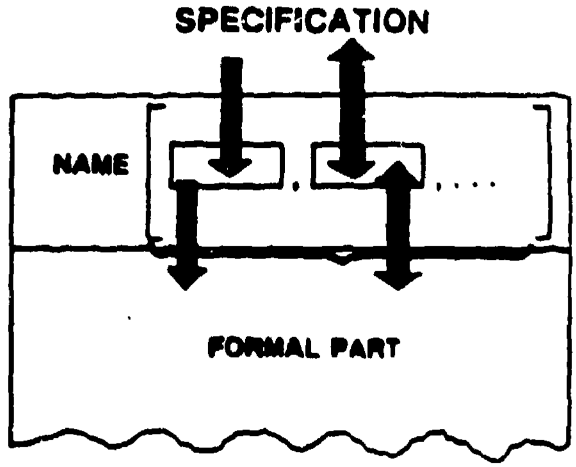
SUBPROGRAM STRUCTURE



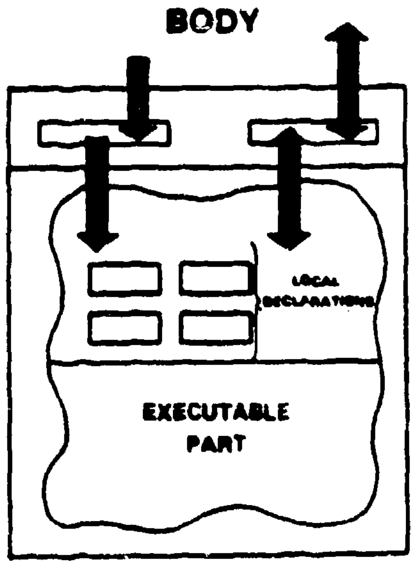
F-45

Student Notes:

F-46



F-47



F-48

Local Declarations ➤

Executable Part ➤

```
procedure  ADD_NUMBERS is

  MAX_NUM : constant     := 40;
  type NUMBER_TYPE is range 0 .. MAX_NUM;
  NUMBER_1, NUMBER_2, NUMBER_3,
  TOTAL    : NUMBER_TYPE := 0;

begin

  NUMBER_1 := 1;
  NUMBER_2 := NUMBER_1 + 1;
  NUMBER_3 := NUMBER_2 + 1;
  TOTAL := NUMBER_1 + NUMBER_2 + NUMBER_3;

end  ADD_NUMBERS;
```

```
procedure ADD_NUMBERS is

  MAX_NUM : constant    := 40;
  type NUMBER_TYPE is range 0 .. MAX_NUM;
  NUMBER_1, NUMBER_2, NUMBER_3,
  TOTAL   : NUMBER_TYPE := 0;
  procedure INCREMENT
    (A_NUMBER : in out NUMBER_TYPE)
                          is separate;
begin

   INCREMENT (NUMBER_1);
   NUMBER_2 := NUMBER_1;
   INCREMENT (NUMBER_2);
   NUMBER_3 := NUMBER_2;
   INCREMENT (NUMBER_3);
   TOTAL    := NUMBER_1 + NUMBER_2 + NUMBER_3;

end ADD_NUMBERS;
```

```
separate (ADD_NUMBERS)

procedure INCREMENT
  (A_NUMBER : in out NUMBER_TYPE) is

begin

  A_NUMBER := A_NUMBER +1;

end INCREMENT;
```

```
with TEXT_IO;

procedure SAY_HI is
  MAX_NAME_LENGTH : constant  := 80;
  subtype NAME_TYPE is STRING
    (1..MAX_NAME_LENGTH);
  YOUR_NAME   : NAME_TYPE := (others => ' ' );
  NAME_LENGTH : NATURAL   := 0;

begin

  TEXT_IO.PUT_LINE("What is your name?");
  TEXT_IO.GET_LINE(YOUR_NAME, NAME_LENGTH);
  TEXT_IO.PUT("Hi");
  TEXT_IO.PUT_LINE(YOUR_NAME(1..NAME_LENGTH));
  TEXT_IO.PUT_LINE("Have a nice day!!");

end SAY_HI;
```

Student Notes:

SOFTWARE COMPONENTS

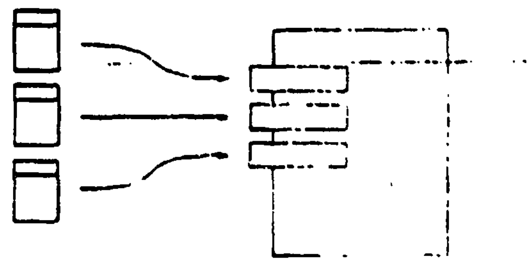
F-52

- Logically and physically self-contained software resources
- Similar in benefit to hardware components
- Provide a convenient mechanism for implementing a reusable program

PACKAGES

- Program units that allow us to collect logically related entities in one physical place
- Allow the definition of reusable software components/resources
- A fundamental feature of Ada which allow a change of mind-set
- An architecture-oriented feature

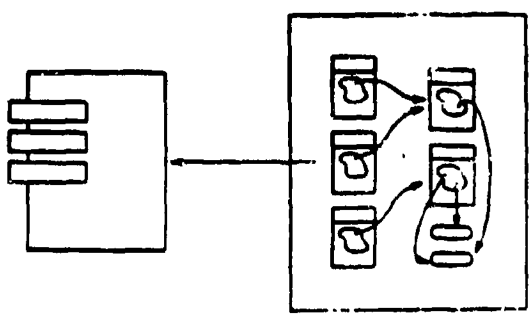
F-53



PACKAGES

- Place a "wall" around resources
- Export resources to users of a package
- May contain local resources hidden from the user of a package

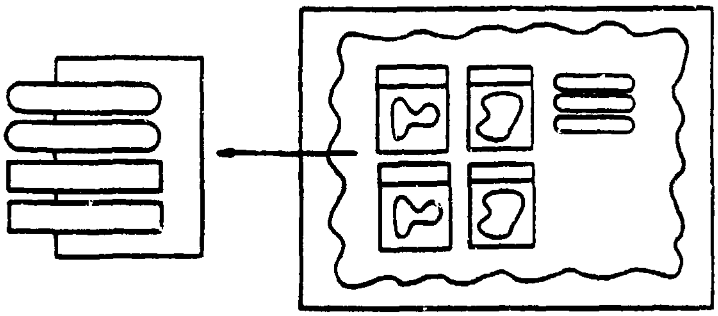
F-54



Student Notes:

STRUCTURE

SPECIFICATION **BODY**



F-55

- Defines resources available to user of the package
- Visible to user

- Defines implementation of resources
- Contains local resources
- Hidden from user

```
package CONSTANTS is

   PI : constant := 3.14159
   e  : constant := 2.71828;

end CONSTANTS;
```

F-56

```
with CONSTANTS;
procedure SOME_PROGRAM is

  MY_VALUE : FLOAT := 2 * CONSTANTS.PI;

begin

  null;

end SOME_PROGRAM;

with CONSTANTS;
procedure ANOTHER_PROGRAM is

  ANOTHER_VALUE : FLOAT := 2 * CONSTANTS.PI;

begin

  null;

end ANOTHER_PROGRAM;
```

F-57

Student Notes:

```
package ROBOT_CONTROL is
  type SPEED is range 0 .. 100;
  type DISTANCE is range 0 .. 500;
  type DEGREES is range 0 .. 359;
  procedure GO_FORWARD
    (HOW_FAST : in SPEED;
     HOW_FAR : in DISTANCE);
  procedure REVERSE
    (HOW_FAST : in SPEED;
     HOW_FAR  : in DISTANCE);
  procedure TURN (HOW_MUCH : in DEGREES);
end ROBOT_CONTROL;


with ROBOT_CONTROL;
procedure DO_A_SQUARE is
begin
  ROBOT_CONTROL.GO_FORWARD(HOW_FAST => 100,
                           HOW_FAR  => 20);
  ROBOT_CONTROL.TURN(90);
  ROBOT_CONTROL.GO_FORWARD(100, 20);
  ROBOT_CONTROL.TURN(90);
  ROBOT_CONTROL.GO_FORWARD(100, 20);
  ROBOT_CONTROL.TURN(90);
  ROBOT_CONTROL.GO_FORWARD(100, 20);
  ROBOT_CONTROL.TURN(90);
end DO_A_SQUARE;

package body ROBOT_CONTROL is
 procedure CLEAR_PORT is
 begin
   ...
 end CLEAR_PORT;
 procedure GO_FORWARD
   (HOW_FAST : in SPEED;
    HOW_FAR  : in DISTANCE) is
 begin
   ...
 end GO_FORWARD;
 procedure REVERSE (HOW_FAST : in SPEED;
                    HOW_FAR  : in DISTANCE) is
 begin
   ...
 end REVERSE;
 procedure TURN (HOW_MUCH : in DEGREES) is
 begin
   ...
 end TURN;
end ROBOT_CONTROL;
```

Student Notes:

```
package NUMBERS is

  MAX_NUM : constant := 40;
  type NUMBER_TYPE is range 0..MAX_NUM;
  procedure INCREMENT
    (A_NUMBER : in out NUMBER_TYPE);

end NUMBERS;
```

F-61

```
with NUMBERS;
procedure ADD_NUMBERS is

  NUMBER_1, NUMBER_2, NUMBER_3,
  TOTAL : NUMBERS.NUMBER_TYPE := 0;
  use NUMBERS;

begin

  NUMBERS.INCREMENT (NUMBER_1);
  NUMBER_2 := NUMBER_1;
  NUMBERS.INCREMENT (NUMBER_2);
  NUMBER_3 := NUMBER_2;
  NUMBERS.INCREMENT (NUMBER_3);
  TOTAL := NUMBER_1 + NUMBER_2 + NUMBER_3;

end ADD_NUMBERS;
```

F-62

SOFTWARE REUSABILITY

- Studies show that between 50% and 75% of code within a system is duplicated
- Treats software systems as a collection of potentially reusable components
- Must be a goal throughout the life cycle

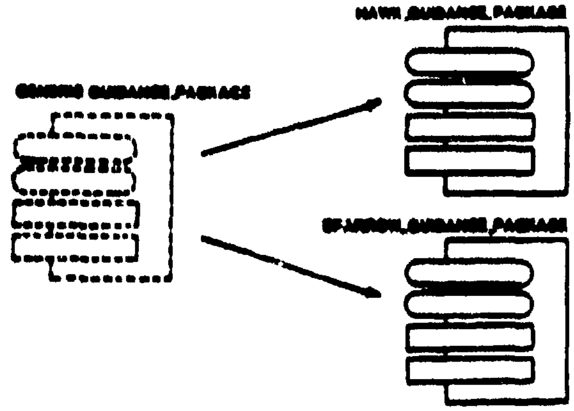
F-

Student Notes:

F-64

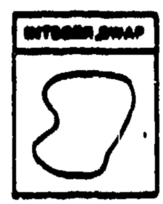
GENERICS

- Template for a subprogram or package
- Tailorable (parameterized)



F-65

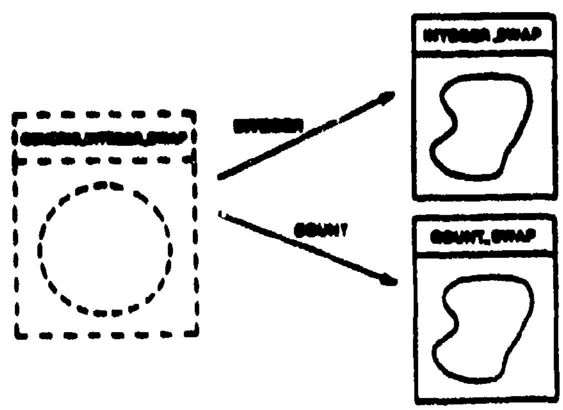
```
procedure INTEGER_SWAP (LEFT, RIGHT : in out INTEGER) is
  TEMP : INTEGER := LEFT;
begin
  LEFT := RIGHT;
  RIGHT := TEMP;
end INTEGER_SWAP;
```



F-66

INSTANTIATION

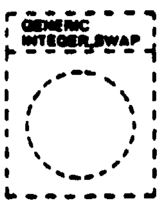
- An actual instance of a subprogram or package from a generic subprogram or package



Student Notes:

```
with GENERIC_INTEGER_SWAP;
procedure SWAP_VALUES is
  MAX_COUNT : constant := 200;
  type COUNT is range 0..MAX_COUNT;
  procedure INTEGER_SWAP
    is new GENERIC_INTEGER_SWAP (INTEGER);
  procedure COUNT_SWAP
    is new GENERIC_INTEGER_SWAP (COUNT);
  INTEGER_1 : INTEGER := 10;
  INTEGER_2 : INTEGER := 20;
  COUNT_1 : COUNT := 100;
  COUNT_2 : COUNT := 50;
begin
  INTEGER_SWAP (INTEGER_1, INTEGER_2);
  COUNT_SWAP (COUNT_1, COUNT_2);
end SWAP_VALUES;
```

F-67




```
generic
  type ANY_INTEGER_TYPE is range <>;
procedure GENERIC_INTEGER_SWAP (LEFT,
  RIGHT: in out ANY_INTEGER_TYPE);
procedure GENERIC_INTEGER_SWAP (LEFT, RIGHT: in out
  ANY_INTEGER_TYPE) is
  TEMP: ANY_INTEGER_TYPE := LEFT;
begin
  LEFT := RIGHT;
  RIGHT := TEMP;
end GENERIC_INTEGER_SWAP;
```

F-68

```
generic
  type ELEMENT_TYPE is private;
procedure GENERIC_SWAP
  (LEFT, RIGHT : in out ELEMENT_TYPE);
procedure GENERIC_SWAP
  (LEFT, RIGHT : in out ELEMENT_TYPE) is
  TEMP : ELEMENT_TYPE := LEFT;
begin
  LEFT  := RIGHT;
  RIGHT := TEMP;
end GENERIC_SWAP;
```

F-69

Student Notes:

F-70

```
with GENERIC_SWAP;

procedure SWAP_THINGS is

   MAX_COUNT : constant := 100;

   type COUNT
      is range - MAX_COUNT .. MAX_COUNT;

   type COLORS is (RED, BLUE, GREEN);

   type REAL is digits 10;

   procedure SWAP_COUNT
      is new GENERIC_SWAP (COUNT);
   procedure SWAP_COLORS
      is new GENERIC_SWAP (COLORS);
   procedure SWAP_REAL
      is new GENERIC_SWAP (REAL);

   COUNT_1 : COUNT := 5;
   COUNT_2 : COUNT := 10;

   COLOR_1 : COLORS := RED;
   COLOR_2 : COLORS := BLUE;

   REAL_1 : REAL := 20.0;
   REAL_2 : REAL := 40.0;

begin

   SWAP_COUNT (COUNT_1, COUNT_2);
   SWAP_COLORS (COLOR_1, COLOR_2);
   SWAP_REAL (REAL_1, REAL_2);

end SWAP_THINGS;
```

Student Notes:

Ada PROGRAM LIBRARY

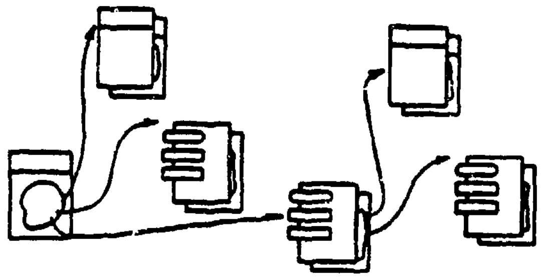
- A record of all the separately compiled program units that make up a program
- Central facility for the development of Ada systems

F-71

SEPARATE COMPILATION

- Program units may be separately compiled
- Separate compilation is possible because of the separation of specification and body
- A system is put together by referencing the specifications of other program units

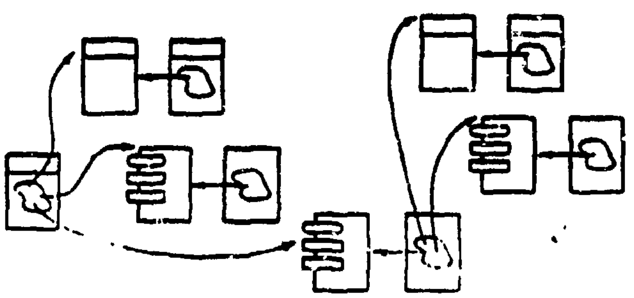
F-72



SEPARATE COMPILATION

- A program unit's specification may be compiled separately from its body
- Realizes not only a logical distinction between architecture and implementation, but also a physical distinction

F-73



Student Notes:

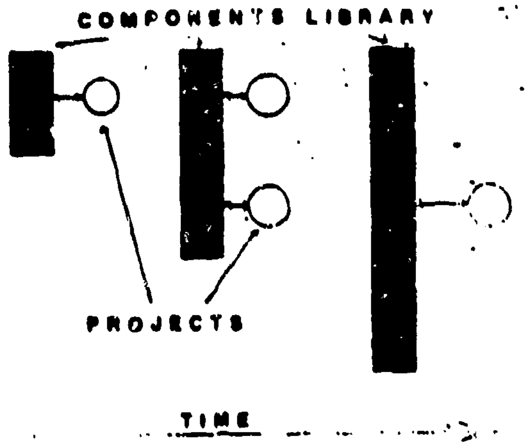
SEPARATE COMPILATION

F-74

- Allows development of independent software components
- Currently we all but lose the human effort going into software: it is disposable
- Separate compilation allows us to reuse components and keep our investment

SOFTWARE COMPONENTS

F-75



INDEPENDENT COMPILATION

F-76

- Widely used
- Modules have no way of sharing knowledge of properties defined in other modules
- Uses lower level of compile-time checking of consistency between units than is possible within a single compilation unit

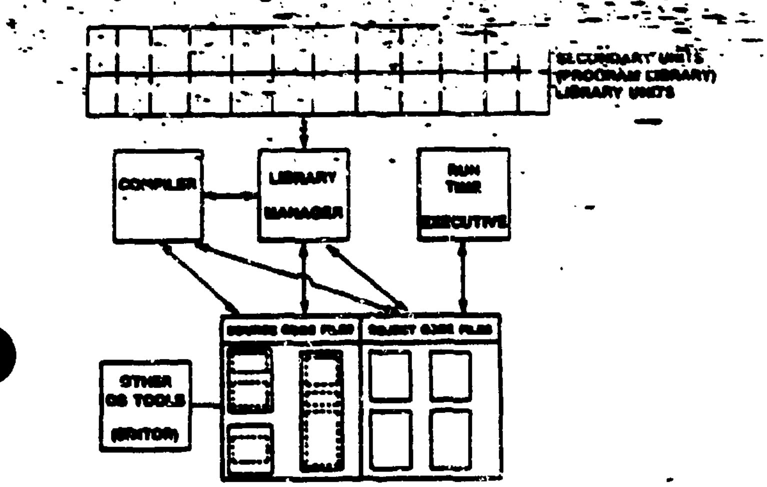
Student Notes:

SEPARATE COMPILATION

F-77

- Uses the program library to perform the same level of checking between units whether compiled in one compilation unit or many
- Resolves safety with reasons for compiling in parts

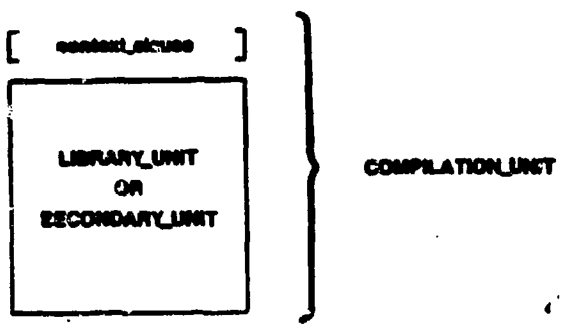
Ada COMPILATION MODEL



F-78

COMPILATION UNIT

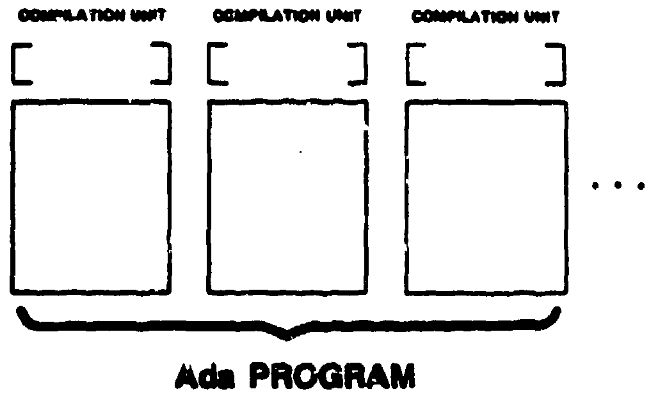
- A complete Ada program is a collection of compilation units submitted to the compiler separately



F-79

Student Notes:

F-80

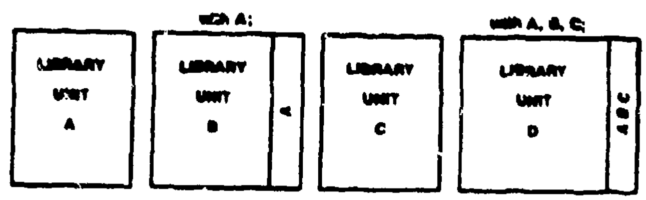


F-81

CONTEXT CLAUSE

- **Specifies previously compiled library units needed in this compilation unit**

with LIBRARY_UNIT_NAME;



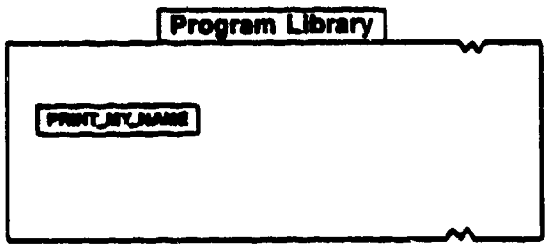
F-82

LIBRARY UNITS

- **Subprogram declaration (specification)**
- **Package declaration (specification)**
- **Generic declaration (specification)**
- **Generic instantiation**
- **Subprogram body (specification and body)**

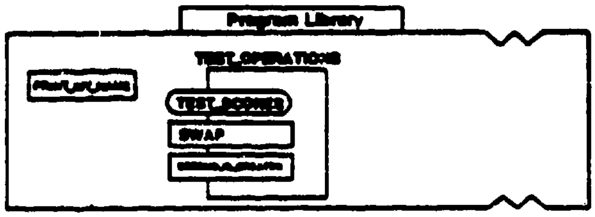
```
procedure PRINT_MY_NAME;
```

F-83



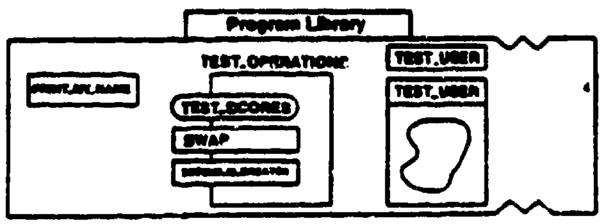

```
package TEST_OPERATIONS is
  type TEST_SCORES is range 0..100;
  procedure SWAP (FIRST, SECOND: in out TEST_SCORES);
  function SECOND_IS_GREATER (FIRST, SECOND: in TEST_SCORES)
           return BOOLEAN;
end TEST_OPERATIONS;
```

F-84




```
with PRINT_MY_NAME, TEST_OPERATIONS;
procedure TEST_USER is
  MY_TEST : TEST_OPERATIONS. TEST_SCORES := 10;
  YOUR_TEST : TEST_OPERATIONS. TEST_SCORES := 5;
begin

  PRINT_MY_NAME;
  TEST_OPERATIONS. SWAP (MY_TEST, YOUR_TEST);
end TEST_USER;
```

F-85



Student Notes:

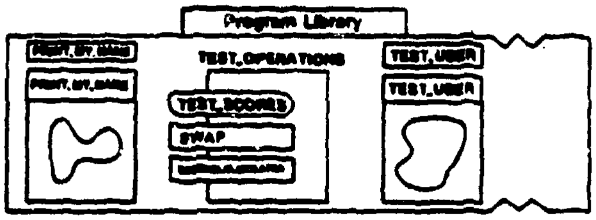
SECONDARY UNITS

- Library unit body
  - Subprogram body
  - Package body
- Subunit

F-86

```
procedure PRINT_MY_NAME is
begin -- PRINT_MY_NAME
  .
  .
  .
end PRINT_MY_NAME;
```

F-87

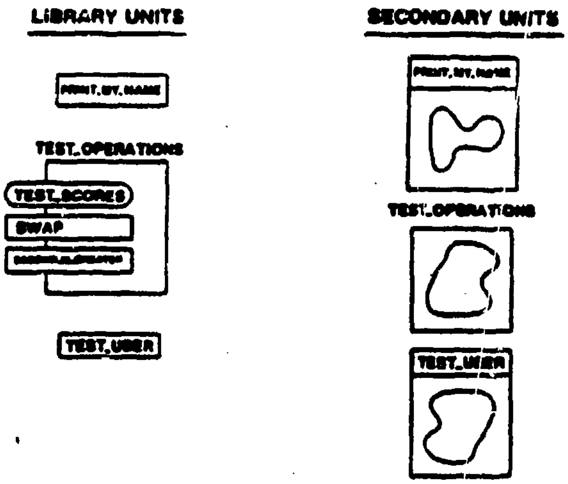


F-88

```
package body TEST_OPERATIONS is

  procedure SWAP
    (FIRST,SECOND : in out TEST_SCORES) is
    TEMP   : TEST_SCORES;
  begin -- SWAP
    TEMP   := FIRST;
    FIRST  := SECOND;
    SECOND := TEMP;
  end SWAP;
  --
  function SECOND_IS_GREATER
    (FIRST, SECOND : in TEST_SCORES)
           return BOOLEAN is
  begin -- SECOND_IS_GREATER
    return SECOND>FIRST;
  end SECOND_IS_GREATER;
end TEST_OPERATIONS;
```

Student Notes:

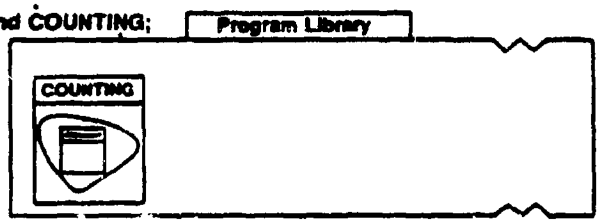
PROGRAM LIBRARY



F-89

SUBUNITS

```
procedure COUNTING is
  type SMALL_NUMBERS is range 1..10;
  --
  VALUE: SMALL_NUMBERS;
  --
  procedure INCREMENT (NUMBER: in out SMALL_NUMBERS)
            is separate;
begin -- COUNTING
  VALUE:= 1;
  INCREMENT (VALUE);
    :
    :
end COUNTING;
```

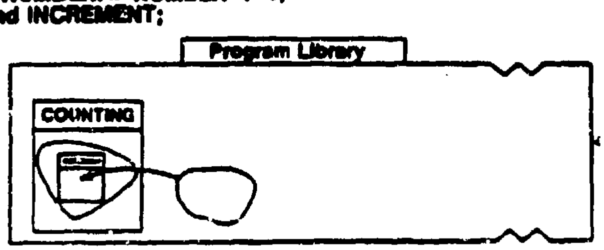


F-90

SUBUNITS

- Visibility rules for the subunit are the same as if the code was embedded as before

```
separate (COUNTING)
procedure INCREMENT (NUMBER: in out SMALL_NUMBERS) is

begin -- INCREMENT
  NUMBER:= NUMBER + 1;
end INCREMENT;
```



F-91

Student Notes:

CONTROL STRUCTURES

F-92

- Control flow of executable sequence of statements
- Define internal logic of a program unit

ASSIGNMENT STATEMENT

F-93

```
procedure CALCULATE_TOTALS is

  MAX_VALUE : constant := 1000;
  type VALUES is range 0..MAX_VALUE;

  VALUE_1, VALUE_2, VALUE_3 : VALUES := 10;
  VALUE_4, VALUE_5, VALUE_6 : VALUES := 0;

begin

  VALUE_4 := 20;
  VALUE_5 := VALUE_4 + 10;
  VALUE_6 := (VALUE_5 + 2) * VALUE_1;
  VALUE_6 := VALUE_6 + VALUE_1 + VALUE_2;

end CALCULATE_TOTALS;
```

F-94

```
package NUMBERS is

  MAX_NUM : constant := 40;
  type NUMBER_TYPE is range 0..MAX_NUM;
  procedure INCREMENT
     (A_NUMBER : in out NUMBER_TYPE);

end NUMBERS;
```

Student Notes:

PROCEDURE CALL

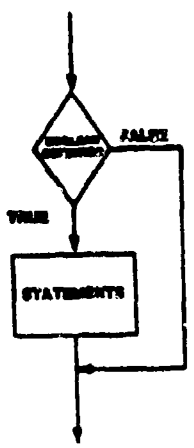
```
with NUMBERS;

procedure TOTAL_VALUES is

  MY_VALUE : NUMBERS.NUMBER_TYPE := 0;
  YOUR_VALUE : NUMBERS.NUMBER_TYPE := 4;
  use NUMBERS;

begin

  MY_VALUE := MY_VALUE + 1;
  NUMBERS.INCREMENT (MY_VALUE);
  YOUR_VALUE := 10;
  NUMBERS.INCREMENT (YOUR

end TOTAL_VALUES;
```

F-95

```
package body NUMBERS is

  procedure INCREMENT
   (A_NUMBER : in out NUMBER_TYPE) is

  begin

    A_NUMBER := A_NUMBER + 1;

  end INCREMENT;

end NUMBERS;
```

F-96

IF STATEMENT



if CONDITION then
STATEMENT;
STATEMENT;
STATEMENT;
end if;

F-97

Student Notes:

F-98

```
with NUMBERS;
procedure COUNT_UP is
 MY_NUMBER : NUMBERS.NUMBER_TYPE := 10;
 YOUR_NUMBER : NUMBERS.NUMBER_TYPE := 0;
 use NUMBERS;
begin
 NUMBERS.INCREMENT (MY_NUMBER);
 if MY_NUMBER = 11 then
   YOUR_NUMBER := 5;
 end if;
 NUMBERS.INCREMENT (YOUR_NUMBER);
 if YOUR_NUMBER = 6 then
   NUMBERS.INCREMENT (YOUR_NUMBER);
   NUMBERS.INCREMENT (MY_NUMBER);
 end if;
end COUNT_UP;
```

IF STATEMENT

F-99

```
if THE_SKY_IS_BLUE then
  THERE_ARE_NO_CLOUDS;
else
  THERE_ARE_CLOUDS;
  THE_SKY_IS_NOT_BLUE;
end if;
```

```
if THE_SKY_IS_BLUE then
  THERE_ARE_NO_CLOUDS;
elsif THE_SKY_IS_RED then
  IT_IS_MORNING;
  IT_IS_EVENING;
elsif THE_SKY_IS_GREEN then
  WE_HAVE_PROBLEMS;
else
  WHO_CARES;
end if;
```

LOOP STATEMENT

F 100

```
with NUMBERS;
procedure COUNT_UP is
 MY_NUMBER : NUMBERS.NUMBER_TYPE := 0;
 use NUMBERS;
begin
 loop
  NUMBERS.INCREMENT (MY_NUMBER);
  exit when MY_NUMBER >= NUMBERS.MAX_NUM;
 end loop;
end COUNT_UP;
```

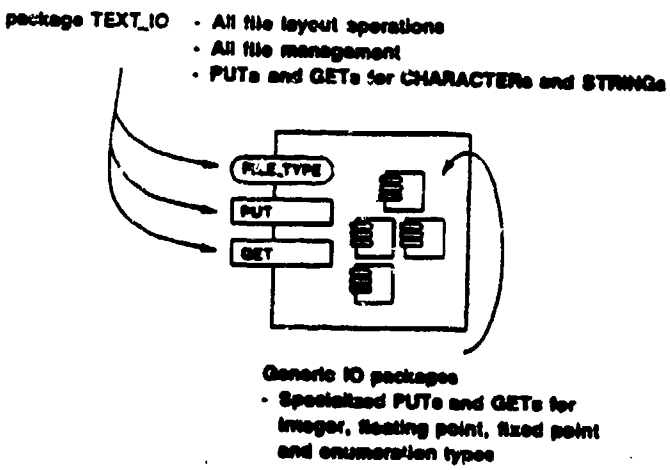
Student Notes:

INPUT/OUTPUT

TEXT_IO

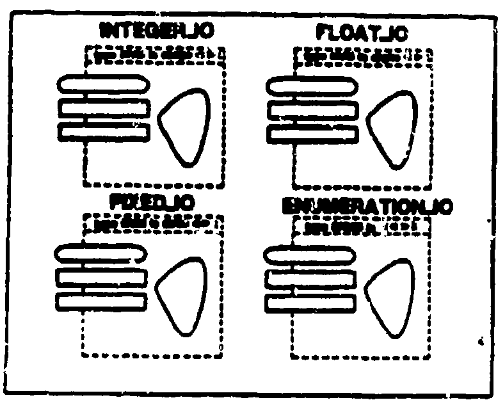
- A predefined package that provides input and output facilities for textual (human readable) objects F-101
- Contains I/O facilities for strings and characters and generic facilities for integers, enumerated, fixed and floating point types

TEXT_IO



F-102

GENERIC TEXT_IO



F-103

Student Notes:

STRING I/O

F-104

```
with TEXT_IO;

procedure OUTPUT_TEXT is
  MAX_LENGTH : constant   := 20;
  subtype LINE_TYPE is STRING(1..MAX_LENGTH);
  MY_LINE     : LINE_TYPE := (others => '#');

begin
  TEXT_IO.PUT("HI THERE,");
  TEXT_IO.PUT("    ");
  TEXT_IO.PUT("WELCOME TO Ada");
  TEXT_IO.NEW_LINE;
  TEXT_IO.PUT(MY_LINE);
  TEXT_IO.NEW_LINE;

end OUTPUT_TEXT;

HI THERE,    WELCOME TO Ada
####################
```

STRING I/O

F-105

```
with TEXT_IO;

procedure OUTPUT_TEXT is
  MAX_LENGTH : constant   := 20;
  subtype LINE_TYPE is STRING (1..MAX_LENGTH);
  MY_LINE     : LINE_TYPE := (others => '#');

begin
  TEXT_IO.PUT_LINE ("HI THERE.");
  TEXT_IO.PUT_LINE ("WELCOME TO Ada");
  TEXT_IO.PUT_LINE (MY_LINE);

end OUTPUT_TEXT;

HI THERE
WELCOME TO Ada
####################
```

STRING I/O

F-106

```
package LINE_PACKAGE is

  MAX_LENGTH : constant := 20;
  subtype LINE_TYPE is STRING (1..MAX_LENGTH);

end LINE_PACKAGE;
```

Student Notes:

STRING I/O

```
with TEXT_IO, LINE_PACKAGE;

procedure ECHO_NAME is

  NAME : LINE_PACKAGE.LINE_TYPE := (others =>' ');

begin

  TEXT_IO.PUT ("WHAT IS YOUR NAME?");
  TEXT_IO.GET (NAME);
  TEXT_IO.PUT ("HI");
  TEXT_IO.PUT_LINE (NAME);

end ECHO_NAME;
```

F-107

CHARACTER_IO

```
with TEXT_IO, LINE_PACKAGE;

procedure ECHO_NAMES is

  NAME : LINE_PACKAGE.LINE_TYPE := (others => ' ');
  ANSWER  :  CHARACTER  := 'N';

begin

    TEXT_IO.PUT ("WHAT IS YOUR NAME?");
    TEXT_IO.GET (NAME);
    TEXT_IO.PUT ("HI");
    TEXT_IO.PUT_LINE (NAME);
    TEXT_IO.PUT ("MORE NAMES? (Y TO CONTINUE):");
    TEXT_IO.GET (ANSWER);
    TEXT_IO.SKIP_LINE;
    exit when ANSWER /= 'Y' or ANSWER /= 'y';

  end loop;

end ECHO_NAMES;
```

F-108

Student Notes:

GENERIC I/O

F-109

```
with TEXT_IO, LINE_PACKAGE, NUMBERS;

procedure ECHO_AGE is

  package NUMBER_IO is new TEXT_IO.INTEGER_IO
    (NUMBERS.NUMBER_TYPE);
  NAME : LINE_PACKAGE.LINE_TYPE :=
    (others => ' ');
  AGE  : NUMBERS.NUMBER_TYPE := 0;

begin

  TEXT_IO.PUT ("WHAT IS YOUR NAME: ");
  TEXT_IO.GET (NAME);
  TEXT_IO.PUT ("HOW OLD ARE YOU? ");
  NUMBER_IO.GET (AGE);

  TEXT_IO.PUT (NAME);
  TEXT_IO.PUT (", YOU ARE ");
  NUMBER_IO.PUT (AGE);
  TEXT_IO.PUT_LINE (" YEARS OLD");

end ECHO_AGE;
```

Student Notes:

BASIC ADA TYPES

- Purpose of Typing
- Type Declarations
- Object Declarations
- Classes of Basic Ada Types

B-1

TYPING

- A type defines a set of values and a set of operations applicable to those values for objects of that type

B-2

PURPOSE OF TYPING

- To impose structure on data for:
  - Factorization of Properties, Maintainability
  - Reliability
  - Abstraction, Hiding of Implementation Details

B-3

Student Notes:

B-4

STRONG TYPING

- Every object must have a specified type that is static
- Cannot mix objects of different types without explicit conversion

B-5

TYPE DECLARATIONS

- Construct used to define a new type
- Creates a new type name which is distinct from other type names
- Form
  type TYPE_NAME is [CLASS OF TYPE];

B-6

TYPE DECLARATIONS

```
type COUNT is range 0 .. 500; -- integer type
type SCALE is ( LOW, MEDIUM, HIGH );  -- enumerated type
type WEIGHT is digits 10 range 0.0 .. 1000.0; -- floating point type
type CURRENT is delta 0.0625 range 0.0 .. 100.0; -- fixed point type

type CHARACTER_COUNT is array ( CHARACTER ) of COUNT; -- array type
type CLASSIFY is record   -- record type
   VALUE : WEIGHT;
   CATEGORY: SCALE;
end record;
```

Student Notes:

OBJECT DECLARATIONS

B-7

- An instance of a type
- Reserves storage with the structure defined by the type
- Form

OBJECT_NAME: TYPE_NAME:= INITIAL_VALUE;

OBJECT DECLARATIONS

B-8

TOTAL_COUNT	:	COUNT	:=	0;
RATING	:	SCALE	:=	LOW;
SMALLEST_WEIGHT	:	WEIGHT	:=	0.0;
LINE_CURRENT	:	CURRENT	:=	0.0;
HOW_MANY	:	CHARACTER_COUNT	:=	(others => 0);
VALUE_CLASSIFICATION	:	CLASSIFY	:=	(0.0, Low);

FORMS

B-9

VARIABLE

TOTAL_COUNT	:	COUNT	:=	0;

CONSTANT

SMALLEST_WEIGHT	:	constant WEIGHT	:=	0.0;

NAMED NUMBER

MAXIMUM_COUNT	:	constant	:=	100;

Student Notes:

```
with TEXT_IO;
procedure TOTAL_NUMBERS is

  NUMBER_TO_GET : constant := 5;

  MAXIMUM_NUMBERS : constant := 10;
  type NUMBERS is range 0 .. MAXIMUM_NUMBERS * NUMBER_TO_GET;
  subtype INPUT_NUMBERS is NUMBERS range 0 .. MAXIMUM_NUMBERS;

  A-NUMBER : INPUT_NUMBERS := 0;
  TOTAL    : NUMBERS       := 0;

  package NUMBER_IO is new TEXT_IO.INTEGER_IO( NUMBERS );

begin

  for TOTAL_LOOP in 1 .. NUMBER_TO_GET loop
    TEXT_IO.PUT("Number -) ");
    NUMBER_IO.GET( A_NUMBER );
    TOTAL := TOTAL + 1_NUMBER;
  end loop;
  TEXT_IO.PUT("Total of numbers is ");
  NUMBER_IO.PUT( TOTAL );

end TOTAL_NUMBERS;
```

B-10

ADA TYPES

B-11

- SCALAR — single values
  - DISCRETE — exact values
    - INTEGER
    - ENUMERATED
  - REAL — approximate values
    - FIXED Point — absolute
    - FLOATING Point — relative
- COMPOSITE — multiple values
  - ARRAY — homogeneous (components have same type)
  - RECORD — heterogeneous (components may have different types)
- ACCESS — dynamic variables
- PRIVATE/LIMITED — abstract data types
- TASK — designate tasks

SCALAR TYPES

B-12

- Objects contain a single value
- Values are ordered

Student Notes:

DISCRETE INTEGER

- FORM

  type IDENTIFIER is range LOWER_BOUND .. UPPER_BOUND;

- EXAMPLE

```
MIN_AGE : constant := 0;
MAX_AGE : constant := 150;

type AGE_TYPE is range MIN_AGE .. MAX_AGE;
```

B-13

SET OF VALUES:
(0, 1, 2, ...150)

```
with TEXT_IO;
procedure AVERAGE_NUMBERS is

NUMBER_TO_GET : constant := 5;

  MAXIMUM_NUMBERS : constant := 10;
  type NUMBERS is range 0 .. MAXIMUM_NUMBERS * NUMBER_TO_GET;
  subtype INPUT_NUMBERS is NUMBERS range 0 .. MAXIMUM_NUMBERS;

  type NUMBER_COUNT is range 0 .. NUMBER_TO_GET;

  A_NUMBER              : INPUT_NUMBERS     := 0;
  TOTAL                 : NUMBERS           := 0;
  HOW_MANY_NUMBERS      : NUMBER_COUNT      := 0;

  package NUMBER_IO is new TEXT_IO.INTEGER_IO( NUMBERS );
  package COUNT_IO  is new TEXT_IO.INTEGER_IO( NUMBER_COUNT );

begin

  TEXT_IO.PUT("How many numbers do you have ->" );
  COUNT_IO.GET ( HOW_MANY_NUMBERS );
  for TOTAL_LOOP in 1 .. HOW_MANY_NUMBERS loop
    TEXT_IO.PUT("Number -> ");
    NUMBER_IO.GET( A_NUMBER );
    TOTAL := TOTAL + A_NUMBER;
  end loop;
  TEXT_IO.PUT("Total of numbers is ");
  NUMBER_IO.PUT( TOTAL );
  TEXT_IO.NEW_LINE(2);
  TEXT_IO.PUT("The average of the numbers is");
  NUMBER_IO.PUT( TOTAL / NUMBERS(HOW_MANY_NUMBERS) );

end AVERAGE_NUMBERS;
```

B-14

Student Notes:

DISCRETE ENUMERATED

B-15

- Enable direct representation of non-integer values
- Example, security classes

UNCLASSIFIED, CONFIDENTIAL, SECRET, TOP_SECRET

```
type SECURITY_TYPE is (UNCLASSIFIED, CONFIDENTIAL,
SECRET, TOP_SECRET);
```

B-16

```
procedure CONTROL_ACCESS is
  type SECURITY_TYPE is ( UNCLASSIFIED, CONFIDENTIAL, SECRET,
  TOP_SECRET );

  procedure GET_CLASS( SECURITY_LEVEL : out SECURITY_TYPE) is
  separate;
  procedure ENABLE_CONFIDENTIAL_ACCESS is separate;
  procedure ENABLE_SECRET_ACCESS is separate;
  procedure ENABLE_TOP_SECRET_ACCESS is separate;

  SECURITY_CLASS : SECURITY_TYPE := SECURITY_TYPE'FIRST;

begin

  GET_CLASS( SECURITY_CLASS );

  if SECURITY_CLASS = TOP_SECRET then

    ENABLE_TOP_SECRET_ACCESS;
    ENABLE_SECRET_ACCESS;
    ENABLE_CONFIDENTIAL_ACCESS;

  elsif SECURITY_CLASS = SECRET then

    ENABLE_SECRET_ACCESS;
    ENABLE_CONFIDENTIAL_ACCESS;

  elsif SECURITY_CLASS = CONFIDENTIAL then

    ENABLE_CONFIDENTIAL_ACCESS;

  end if;

end CONTROL_ACCESS;
```

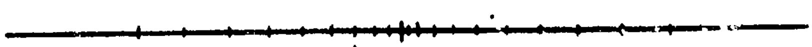
Student Notes:

REAL

- Provide approximations for real numbers B-17
- Two ways of handling error bounds
  - Floating Point — relative error bound
  - Fixed Point — absolute error bound
- Model Numbers give implementation-independent accuracy
- Safe Numbers give implementation-dependent accuracy

FLOATING POINT TYPES

- Error bound between numbers is expressed as relative to the position of the number over the entire range of values B-18
- Accuracy is specified in terms of the number of significant digits required



FLOATING POINT TYPES

- Form B-19

  type TYPE_NAME is digits 10 [range 0.0 .. 100.0];

  type REAL is digits 15 range -100.0 .. 100.0;

Student Notes:

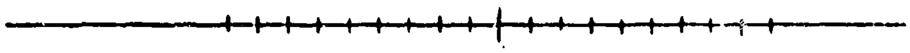
```
with TEXT_IO;
procedure AVERAGE_NUMBERS is

  NUMBER_TO_GET : constant := 5;

  MAXIMUM_NUMBERS : constant := 10.0;
  type NUMBERS is digits 10 range 0.0 ..
    MAXIMUM_NUMBERS * NUMBER_TO_GET;

  subtype INPUT_NUMBERS is NUMBERS range 0.0 ..
    MAXIMUM_NUMBERS;

  type NUMBER_COUNT is range 0 .. NUMBER_TO_GET;

    A_NUMBER          : INPUT_NUMBERS  := 0.0;
    TOTAL             : NUMBERS        := 0.0;
    HOW_MANY_NUMBERS  : NUMBER_COUNT   := 0;

  package NUMBER_IO is new TEXT_IO.FLOAT_IO( NUMBERS );
  package COUNT_IO is new TEXT_IO.INTEGER_IO( NUMBER_COUNT );

begin

  TEXT_IO.PUT("How many numbers do you have -> ");
  COUNT_IO.GET ( HOW_MANY_NUMBERS );
  for TOTAL_LOOP in 1 .. HOW_MANY_NUMBERS loop
    TEXT_IO.PUT("Number -> ");
    NUMBER_IO.GET( A_NUMBER );
    TOTAL := TOTAL + A_NUMBER;
  end loop;
  TEXT_IO.PUT("Total of numbers is ");
  NUMBER_IO.PUT( TOTAL );
  TEXT_IO.NEW_LINE(2);
  TEXT_IO.PUT("The average of the numbers is");
  NUMBER_IO.PUT( TOTAL / NUMBERS(HOW_MANY_NUMBERS) );

end AVERAGE_NUMBERS;
```

B-20

FIXED POINT TYPES

B-21

- Error bound between numbers is expressed as a fixed value between any two numbers
- Accuracy is specified in terms of the delta (change) required



FIXED POINT TYPES

Student Notes:

- Form

type TYPE_NAME is delta 1.0/8 range 0.0 .. 1000.0; B-22

type FIXED_TYPE is delta 1.0/16 range 0.0 .. 1000.0;

```
with TEXT_IO;
procedure AVERAGE_NUMBERS is

  NUMBER_TO_GET : constant := 5;

  MAXIMUM_NUMBERS : constant := 10.0;

  EIGHTS : constant := 1.0/8;                                  B-23

  type NUMBERS is delta EIGHTHS range 0.0 ..
    MAXIMUM NUMBERS * NUMBER_TO_GET;

  subtype INPUT_NUMBERS is NUMBERS range 0.0 ..
    MAXIMUM_NUMBERS;

  type NUMBER_COUNT is range 0 .. NUMBER_TO_GET;

    A_NUMBER           : INPUT_NUMBERS  := 0.0;
    TOTAL              : NUMBERS        := 0.0;
    HOW_MANY_NUMBERS   : NUMBER_COUNT   := 0;

  package NUMBER_IO is
    new TEXT_IO.FIXED_IO( NUMBERS );
  package COUNT_IO is
    new TEXT_IO.INTEGER_IO( NUMBER_COUNT );

begin

  TEXT_IO.PUT("How many numbers do you have -> ");
  COUNT_IO.GET ( HOW_MANY_NUMBERS );
  for TOTAL_LOOP in 1 .. HOW_MANY_NUMBERS loop
    TEXT_IO.PUT("Number -> ");
    NUMBER_IO.GET( A_NUMBER );
    TOTAL := TOTAL + A_NUMBER;
  end loop;
  TEXT_IO.PUT("Total of numbers is ");
  NUMBER_IO.PUT( TOTAL );
  TEXT_IO.NEW_LINE(2);
  TEXT_IO.PUT("The average of the numbers is");
  NUMBER_IO.PUT
    ( NUMBERS(TOTAL / NUMBERS(HOW_MANY_NUMBERS)) );

end AVERAGE_NUMBERS;
```

Student Notes:

COMPOSITE TYPES

B-24
- Objects may contain multiple values
- Two kinds
  - Arrays – values have same type
  - Records – values may have different types

ARRAYS

- In declaration must specify
  - The type of the components
  - The type of the index

B-25
- Form

```
NUMBER_OF_TILES : constant := 7;
type TILE_NUMBER is range 1 .. NUMBER_OF_TILES;
type LETTER is (A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,
  P,Q,R,S,T,U,V,W,W,X,Y,Z,BLANK);
type RACK is array (TILE_NUMBER) of LETTER;

MY_RACK : RACK := (A,J,B,BLANK,K,S,S);
```

1	2	3	4	5	6	7
A	J	B	Blank	K	S	S

ARRAY INDEXING

- To reference a particular component of an array must specify index

B-26

```
MY_RACK (3)       := BLANK;

MY_RACK (6 .. 7)  := (A,B);

MY_RACK (4)       := MY_RACK (6);
```

Student Notes:

```
package SAMPLER is

  BANDWIDTH : constant := 100;
  type FREQUENCIES is range -BANDWIDTH .. BANDWIDTH;

  MAX_MAGNITUDE : constant := 10;
  type MAGNITUDE is range 0 .. MAX_MAGNITUDE;

  type SPECTRUM is array( FREQUENCIES ) of MAGNITUDE;

  function HIGH_FREQUENCY
    (A_SPECTRUM : SPECTRUM ) return FREQUENCIES;

end SAMPLER;
```

B-27

```
package body SAMPLER is

  function HIGH_FREQUENCY ( A_SPECTRUM : SPECTRUM ) return FREQUENCIES is

    HIGH_MAGNITUDE : MAGNITUDE := MAGNITUDE'FIRST;
    HIGHEST_FREQUENCY : FREQUENCIES := A_SPECTRUM'FIRST;

  begin

    for FREQUENCY in A_SPECTRUM'RANGE loop

      if HIGH_MAGNITUDE < A_SPECTRUM(FREQUENCY) then
        HIGH_MAGNITUDE := A_SPECTRUM(FREQUENCY);
        HIGHEST_FREQUENCY := FREQUENCY;
      end if;
    end loop;

    return HIGHEST_FREQUENCY;

  end HIGH_FREQUENCY;

end SAMPLER;
```

B-28

UNCONSTRAINED ARRAYS

- Give the ability to declare varying sized objects from the same array type declaration.

```
type STRING is array (POSITIVE range <>) of CHARACTER;

MAX_TEXT_LINE : constant := 80;
subtype TEXT is STRING (1 .. MAX_TEXT_LINE);
SMALL_TEXT_SIZE : constant := 10;
subtype SHORT_TEXT is STRING (1 .. SMALL_TEXT_SIZE);
SHORT_LINE  : SHORT_TEXT;
LONG_LINE   : TEXT;
LINE        : STRING (1 .. 12);
```

B-29

Student Notes:

```
package SAMPLER is

   BANDWIDTH : constant := 100;
   SMALL : constant := 10;
   MEDIUM : constant := 50;
   type FREQUENCIES is range -BANDWIDTH .. BANDWIDTH;

   MAX_MAGNITUDE : constant := 10;
   type MAGNITUDE is range 0 .. MAX_MAGNITUDE;

   type SPECTRUM is array( FREQUENCIES range <> ) of MAGNITUDE;

   subtype SMALL_SPECTRUM is SPECTRUM(-SMALL..SMALL);
   subtype MEDIUM_SPECTRUM is SPECTRUM(-MEDIUM..MEDIUM);
   subtype FULL_SPECTRUM is SPECTRUM(-BANDWIDTH..BANDWIDTH);

   function HIGH_FREQUENCY
     ( A_SPECTRUM : SPECTRUM ) return FREQUENCIES;
end SAMPLER;
```

B-30

```
with SAMPLER;
procedure FIND_HIGHEST is

   SHORT_RANGE : SAMPLER.SMALL_SPECTRUM := (others => 5 );
   FULL_RANGE  : SAMPLER.FULL_SPECTRUM  := (others => 1);
   HIGHEST     : SAMPLER.FREQUENCIES := SAMPLER.FREQUENCIES'FIRST;
   SHORT_HIGH  : SAMPLER.FREQUENCIES := SAMPLER.FREQUENCIES'FIRST;
   FULL_HIGH   : SAMPLER.FREQUENCIES := SAMPLER.FREQUENCIES'FIRST;

begin
   SHORT_HIGH := SAMPLER.HIGH_FREQUENCY( SHORT_RANGE );
   FULL_HIGH := SAMPLER.HIGH_FREQUENCY( FULL_RANGE );
      if HIGHEST < SHORT_HIGH then
        HIGHEST := SHORT_HIGH;
   end if;
      if HIGHEST < FULL_HIGH then
        HIGHEST := FULL_HIGH;
   end if;
end FIND_HIGHEST;
```

B-31

MULTI DIMENSIONAL ARRAYS

```
package GAME_PIECES is

   NUMBER_OF_TILES : constant := 7;
   type TILE_NUMBER is range 1 .. NUMBER_OF_TILES;

   NUMBER_OF_SQUARES : constant := 15;
   type SQUARES is range 1 .. NUMBER_OF_SQUARES;
   type TILES is (A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T,
                                 U, V, W, X, Y, Z, BLANK, EMPTY )
   type RACK is array ( TILE_NUMBER ) of TILES;
   type BOARD is array ( SQUARES, SQUARES ) of TILES;
end GAME_PIECES;
```

B-32

```
package FORMAT is

  procedure ADD_DOLLAR_SIGN ( A_STRING : in out STRING );
end FORMAT;
```

B-33

```
package body FORMAT is

  procedure ADD_DOLLAR_SIGN ( A_STRING : in out STRING ) is

    DOLLAR : CHARACTER := '$';

  begin

    A_STRING( A_STRING'FIRST+1 .. A_STRING'LAST ) :=
      A_STRING( A_STRING'FIRST .. A_STRING'LAST-1 );
    A_STRING( A_STRING'FIRST ) := DOLLAR;

  end ADD_DOLLAR_SIGN;

end FORMAT;
```

B-34

```
with FORMAT, TEXT_IO;

procedure FORMAT_NUMBER is

  MAX_LENGTH : constant := 80;
  subtype NUMBER_STRING is STRING(1..MAX_LENGTH);
  A_NUMBER_STRING : NUMBER_STRING := ( others => ' ' );

  LENGTH : NATURAL := 0;

begin

  TEXT_IO.GET_LINE( A_NUMBER_STRING, LENGTH );
  FORMAT.ADD_DOLLAR_SIGN ( A_NUMBER_STRING(1..LENGTH+1) );
  TEXT_IO.PUT( A_NUMBER_STRING );

end FORMAT_NUMBER;
```

B-35

Student Notes:

RECORDS

- Components may have different types
- Form

B-36

```
NUMBER_OF_DAYS_IN_MONTH : constant := 31;
type DAY-TYPE is range 1 .. NUMBER_OF_DAYS_IN_MONTH;
type MONTH TYPE is ( JAN, FEB, MAR, APR, MAY, JUN, JUL,
                     AUG, SEP, OCT, NOV, DEC );
LAST_DAY_ON_EARTH : constant := 2085;
type YEAR_TYPE is range 1 .. LAST_DAY_ON_EARTH;

type DATE is record
  DAY   : DAY_TYPE;
  MONTH : MONTH_TYPE;
  YEAR  : YEAR_TYPE;
end record;
```

- Components are referenced using "dot notation"

B-37

TODAY

3	TODAY.DAY
JUN	TODAY.MONTH
1987	TODAY.YEAR

```
TODAY : DATE;
TOMORROW:DATE;
YESTERDAY : DATE;

begin

TODAY.DAY := 3;
TODAY.MONTH := JUN;
TODAY.YEAR := 1987;

YESTERDAY.DAY := TODAY.DAY -1;
YESTERDAY.MONTH := TODAY.MONTH;
YESTERDAY.YEAR := TODAY.YEAR;

TOMORROW := TODAY;

TOMORROW.DAY := TOMORROW.DAY + 1;

TODAY := ( 4, JUN, 1987 );
```

B-38

Student Notes:

OTHER RECORD FORMS

B-39

- Discriminated
- Variant

OTHER ADA TYPES

B-40

- Access Types
  - ●●Equivalent to dynamic variables in other languages
  - ●●Used to dynamically allocate/deallocate storage at run time
- Task Types -- Designate tasks
- Private Types -- Abstract data types

Student Notes:

CONTROL STRUCTURES

C-1

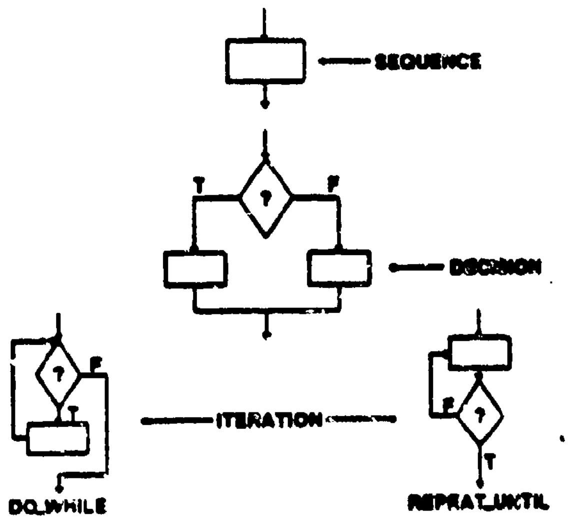
- Structured Programming
- Sequential
- Conditional
- Iterative

STRUCTURED PROGRAMMING

C-2

- A methodological style for constructing programs by connecting well understood constructs called control structures
- Three different control structures are sufficient for writing any logic (Bohm/Jacopini '64)
  - Sequence (executable statements
  - Decision clause (If then else)
  - Iteration construct (While or until)

C-3



Student Notes:

BENEFITS

- Understandability
- Modifiability
- Reliability

C-5

SEQUENTIAL STATEMENTS

- Assignment :=
- Null null;
- Block Statement

BLOCK STATEMENT

- Localizes declarations and/or effects
- Form

```
declare
-- local declarative part - OPTIONAL
begin
  -- statements
end;
```

C-6

```
with TEXT_IO;
procedure FILL_LIST is
  MAX_NUMBER : constant := 100;
  type NUMBERS is range 1 .. MAX_NUMBER;
  package NUMBER_IO is new TEXT_IO.INTEGER_IO( NUMBERS );

  LIST_SIZE : constant := 1000;
  type LIST_INDEX_TYPE is range 1 .. LIST_SIZE;
  package INDEX_IO is new TEXT_IO.INTEGER_IO( LIST_INDEX_TYPE );

  type LIST_TYPE is array( LIST_INDEX_TYPE range <> ) of NUMBERS;

  LOWER_BOUND,
  UPPER_BOUND : LIST_INDEX_TYPE := LIST_INDEX_TYPE'FIRST;

begin
  INDEX_IO.GET( LOWER_BOUND );
  INDEX_IO.GET( UPPER_BOUND );

  declare
    LIST_OF_NUMBERS : LIST_TYPE( LOWER_BOUND .. UPPER_BOUND );

  begin
    for LIST_ITEM in LIST_OF_NUMBERS'RANGE loop
      NUMBER_IO.GET( LIST_OF_NUMBERS(LIST_ITEM) );
    end loop;

    for LIST_ITEM in LIST_OF_NUMBERS'RANGE loop
      NUMBER_IO.PUT( LIST_OF_NUMBERS(LIST_ITEM) );
    end loop;

  end;  --block statement

end FILL_LIST;
```

C-7

C-8

```
with TEXT_IO;
procedure FILL_LIST is

  MAX_NUMBER : constant := 100;
  type NUMBERS is range 1 .. MAX_NUMBER;
  package NUMBER_IO is new TEXT_IO.INTEGER_IO( NUMBERS );

  LIST_SIZE : constant := 1000;
  type LIST_INDEX_TYPE is range 1 .. LIST_SIZE;
  package INDEX_IO is new TEXT_IO.INTEGER_IO( LIST_INDEX_TYPE );

  type LIST_TYPE is array( LIST_INDEX_TYPE range <> ) of NUMBERS;

  LOWER_BOUND,
  UPPER_BOUND : LIST_INDEX_TYPE := LIST_INDEX_TYPE'FIRST;

begin
 loop
  begin
   INDEX_IO.GET( LOWER_BOUND );
   INDEX_IO.GET( UPPER_BOUND );
   exit;

  exception
   when others => TEXT_IO.PUT_LINE("Illegal bounds, try again");
  end;

 end loop;

 declare
  LIST_OF_NUMBERS : LIST_TYPE( LOWER_BOUND .. UPPER_BOUND
 begin
  for LIST_ITEM in LIST_OF_NUMBERS'RANGE loop
   NUMBER_IO.GET( LIST_OF_NUMBERS(LIST_ITEM) );
  end loop;

  for LIST_ITEM in LIST_OF_NUMBERS'RANGE loop
   NUMBER_IO.PUT( LIST_OF_NUMBERS(LIST_ITEM) );
  end loop;

 end;   --block statement
 end FILL_LIST;
```

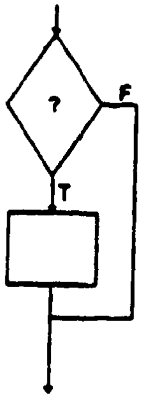
Student Notes:

CONDITIONAL

C-9

- Change control flow based on the value of an expression
- If statement
- Case statement

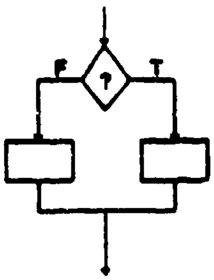
IF STATEMENT



C-10

```
If CONDITION then
  -- statements
end if;

If ITEM < LIST (CHECK) then
    TEMP := ITEM;
    ITEM := LIST (CHECK);
    LIST (CHECK) := TEMP;
end if;
```

IF - THEN - ELSE



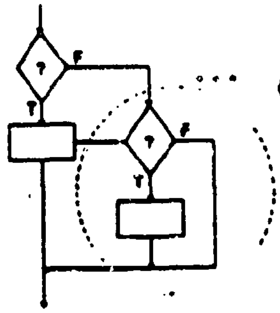
C-11

```
If CONDITION then
  -- statements
else
  -- statements
end if;
If ITEM < LIST (CHECK) then
    TEMP := ITEM;
    ITEM := LIST (CHECK);
    LIST (CHECK) := TEMP;
else
    CHECK := CHECK + 1;
    FOUND := FALSE;
end if;
```

Student Notes:

C-12

IF - THEN - ELSIF - THEN




```
if CONDITION then
  -- statements
{elsif  CONDITION then}
{  -- statements      }
end if;
if MESSAGE = LOW_PRIORTITY then
     SET_PRIORITY_FLAG (LOW);
     ROUTE_LOW_MESSAGE;
elsif MESSAGE = HIGH_PRIORITY then
     ROUTE_HIGH_MESSAGE;
     INCREMENT_HIGH_COUNT
end if;
```

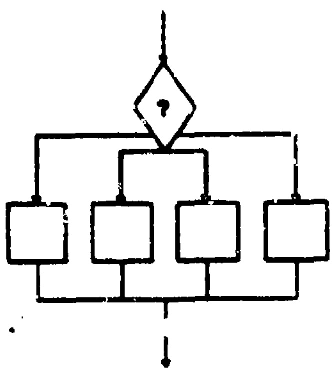
C-13

FULL IF STATEMENT

```
if CONDITION then
  -- statements

{elsif CONDITION then}
{  -- statements     }

[else            ]
[  -- statements ]

end if;
```

C-14

CASE STATEMENT




```
case DISCRETE_EXPRESSION is
  when VALUE_1 =>  -- statements
  when VALUE_2 =>  -- statements
     :
     :
end case;
```

- Alternative must be mutually exclusive and exhaustive

Student Notes:

CASE STATEMENT

```
GET_COLOR ( USER_COLOR );

case USER_COLOR is

  when RED => INCREMENT_COUNT ( PRIMARY_COLOR );
              NUMBER_RECEIVED := NUMBER_RECEIVED +1;

  when BLUE => INCREMENT_COUNT ( PRIMARY_COLOR );
               NUMBER_RECEIVED := NUMBER_RECEIVED +1;

  when YELLOW => INCREMENT_COUNT ( PRIMARY_COLOR );
                 NUMBER_RECEIVED := NUMBER_RECEIVED +1;

  when others => INCREMENT_COUNT ( SECONDARY_COLOR );

end case;
```

C-15

```
case USER_COLOR is

 when RED | BLUE | YELLOW =>
  INCREMENT_COUNT ( PRIMARY_COLOR );
  NUMBER_RECEIVED := NUMBER_RECEIVED +1;

 when others => INCREMENT_COUNT ( SECONDARY_COLOR );

end case;
```

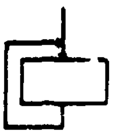
C-16

ITERATIVE STATEMENT

- Iteration is performed using the Ada loop which can be flavored for different kinds of looping

C-17

BASIC LOOP



```
loop
  -- statements
end loop;
```

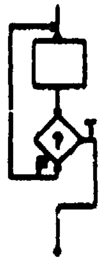
```
loop
  SAMPLE_LINE;
  UPDATE_TOTALS;
end loop;
```

Student Notes:

C-18

EXIT STATEMENTS

- Exits the innermost enclosing loop
- May have multiple exit statements



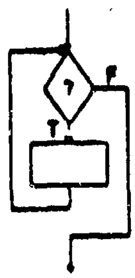

```
loop
  [- - statements]
  exit [when CONDITION];
  [- - statements]
end loop;
```

```
loop
  SAMPLE_LINE;
  UPDATE_TOTALS;
  exit when FINISHED;
end loop;
```

C-19

WHILE LOOP

- Indefinite iteration




```
while CONDITION loop
  - - statements
end loop;
```

```
while not FOUND loop
  SEARCH_LIST (ITEM, LIST, FOUND);
  ITEM := ITEM + 1;
end loop;
```

C-20

FOR LOOP

- Definite iteration

```
for LOOP_PARAMETER in DISCRETE_RANGE loop

  - - statements

end loop;
```

```
for COLOR in COLOR_TYPE loop
  COLOR_IO.PUT ( COLOR );
  NEXT_IO.NEW-LINE;
end loop;
```

```
for INDEX in LIST'RANGE loop
  TOTAL := TOTAL + LIST (INDEX);
end loop;
```

SUBPROGRAMS

- Modularity S-1
- Abstraction
- Information Hiding

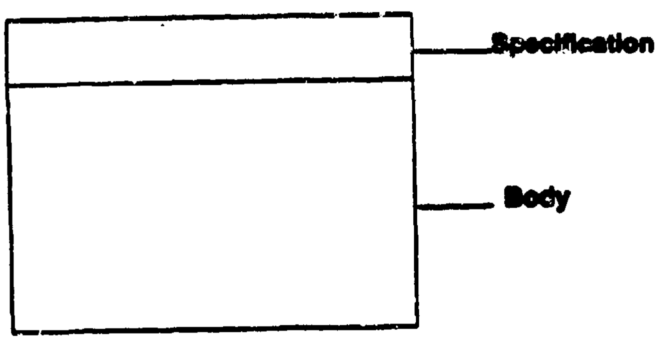
FORMS

- Procedures
  - Abstract action S-2
  - Invoked by a procedure call
- Functions
  - Returns a single value
  - An expression

PROCEDURE PARTS

Procedures are divided into two parts:

- Specification
  - Defines interfaces
- Body
  - Defines implementation details S-3



Student Notes:

SPECIFICATIONS — DECLARATIONS

- Specifies the procedure name
- Used in packages and for specifying visibility with embedded procedures

Example:

S-4

```
procedure MASSIVE_RETALIATION;  } Procedure
procedure ENABLE_ECM;           } Declarations

procedure ANSWER_PHONE is -- specification part of procedure

begin
   :
  -- implementation details
   :
end ANSWER_PHONE;
```

PROCEDURE BODIES

S-5

- Procedure bodies are further broken down into two parts
  - Declarative Part
    - Declare items local to that procedure
    - Between "is" and "begin"
  - Executable Part
    - Contains executable statements
    - Following "begin" and through "end"

PROCEDURES

S-6

```
procedure STUFF_IT is
   -- declare some stuff - Declarative Part
begin
   :
  -- do some stuff - Executable Part
   :
end STUFF_IT;
```

PARAMETERLESS PROCEDURES

- No information passed to or from the procedure
- All information used is purely local

```
with TEXT_IO;
procedure DISPLAY_MENU is

begin
  TEXT_IO.PUT("[1] ENTER AN ITEM");
  TEXT_IO.NEW_LINE;
  TEXT_IO.PUT("[2] RETRIEVE AN ITEM");
  TEXT_IO.NEW_LINE;
  TEXT_IO.PUT("[3] CLEAR ALL ITEM");
  TEXT_IO.NEW_LINE;
end DISPLAY_MENU;
```

S-7

PROCEDURE CALLS

- Invokes execution of corresponding procedure
- Passes control to called procedure
- Control passed back upon completion of execution

```
with TEXT_IO;
with DISPLAY_MENU;
procedure PROCESS_ITEM is
  MAX_OPTIONS : constant := 3;
  type OPTIONS is range 1.. MAX_OPTIONS;
  CHOICE:  OPTIONS;
  package CHOICE_IO is new TEXT_IO.INTEGER_IO(OPTIONS);
begin
  DISPLAY_MENU; -- Control passed to DISPLAY_MENU
  CHOICE_IO.GET (CHOICE);
  case CHOICE is

   when 1 => ____
             ____
             ____
   when 2 => ____
             ____
             ____
   when 3 => ____
             ____
             ____
  end case;
end PROCESS_ITEM;
```

S-8

Student Notes:

PARAMETERS

S-9

- A way to pass information to and from procedures
- Enforces S.F. principle of localization
- Two kinds of subprogram parameters
  - FORMAL
  - ACTUAL

FORMAL PARAMETERS

S-10

- Defined in procedure specification
- Defines object names and types to be used locally in procedure

```
procedure MASSIVE_RETALIATION (CODE: in STRING);
```

ACTUAL PARAMETERS

S-11

- Declared in calling program unit
- Types of actual and formal parameters must be compatible

```
with MASSIVE_RETALIATION;
procedure RESPONSE is
 MAX_ECM_LEVEL : constant := 5;
 type ECM_LEVEL_TYPE is range 1..MAX_ECM_LEVEL;
 MY_CODE : constant STRING := "BLASTEM";
 MY_LEVEL : ECM_LEVEL_TYPE := 3;
 procedure ENABLE_ECM (LEVEL : in ECM_LEVEL_TYPE) is separate;

begin
  .
  .
  .
  MASSIVE_RETALIATION (MY_CODE);
  .
  .
  .
  ENABLE_ECM (MY_LEVEL);
  .
  .
  .
end RESPONSE;
```

PARAMETER MODES

- Procedure formal parameters have three allowable modes
  - in—actual parameters send information to the called procedure (treated as constant)
  - out—actual parameters receive information from the called procedure (can only be updated)
  - in out—actual parameters send and receive information from the called procedure (treated as object)
  - If no mode is stated in the formal part, then "in" is used as the default

S-12

```
with TEXT_IO;
procedure PUT_N_GET is

 MAX_INT : constant := 100;
 type MY_INT is range 1 .. MAX_INT;
 package INT_IO is new TEXT_IO.INTEGER_IO(MY_INT);

 INT_1, INT_2 : MY_INT := MY_INT'FIRST;

 procedure EXCHANGE ( FIRST, SECOND : in out MY_INT ) is
  TEMP : MY_INT := FIRST;
 begin
  FIRST := SECOND;
  SECOND : TEMP;
 end EXCHANGE;

begin

 INT_IO.GET(INT_1);
 INT_IO.GET(INT_2);

 EXCHANGE ( INT_1, INT_2 );

 INT_IO.PUT(INT_1);
 INT_IO.PUT(INT_2);

end PUT_N_GET;
```

Student Notes:

FUNCTIONS

S-13

- Different than procedures
  - Return a single value
  - Must have a return statement (optional with procedures)
  - Called as an expression
  - Only has "in" modes
- Same as procedures
  - Has a specification and a body
  - Enforces S.E. principles

STRUCTURE

S-14

- Specification
  - Defines interfaces

```
function END_OF_FILE return BOOLEAN;
```

- Body
  - Implementation details
  - Must have a return statement

```
function END_OF_FILE return BOOLEAN is
.
.
.
begin
.
.
.
 if _____ then
  return TRUE;
 else
  return FALSE;
 end if;
end END_OF_FILE;
```

Student Notes:

PARAMETERLESS FUNCTIONS

- All information needed is local to that function

S-15

```
function HOSTILE return BOOLEAN;

function WEAPON_ARMED return BOOLEAN;

function NUMBER_BOGEYS return BOGEY_COUNT;
```

FUNCTION CALLS

- Called as an expression

S-16

```
BOGEYS := NUMBER_BOGEYS;
if HOSTILE then
  if WEAPON_ARMED then
    FIRE_WEAPON;
  end if;
end if;
.
.
.
```

FUNCTION PARAMETERS

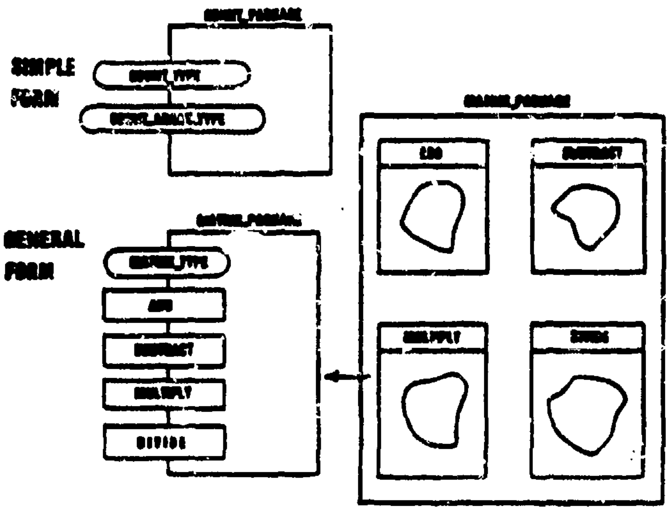
- Can only be of mode "in"

S-17

```
.
.
AMRAAM : WEAPON_TYPE;
CLOSEST_BOGEY : BOGEY_TYPE;
function HOSTILE (BOGEY : BOGEY_TYPE) return BOOLEAN is separate;
function WEAPON_ARMED
  (WEAPON : in WEAPON_TYPE) return BOOLEAN is separate;
begin
 .
 .
 .
 if HOSTILE (CLOSEST_BOGEY) then
   if WEAPON_ARMED (AMRAAM) then
     FIRE_WEAPON (AMRAAM);
   end if;
 end if;
 .
 .
 .
 .
```

PACKAGES

- Allow the specification of groups of logically related entities P-1
- Simple form—collection of data declarations
- General form—groups of related entities including subprograms which can be called from outside the package while inner details remain concealed and protected



P-2

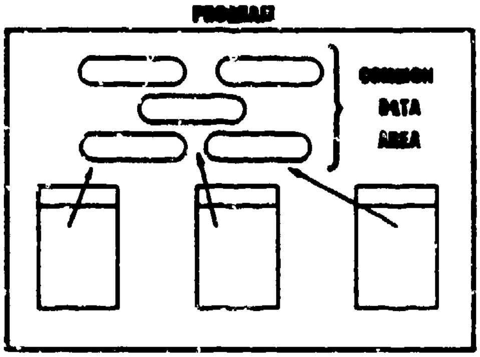
STRUCTURING TOOL

- Packages allow us to partition up the solution space into logically distinct, self-contained components P-3
- A different structuring capability from the traditional use of nesting and independent compilation
- Different topology

Student Notes:

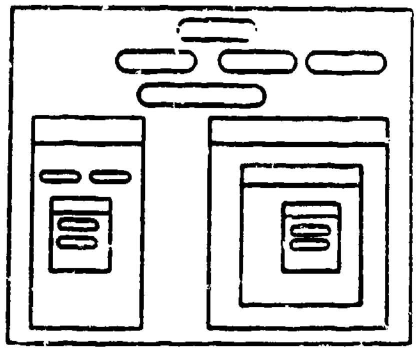
P-4

TRADITIONAL LANGUAGE TOPOLOGY
(FORTRAN / COBOL)



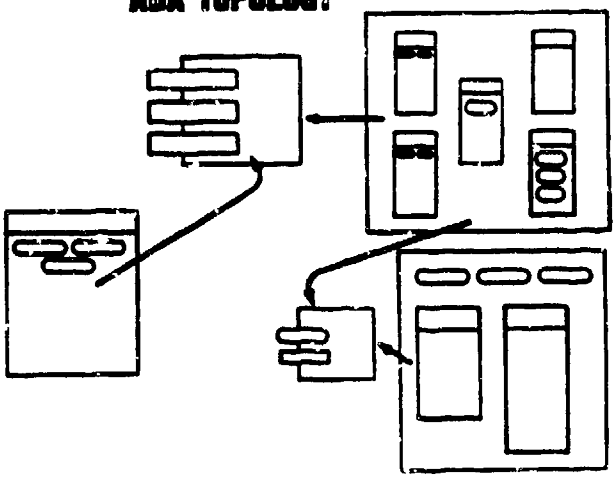
P-5

TRADITIONAL LANGUAGE TOPOLOGY
(PASCAL)



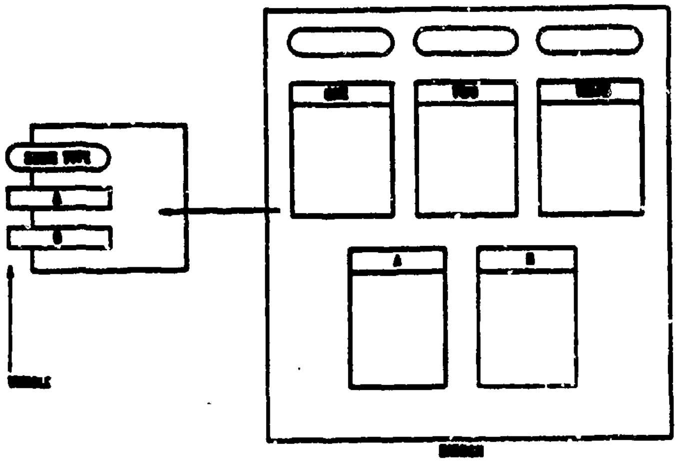
P-6

ADA TOPOLOGY



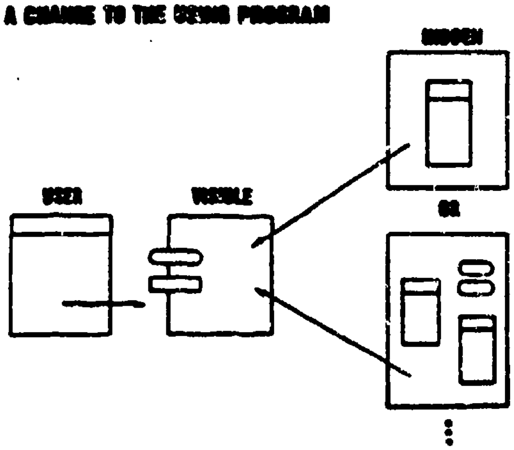
Student Notes:

- PACKAGES DIRECTLY SUPPORT ABSTRACTION, INFORMATION HIDING, AND LOCALIZATION



P-7

- SINCE LOCALIZED DETAILS CAN BE HIDDEN, MODIFICATIONS CAN BE MADE TO THESE DETAILS WITHOUT THE REQUIREMENT FOR A CHANGE TO THE USING PROGRAM



P-8

- Packages provide a facility for progressing toward the development of a reusable software components industry
- Different approach from current practice

P-9

Student Notes:

PACKAGE STRUCTURE

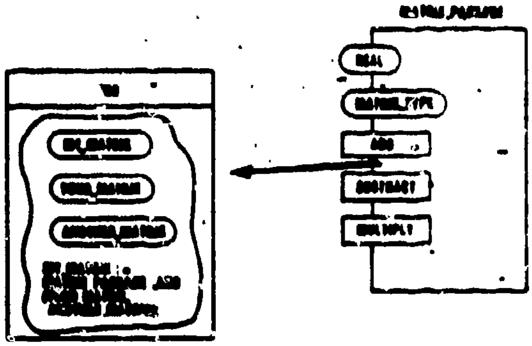
P-10

- Visible part – Package Specification
- Hidden part – Package Body
  - – Sometimes Optional
  - – Separately Compileable

P-11

PACKAGE SPECIFICATION

- DEFINES ENTITIES VISIBLE TO THE USER OF THE PACKAGE



P-12

```
package MATRIX_PACKAGE is

 ACCURACY          : constant := 15;
 MAX_MATRIX_SIZE : constant := 10;

 type REAL is digits ACCURACY;
 type MATRIX_INDEX_TYPE is range 1 .. MAX_MATRIX_SIZE;
 type MATRIX_ELEMENTS_TYPE is array ( MATRIX_INDEX_TYPE range <> ,
            MATRIX_INDEX_TYPE range <> ) of REAL;

 type MATRIX_TYPE( SIZE : MATRIX_INDEX_TYPE := 2 ) is record
  ELEMENTS : MATRIX_ELEMENTS_TYPE( 1 .. SIZE, 1 .. SIZE );
 end record;

 function ADD ( FIRST, SECOND : MATRIX_TYPE ) return MATRIX_TYPE;

 function SUBTRACT ( FIRST, SECOND : MATRIX_TYPE ) return MATRIX_TYPE;

 function MULTIPLY ( FIRST, SECOND : MATRIX_TYPE ) return MATRIX_TYPE;

end MATRIX_PACKAGE;
```

Student Notes:

```
with MATRIX_PACKAGE;
procedure USER is

  MY_MATRIX       : MATRIX_PACKAGE.MATRIX_TYPE(3);
  YOUR_MATRIX     : MATRIX_PACKAGE.MATRIX_TYPE(3);
  ANOTHER_MATRIX  : MATRIX_PACKAGE.MATRIX_TYPE(3);

begin

  YOUR_MATRIX.ELEMENTS    := ( others => ( others => 2.0 ) );
  ANOTHER_MATRIX.ELEMENTS := ( others => ( others => 4.0 ) );
  MY_MATRIX               := MATRIX_PACKAGE.ADD ( YOUR_MATRIX, ANOTHER_MATRIX );

  ANOTHER_MATRIX          := MATRIX_PACKAGE.MULTIPLY ( MY_MATRIX, MY_MATRIX );

end USER;
```

P-13

PACKAGE BODY

P-14

- If needed contains
  - Bodies of units declared in the specification
  - Any other declarations needed (these are not available to user)

Student Notes:

P-15

```
package body MATRIX_PACKAGE is

 function ADD ( FIRST, SECOND : MATRIX_TYPE ) return MATRIX_TYPE is
  TEMP_MATRIX : MATRIX_TYPE(FIRST.SIZE);
 begin
  for INDEX_1 in FIRST.ELEMENTS'RANGE(1) loop
   for INDEX_2 in FIRST.ELEMENTS'RANGE(2) loop
    TEMP_MATRIX.ELEMENTS(INDEX_1, INDEX_2) :=
                              FIRST.ELEMENTS(INDEX_1, INDEX_2) +
                              SECOND.ELEMENTS(INDEX_1, INDEX_2);
   end loop;
  end loop;
  return TEMP_MATRIX;
 end ADD;

 function SUBTRACT ( FIRST, SECOND : MATRIX_TYPE ) return MATRIX_TYPE is
  TEMP_MATRIX : MATRIX_TYPE(FIRST.SIZE);
 begin
  for INDEX_1 in FIRST.ELEMENTS'RANGE(1) loop
   for INDEX_2 in FIRST.ELEMENTS'RANGE(2) loop
    TEMP_MATRIX.ELEMENTS(INDEX_1, INDEX_2) :=
                               FIRST.ELEMENTS(INDEX_1, INDEX_2) -
                               SECOND.ELEMENTS(INDEX_1, INDEX_2);
   end loop;
  end loop;
  return TEMP_MATRIX;
 end SUBTRACT;

 function MULTIPLY ( FIRST, SECOND : MATRIX_TYPE ) return MATRIX_TYPE is

  TEMP_MATRIX : MATRIX_TYPE(FIRST.SIZE);
  SUM : REAL := 0.0;

 begin
  for INDEX_1 in FIRST.ELEMENTS'RANGE(1) loop
   for INDEX_2 in FIRST.ELEMENTS'RANGE(2) loop
    SUM := 0.0;
    for INDEX_3 in FIRST.ELEMENTS'RANGE(2) loop
     SUM := SUM +
            FIRST.ELEMENTS(INDEX_1, INDEX_3) *
            SECOND.ELEMENTS(INDEX_3, INDEX_2);
    end loop;
    TEMP_MATRIX.ELEMENTS(INDEX_1, INDEX_2) := SUM;
   end loop;
  end loop;
  return TEMP_MATRIX;
 end MULTIPLY;

end MATRIX_PACKAGE;
```

Student Notes:

PRIVATE TYPES

P-16

- Allow the definition of powerful abstract data types
- Defined in the private portion of the package specification

FORMS

P-17

- Private
  - — Provide operations := = /=
  - — Provide subprograms defined in package specification
- Limited Private
  - — Only subprograms defined in package specification

P-18

```
package BASKIN_ROBBINS is

   MAX_NUMBER : constant := 100;
   type NUMBERS is range 1 .. MAX_NUMBER;

   procedure GET_NUMBER (NEXT_NUMBER : out NUMBERS );
   function NOW_SERVING return NUMBERS;
   procedure SERVE ( A_NUMBER : in NUMBERS );

end BASKIN_ROBBINS;
```

Student Notes:

```
package body BASKIN_ROBBINS is

 NUMBER_HOLDER : NUMBERS := NUMBERS'FIRST;

 procedure GET_NUMBER ( NEXT_NUMBER : out NUMBERS ) is
 begin
  NEXT_NUMBER     := NUMBER_HOLDER;
  NUMBER_HOLDER := NUMBER_HOLDER + 1;
 end GET_NUMBER;

 function NOW_SERVING return NUMBERS is separate;

 procedure SERVE ( A_NUMBER : in NUMBERS ) is separate;

end BASKIN_ROBBINS;
```

P-19

```
with BASKIN_ROBBINS; use BASKIN_ROBBINS;
procedure ICE_CREAM is

 YOUR_NUMBER : BASKIN_ROBBINS.NUMBERS;

begin
 BASKIN_ROBBINS.GET_NUMBER( YOUR_NUMBER );
 loop

  if YOUR_NUMBER = NOW_SERVING then

   BASKIN_ROBBINS.SERVE( YOUR_NUMBER );
   exit;
  end if;
 end loop;
end ICE_CREAM;
```

P-20

```
with BASKIN_ROBBINS;
procedure ICE_CREAM is
 YOUR_NUMBER : BASKIN_ROBBINS.NUMBERS;
 use BASKIN_ROBBINS;
begin
 BASKIN_ROBBINS.GET_NUMBER( YOUR_NUMBER );
 loop
 if YOUR_NUMBER = NOW_SERVING then
   BASKIN_ROBBINS.SERVE( YOUR_NUMBER );
   exit;
  else
   YOUR_NUMBER := YOUR_NUMBER + 1;
  end if;
 end loop;
end ICE_CREAM;
```

P-21

Student Notes:

```
package BASKIN_ROBBINS is

 type NUMBERS is private;


 procedure GET_NUMBER( NEXT_NUMBER : out NUMBERS);
 function NOW_SERVING return NUMBERS;
 procedure SERVE ( A_NUMBER : in NUMBERS );

private

  MAX_NUMBER : constant := 100;
  type NUMBERS is range 0 .. MAX_NUMBER;

end BASKIN_ROBBINS;
```

P-22

```
with BASKIN_ROBBINS;  use BASKIN_ROBBINS;
procedure ICE_CREAM is

 YOUR_NUMBER : BASKIN_ROBBINS.NUMBERS;

begin

 BASKIN_ROBBINS.GET_NUMBER( YOUR_NUMBER );

 loop

  if YOUR_NUMBER = NOW_SERVING then


    BASKIN_ROBBINS.SERVE( YOUR_NUMBER );
    exit;

  else

    YOUR_NUMBER := BASKIN_ROBBINS.NOW_SERVING;

  end if;

 end loop;

end ICE_CREAM;
```

P-23

Student Notes:

```
package BASKIN_ROBBINS is

 type NUMBERS is limited private.

 procedure GET_NUMBER( NEXT_NUMBER   out NUMBERS)
 function NOW_SERVING return NUMBERS.
 function IS_EQUAL (LEFT, RIGHT : NUMBERS) return BOOLEAN;
 procedure SERVE ( A_NUMBER : in NUMBERS );

private

 MAX_NUMBER : constant := 100;
 type NUMBERS is range 0 .. MAX_NUMBER;

end BASKIN_ROBBINS,
```

P-24

```
with BASKIN_ROBBINS;
procedure ICE_CREAM is

 YOUR_NUMBER   BASKIN ROBBINS NUMBERS

 procedure GOTO  DQ is separate.

begin

 BASKIN_ROBBINS.GET_NUMBER( YOUR_NUMBER );

 loop

  if BASKIN_ROBBINS IS_EQUAL( YOUR_NUMBER
                             BASKIN ROBBINS NOW_SERVING ) then

    BASKIN_ROBBINS SERVE( YOUR_NUMBER ).
    exit;

  else

    GOTO DO.
    exit.

  end if;

 end loop;

end ICE_CREAM;
```

P-25

Student Notes:

APPLICATIONS OF PACKAGES

- Named collections of entities P-26
- Groups of related subprograms
- Encapsulated data types

NAMED COLLECTION OF ENTITIES

P-27

```
package METRIC_CONVERSIONS is

  CM_PER_INCH  : constant := 2.54;
  CM_PER_FOOT  : constant := 12*CM_PER_INCH;
  CM_PER_YARD  : constant := 3*CM_PER_FOOT;
  KM_PER_MILE  : constant := 1.609_344;

end METRIC_CONVERSIONS;
```

GROUPS OF RELATED SUBPROGRAMS

- Visible declarations of externally usable subprograms P-28
- Hidden implementation/shared internal entities

ENCAPSULATED DATA TYPES

P-29

- Define abstract data types
- Private/limited private types

Students Notes:

EXCEPTIONS

E-1

- Purpose
- Declaring Exceptions
- Exception Handlers
- Raising Exceptions
- Propagation

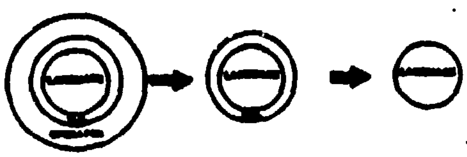
RELIABILITY

E-2

- A critical element of many mission critical systems
- A traditional problem area
- Life and property depend on software

ERRORS HAVE OCCURRED, DO OCCUR, AND
WILL CONTINUE TO OCCUR!

ERROR HANDLING LEVELS



E-3

Student Notes:

UNDERSTANDABILITY

E-4

- Much of the code written/read deals with abnormal circumstances.
- To grasp the meaning of a section of code, a maintenance programmer must sort through the abnormal to find the main meaning.
- Traditional languages lack the ability to deal with normal and abnormal as distinct features.

DEFINITION

E-5

- An "exception" is the name attached to a particular exceptional situation, user-defined or predefined.
- When the particular situation occurs, the exception is said to be "raised."
- The response to the raised exception is called the exception "handler."

PREDEFINED EXCEPTION

E-6

PACKAGE_STANDARD

- NUMERIC_ERROR
- CONSTRAINT_ERROR
- PROGRAM_ERROR
- STORAGE_ERROR
- TASKING_ERROR

- TEXT_IO
- DATA_ERROR
- USE_ERROR
- NAME_ERROR
- STATUS_ERROR
- MODE_ERROR
- END_ERROR
- LAYOUT_ERROR
- DEVICE_ERROR

DECLARATION

- An exception can be declared in any declarative part
- It follows the same visibility rules as any other declaration. E-7
- Form

```
OUT_OF_LIMITS  : exception;
RANGE_ERROR    : exception;
STACK_OVERFLOW : exception;
```

```
package INTEGER_STACK is

  MAX_NUMBER   :  constant  :=  10_000;
  type NUMBERS is range -MAX_NUMBER .. MAX_NUMBER;
  type STACK_TYPE is private;
  procedure PUSH (A_NUMBER : in NUMBERS;
                  ON       : in out STACK_TYPE);
  procedure POP  (A_NUMBER : out NUMBERS;
                  OFF_OF   : in out STACK_TYPE);
  STACK_OVERFLOW  : exception;
  STACK_UNDERFLOW : exception;
private
  ...
end INTEGER_STACK;
```

E-8

FRAME

- Construct used to capture exceptions and declare exception handlers

```
begin
  --sequence of statements
  [exception
    exception_handler
    {exception_handler}]
end;

exception_handler::=
  when exception_choice {| exception_choice} =>
                sequence_of_statements

exception choice ::= exception_name | others
```

E-9

Student Notes:

EXCEPTION HANDLER

E-10

- Optional part of a frame that can contain responses to exceptions raised in the frame

```
begin
  -- statements
exception
  when DATA_ERROR => -- statements
  when CONSTRAINT_ERROR => -- statements
  when others => -- statements
end;
```

PROCESS

E-11

- When an exception is raised within a frame, processing is immediately suspended.
- What happens next depends on the presence or absence of an appropriate exception handler.
  - Handle exception within an exception handler
  - Propagate exception

RAISING AN EXCEPTION

E-12

- Can be raised implicitly by the run time system
- Can be raised explicitly by use of the raise statement

  raise EXCEPTION_NAME;

Student Notes:

```
package SIMPLE_STACK is

  type STACK_TYPE is limited private;
  subtype ELEMENT_TYPE is CHARACTER;

  procedure PUSH (A_VALUE : in ELEMENT_TYPE;
                  A_STACK : in out STACK_TYPE);

  procedure POP (A_VALUE  : out ELEMENT_TYPE;
                 A_STACK  : in out STACK_TYPE);

  STACK_OVERFLOW, STACK_UNDERFLOW : exception;

private
  MAXIMUM_SIZE : CONSTANT := 50;
  type STACK_SIZE is range 1 .. MAXIMUM SIZE;
  type LIST_TYPE is array (STACK_SIZE) of
                             ELEMENT_TYPE;
  type STACK_TYPE is
    record
      LIST : LIST_TYPE;
      CURRENT_POSITION : STACK_SIZE := 1;
    end record;

end SIMPLE_STACK;
```

E-13

```
package body SIMPLE_STACK is

  procedure POP (A_VALUE : out ELEMENT_TYPE;
                 A_STACK : in out STACK_TYPE) is
begin
  A_STACK.CURRENT_POSITION :=
                 A_STACK.CURRENT_POSITION - 1;
  A_VALUE := A_STACK.LIST (A_STACK.CURRENT_POSITION);
  exception
    when CONSTRAINT_ERROR =>
      raise STACK_UNDERFLOW;
end POP;

procedure PUSH (A_VALUE : in ELEMENT_TYPE;
                A_STACK : in out STACK_TYPE) is

begin

  A_STACK.LIST (A_STACK.CURRENT_POSITION) := A_VALUE;

  A_STACK.CURRENT_POSITION :=
          A_STACK.CURRENT_POSITION +1;

exception
  when CONSTRAINT_ERROR =>
    raise STACK_OVERFLOW;

end PUSH;
```

E-14

Student Notes:

E-15

```
with TEXT_IO, SIMPLE_STACK;
procedure STACK_USER is

  package COUNT_IO is new TEXT_IO.INTEGER_IO
                                   (LONG_INTEGER);
  MY_STACK : SIMPLE_STACK.STACK_TYPE;
  COUNTER  : LONG_INTEGER := 0;

begin

  loop

    SIMPLE_STACK.PUSH ('a', MY_STACK);
    COUNTER := COUNTER + 1;

  end loop;

exception

  when SIMPLE_STACK.STACK_OVERFLOW =>
    TEXT_IO.PUT ("Pushed ");
    COUNT_IO.PUT (COUNTER);
    TEXT_IO.PUT_LINE (" times");

end STACK_USER;
```

GENERICS

- Purpose
- Generic Declarations
- Generic Instantiations
- Generic Parameters

G-1

GOALS AND PRINCIPLES OF SOFTWARE ENGINEERING SUPPORTED BY GENERICS

- Reliability
- Understandability
- Modifiability
- Efficiency
- Modularity
- Abstraction
- Localization
- Information Hiding

G-2

What is Software Reusability?

Why is Reusability important?

Who should be concerned with Reusability?

G-3

Student Notes:

G-4

```
procedure DUPLICATION is

  type PERSON is ...
  type TABLE  is ...
  type COUNT  is ...
  type NAME   is ...

  procedure SWAP_PEOPLE (LEFT, RIGHT : in out PERSONS) is
    TEMP  : PERSON := LEFT;
  begin
    LEFT  := RIGHT;
    RIGHT := TEMP;
  end SWAP_PEOPLE;

  procedure SWAP_TABLES (LEFT, RIGHT : in out TABLE) is
    TEMP  : TABLE := LEFT;
  begin
    LEFT  := RIGHT;
    RIGHT := TEMP;
  end SWAP_TABLES;
  procedure SWAP_COUNTS (LEFT, RIGHT : in out COUNT) is
    ...

  procedure SWAP_NAMES (LEFT, RIGHT : in out NAME) is
    ...
begin
  .
  .
  .
end DUPLICATION;
```

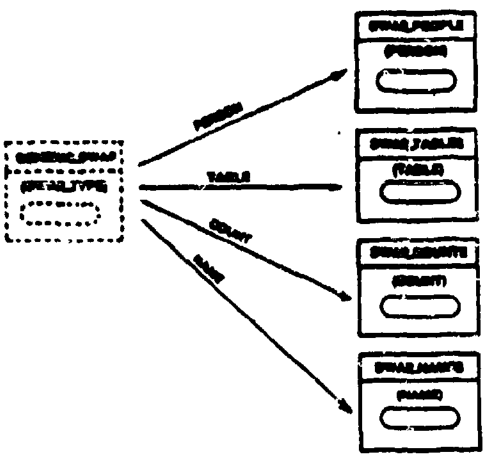
G-5

```
generic
  type SWAP_TYPE is private;
procedure GENERIC_SWAP (LEFT, RIGHT : in out SWAP_TYPE);

procedure GENERIC_SWAP (LEFT, RIGHT : in out SWAP_TYPE) is
  TEMP  : SWAP_TYPE := LEFT;
begin
  LEFT  := RIGHT;
  RIGHT := TEMP;
end GENERIC_SWAP;
```

Student Notes:

```
with GENERIC_SWAP;
procedure NON_DUPLICATION is

  type PERSON is ...
  type TABLE  is ...
  type COUNT  is ...
  type NAME   is ...

  procedure SWAP_PEOPLE is new GENERIC_SWAP (SWAP_TYPE =>
                                               PERSON);
  procedure SWAP_TABLES is new GENERIC_SWAP (SWAP_TYPE =>
                                               TABLE);
  procedure SWAP_COUNTS is new GENERIC_SWAP (SWAP_TYPE =>
                                               COUNT);
  procedure SWAP_NAMES  is new GENERIC_SWAP (SWAP_TYPE =>
                                               NAME);
begin
  ...
end NON_DUPLICATION;
```

G-6



G-7

DEFINITION

G-8

- A generic is a template for a program unit.
- Instantiation gives us an actual program unit from that template.

Student Notes:

GENERIC DECLARATIONS

G-9

- Two Classes
  - — Generic Subprograms

    ```
    generic
      — GENERIC FORMAL PARAMETERS
    procedure (function) ...
    ```

  - — Generic Packages

    ```
    generic
      — GENERIC FORMAL PARAMETERS
    package . . .
    ```

GENERIC INSTANTIATION

G-10

- Creates an actual instance of a generic unit
- "Fills in" the generic formal parameter with an actual parameter

GENERIC PARAMETERS

G-11

- Type
- Value
- Object
- Subprogram

Student Notes:

MATCHING RULES

G-12

type IDENTIFIER is digits <>;	Any floating point type
type IDENTIFIER is delta <>;	Any fixed point type
type IDENTIFIER is range <>;	Any integer type
type IDENTIFIER is (<>);	Any discrete type
type IDENTIFIER is array (INDEX_TYPE) of COMPONENT_TYPE;	Any constrained array type with same INDEX_TYPE and COMPONENT_TYPE
type IDENTIFIER is array (INDEX_TYPE range <>) of COMPONENT_TYPE;	Any unconstrained array type with same INDEX_TYPE and COMPONENT_TYPE

MATCHING RULES (Continued)

G-13

type IDENTIFIER is access NAME;	Any access type that designates same NAME type (subject to constraint rule)
type IDENTIFIER is private;	Any type except a limited type
type IDENTIFIER is limited private;	Any type
OBJECT : in TYPE_NAME;	Value or object that is of same type as TYPE_NAME
OBJECT: in out TYPE_NAME;	Object that is of same type as TYPE_NAME
with procedure NAME (PARAMETERS) [is <> | is DEFAULT_NAME];	Procedure that conforms to parameter number and types
with function NAME (PARAMETERS) [is <> | is DEFAULT_NAME];	Function that conforms to parameter number and types and has same result type

Student Notes:

G-14

```
generic
  type ELEMENT_TYPE is private.
  SIZE_OF_STACK : in POSITIVE;

package BOUNDED_GENERIC_STACK is

  type STACK_TYPE is limited private.
  procedure PUSH (AN_ELEMENT : in ELEMENT_TYPE;
                  ON : in out STACK_TYPE).
  procedure POP  (AN_ELEMENT  : out ELEMENT_TYPE:
                  OFF_OF : in out STACK_TYPE);
private

  type STACK_COUNT is range 0 . SIZE_OF_STACK.
  type STACK_ELEMENTS is array (STACK_COUNT)
                             of ELEMENT_TYPE
  type STACK_TYPE is
    record
      TOP    . STACK_COUNT := 0.
      BOTTOM : STACK_COUNT := 1:
      LIST   : STACK_ELEMENTS:
    end record
end BOUNDED_GENERIC_STACK;
```

Student Notes:

G-15

```
with BOUNDED_GENERIC_STACK.
procedure DEMO_STACK is
  LENGTH : constant :- 80;
  subtype NAME_TYPE is STRING (1 .LENGTH);

  package NAME_STACK is new BOUNDED_GENERIC_STACK
                          (ELEMENT_TYPE => NAME_TYPE,
                           SIZE_OF_STACK => 100);

  STACK_OF_NAMES    : NAME_STACK.STACK_TYPE;

begin
  ...
end DEMO_STACK;
```

G-16

GENERIC BODIES

- Generic Formal Parameters
- Writing Generic Bodies

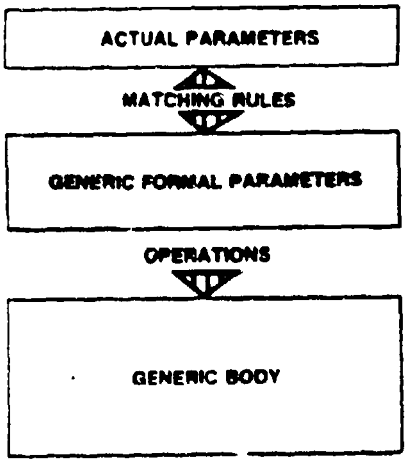
G-17

GENERIC FORMAL PARAMETERS

Describes two things:

- Matching requirements for actual parameters
- Operations that can be assumed within the generic body

Student Notes:



G-18

```
generic
  type INTEGER_TYPE is range <>;
procedure NEXT (ANY_INTEGER : in out INTEGER_TYPE);

procedure NEXT (ANY_INTEGER : in out INTEGER_TYPE) is
begin
  ANY_INTEGER := ANY_INTEGER + 1;
exception
  when CONSTRAINT_ERROR =>
    ANY_INTEGER := INTEGER_TYPE'FIRST;
end NEXT
```

G-19

```
generic
  type DISCRETE_TYPE is (<>);
procedure NEXT (ANY_DISCRETE_VALUE : in out DISCRETE_TYPE);

procedure NEXT (ANY_DISCRETE_VALUE : in out DISCRETE_TYPE) is
begin
  -- "+" NOT AVAILABLE
  ANY_DISCRETE_VALUE := DISCRETE_TYPE 'SUCC (ANY_DISCRETE_VALUE);
exception
  when CONSTRAINT_ERROR =>
    ANY_DISCRETE_VALUE := DISCRETE_TYPE'FIRST;
end NEXT;
```

G-20

Student Notes:

G-21

GENERIC BODY

- Defines implementation of the generic unit
- Can use operations available from the generic formal parameters

G-22

USING FORMAL TYPE PARAMETERS

Specify which operations are available for the type

G-23

```
generic
   PROMPT : in STRING;
   type ANY_INTEGER_TYPE is range <>;
procedure GET_VALID_INTEGER (AN_INTEGER : out ANY_INTEGER_TYPE);
```

Student Notes:

```
with TEXT_IO,
procedure GET_VALID_INTEGER (AN_INTEGER : out ANY_INTEGER_TYPE) is

  package INT_IO is new TEXT_IO.INTEGER_IO (ANY_INTEGER_TYPE);

begin
  loop
    begin
      TEXT_IO.PUT (PROMPT);
      INT_IO.GET (AN_INTEGER);
      exit;
    exception
      when others =>
        TEXT_IO.SKIP_LINE;
        TEXT_IO.PUT_LINE ("**INVALID**")
    end;
  end loop;
end GET_VALID_INTEGER.
```

G-24

```
generic
  SIZE : in NATURAL,
  type ELEMENTS is private.
package STACKS is

  type STACK_TYPE is limited private

  procedure PUSH (STACK : in out STACK_TYPE
                  VALUE : in    ELEMENTS)

  procedure POP  (STACK : in out STACK_TYPE
                  VALUE :    out ELEMENTS)

private
                    -- Stack size determined by
                    -- generic value parameter
  type NUMBER_OF_ELEMENTS is range 0 .. SIZE
  type ELEMENT_ARRAY is array (NUMBER_OF_ELEMENTS)
                                              of ELEMENTS
  type STACK_TYPE is
    record
      DATA : ELEMENT_ARRAY.
      TOP : NUMBER_OF_ELEMENTS    0
    end record.
end STACKS.
```

G-25

```
generic
   type ELEMENTS is private;
   type INDEX is (<>);
   type ARRAY_TYPE is array (INDEX) of ELEMENT;
   with function "<" (LEFT, RIGHT : ELEMENT) return BOOLEAN;

procedure SORT (LIST : in out ARRAY_TYPE);

procedure SORT (LIST : in out ARRAY_TYPE) is
   TEMP : ELEMENT;

begin -- SORT
   for OUTER in INDEX'first..INDEX'pred(INDEX'last) loop
      for INNER in INDEX'succ(OUTER)..INDEX'last loop
         if LIST(INNER) < LIST(OUTER) then
            TEMP := LIST(INNER);
            LIST(INNER) := LIST(OUTER);
            LIST(OUTER) := TEMP;
         end if;
      end loop;
   end loop;
end SORT;
```

G-26

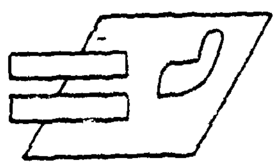
Student Notes:

TASKS

- Purpose
- Independent Tasks
- Communicating Tasks
- Tasking Statements

T-1

TASKS



-- A task is an entity that operates in parallel with other entities

-- Tasking may be implemented on
  -- Single Processors
  -- Multi-processors
  -- Multi-computers

T-2

TASKS

- Important aspect of embedded systems
- Neglected in most languages currently in production use
  — Lack of confidence in control of parallelism
  — Low level feature
- Need an implementation independent model
- Ada draws up operating system features into the language

T-3

Student Notes:

ADA TASKING MODEL

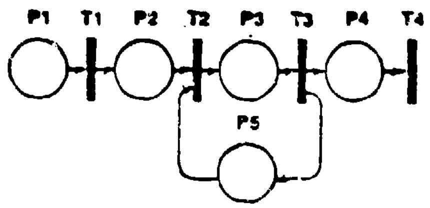
(Communicating Sequential Processes)

T-4

- Petri Net Graphs
  - Used as a tool to explain tasking model
- Parallel Independent Processes
  - "Simple" form of tasking model
- Communicating Sequential Processes
  - "Full" Ada tasking model

T-5

PETRI NET GRAPHS



T-6

PETRI NET TRANSITION RULE

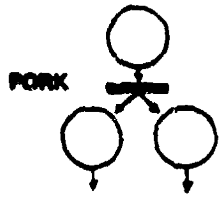
Take one token from each of the enabled transition's input places; deposit one token in each of the transition's output places

Student Notes:

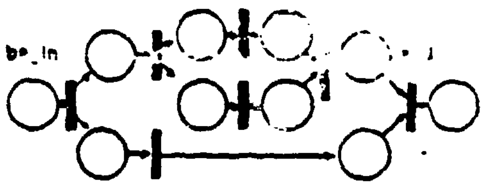
CONCURRENT PETRI NETS

- Based on the idea of a fork, to create a new thread of control

T-7



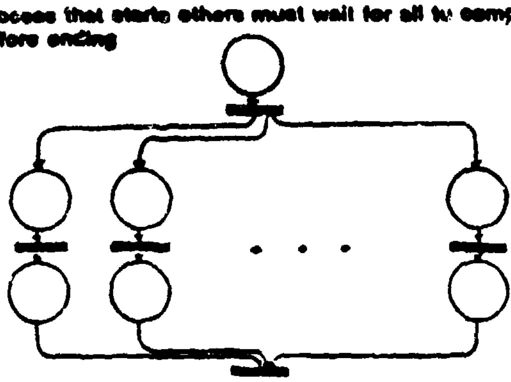
- For a process to end, it must return back to one thread of control



PARALLEL INDEPENDENT PROCESSES

- No communication, no rendezvous
- Process that starts others must wait for all to complete before ending

T-9



Student Notes:

T-10

```
procedure MAIN is
  task T1;
  task T2;
  task body T1 is
  begin
    null;
  end T1;
  task body T2 is
  begin
    null;
  end T2;
begin
  null;
end MAIN;
```

T-11

```
with TEXT_IO; use TEXT_IO;
procedure TASK_EXAMPLES is
  task PLAIN;
  task WITH_LOCAL_DECLARATIONS;
  task body PLAIN is
  begin
    null;
  end PLAIN;
  task body WITH_LOCAL_DECLARATIONS is
    FOREVER : constant STRING := "forever";
  begin
    loop
      PUT_LINE ("This task puts this message out");
      PUT (FOREVER);
      NEW_LINE;
    end loop;
  end WITH_LOCAL_DECLARATIONS;
begin
  -- both tasks activated here
  null;
  -- This subprogram does not terminate execution until
  -- all dependent tasks are ready to terminate
end TASK_EXAMPLES;
```

T-12

```
procedure MONITOR_GATE is
  task WATCH_HEAT_SENSOR;
  task WATCH_SOUND_SENSOR;
  procedure SOUND_ALARM is separate;
  task body WATCH_HEAT_SENSOR is separate;
  task body WATCH_SOUND_SENSOR is separate;
begin
  -- tasks are activated
  null;
end MONITOR_GATE;
```

Student Notes:

T-13

```
separate (MONITOR_GATE)
task body WATCH_HEAT_SENSOR is

  function DETECT_HEAT return BOOLEAN is separate;

begin
  loop
    if DETECT_HEAT then
      SOUND_ALARM;
    end if;
  end loop;
end WATCH_HEAT_SENSOR;
```

T-14

```
separate (MONITOR_GATE)
task body WATCH_SOUND_SENSOR is

  function DETECT_SOUND return BOOLEAN is separate;

begin
  loop
    if DETECT_SOUND then
      SOUND_ALARM;
    end if;
  end loop;
end WATCH_SOUND_SENSOR;
```

COMMUNICATING TASKS

T-15

- Ada Tasking Model
- Rendezvous
- Task Entries
- Communication Process

Student Notes:

T-16

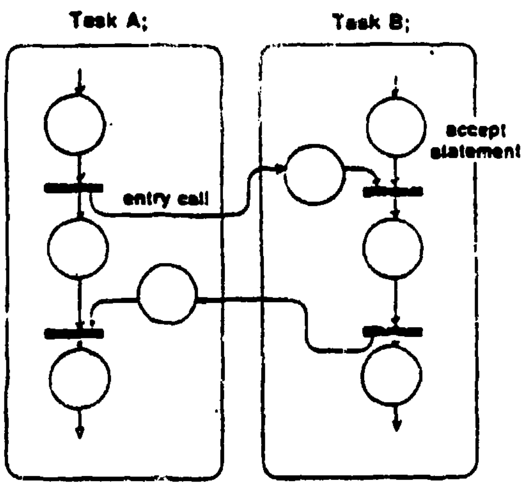
TASK COMMUNICATION

Ada Tasking Model:

Communicating Sequential Processes

T-17

RENDEZVOUS



T-18

RENDEZVOUS

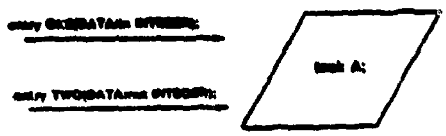
- The process in which two parallel tasks synchronize and optionally communicate
- A rendezvous is the interaction that occurs between two parallel tasks when one task has called an entry of the other task and a corresponding accept statement is being executed by the other task on behalf of the calling task

Student Notes:

RENDEZVOUS

- Defined in the specification of a task
- Define the communication paths to a task
- Are called from outside a task when the caller wishes to communicate with a task

T-19



TASK ENTRIES

```
task PRINTER_CHANNEL is
   entry PRINT (JOB : in LISTING_TYPE);
end PRINTER_CHANNEL;

task CLOCK is
   entry SET_TIME  (CURRENT : in TIME);
   entry GIVE_TIME (CURRENT : out TIME);
end CLOCK;

task LAUNCH_BOMBERS is
   entry LAUNCH;
   entry FAIL_SAFE (CODE : in PASSWORD);
ends LAUNCH_BOMBERS;
```

T-20

COMMUNICATING WITH A TASK

- Tasks are communicated with through their entries using an entry call

```
PRINTER_CHANNEL.PRINT (MY_JOB);

CLOCK.SET_TIME(NEW_TIME);
CLOCK.GIVE_TIME(THE_TIME);

LAUNCH_BOMBERS.LAUNCH;
```

T-21

Student Notes:

ENTRY CALL

T-22

- Places an entry call on the queue associated with the entry of a task
- Does not immediately start a rendezvous

ACCEPT STATEMENT

T-23

- Occurs in a task body
- Corresponds to task entries
- Specifies actions to be performed during rendezvous

RENDEZVOUS

T-24

When an entry has been called and the corresponding accept statement is reached, rendezvous occurs

– Rendezvous is the execution of the sequence of statements following the "do" and continuing to the "end"

After rendezvous is completed, the two tasks execute in parallel again

ACCEPT STATEMENT

- Syntax

```
accept_statement ::=
  accept entry_simple_name [(entry_index)][formal_part][do
    sequence_of_statements
  end [entry_simple_name]];
```

T-25

- Examples

```
accept PRINT (JOB : in LISTING_TYPE) do
  ...--sequence of statements
end;

accept SET_TIME (CURRENT : in TIME) do
  ...--sequence of statements
end;

accept LAUNCH;
```

```
...
task CHANNEL_IO is
  entry PRINT (JOB : in LISTING_TYPE);
end CHANNEL_IO;

function FREE return BOOLEAN is separate;
procedure SEND (JOB_TO_PRINT : in LISTING_TYPE) is separate;

task body CHANNEL_IO is
  LOCAL_COPY : LISTING_TYPE;
begin
  loop

    accept PRINT (JOB : in LISTING_TYPE) do
      LOCAL_COPY := JOB;
    end;
    loop
      exit when FREE;
    end loop;
    SEND (LOCAL_COPY);
  end loop;
end CHANNEL_IO;
begin  --main program
 ...
 CHANNEL_IO.PRINT (MY_JOB);
 ...
```

T-26

Student Notes:

T-27

```
procedure COUNT_DOWN is

  task SEQUENCER is
    entry ONE;
    entry TWO;
    entry THREE;
  end SEQUENCER;

  procedure DO_NOTHING is
  begin
    for INDEX in 0 .. 10_000 loop
      null;
    end loop;
  end DO_NOTHING;

  task body SEQUENCER is
  begin
    accept ONE; DO_NOTHING;
    accept TWO; DO_NOTHING;
    accept THREE;
  end SEQUENCER;
begin   --COUNT_DOWN
  SEQUENCER.ONE;
  SEQUENCER.TWO;
  SEQUENCER.THREE;
end COUNT_DOWN;
```

Student Notes:

TASKING STATEMENTS

- Delay Statement
- Select Statement
- Abort Statement

T-28

DELAY STATEMENT

delay_statement ::= delay simple_expression;

T-29

- Suspends further execution of the task for at least the time interval specified
- Simple expression must be of the predefined fixed point type DURATION

```
SECONDS : DURATION;

delay DURATION (3.0 * SECONDS);
```

```
procedure MONITOR is

  task CHECK_RADIATION_LEVEL;
  function OUT_OF_LIMITS return BOOLEAN is separate;
  procedure SOUND_ALARM is separate;

  task body CHECK_RADIATION_LEVEL is
  begin
    loop
      if OUT_OF_LIMITS then
        SOUND_ALARM;
      else
        delay 5.0;
      end if;
    end loop;
  end CHECK_RADIATION_LEVEL;

begin
  null;
end MONITOR;
```

T-30

Student Notes:

Student Notes:

SELECT STATEMENT

T-31

- Allows for choosing between multiple entries for rendezvous
- Allows for choosing the semantics of an entry call

```
select_statement ::= selective_wait |
                     conditional_entry_call |
                     timed_entry_call
```

T-32

```
task BANK_TELLER is
   entry MAKE_DEPOSIT (AMOUNT : in FLOAT);
   entry MAKE_WITHDRAWAL (DESIRED : in FLOAT,
                          AMOUNT : out FLOAT);
end BANK_TELLER;
```

T-33

```
task body BANK_TELLER is
   ...
begin
  loop
    select
      accept MAKE_DEPOSIT (AMOUNT : in FLOAT) do
        ...
      end;

    or

      accept MAKE_WITHDRAWAL (DESIRED : in FLOAT;
                              AMOUNT  : out FLOAT) do
        ...
      end;
    end select;
  end loop;
end BANK_TELLER;
```

Student Notes:

```
task BANK_TELLER is
   entry MAKE_DEPOSIT (AMOUNT : in FLOAT);
   entry MAKE_DRIVE_UP_DEPOSIT (AMOUNT : in FLOAT);

end BANK_TELLER;
```

T-34

SELECTIVE WAIT WITH ELSE

```
loop
   select
      accept MAKE_DEPOSIT (AMOUNT : in FLOAT) do
         ...
      end;
   or
      accept MAKE_DRIVE_UP_DEPOSIT (AMOUNT : in FLOAT) do
         ...
      end;
   else
      DO_FILING;
   end select;
end loop;
```

T-35

SELECTIVE WAIT WITH GUARDS

```
loop
   select
      when BANKING_HOURS =>
      accept MAKE_DEPOSIT (AMOUNT : in FLOAT) do
         ...
      end;
   or
      when DRIVE_UP_HOURS =>
      accept MAKE_DRIVE_UP_DEPOSIT (AMOUNT : in FLOAT) do
         ...
      end;
   else
      DO_FILING;
   end select;
end loop;
```

T-36

Student Notes:

SELECTIVE WAIT WITH A DELAY

T-37

```
loop
  select
    when BANKING_HOURS = >
    accept MAKE_DEPOSIT (AMOUNT : in FLOAT) do
      ...
    end;
  or
    when DRIVE_UP_HOURS = >
    accept MAKE_DRIVE_UP_DEPOSIT (AMOUNT : in FLOAT) do
      ...
    end;
  or
    delay DURATION (2.0 * HOURS);
    TAKE_A_BREAK;
  end select;
end loop;
```

SELECTIVE WAIT WITH TERMINATE

T-38

```
loop
  select
    accept MAKE_DEPOSIT (AMOUNT : in FLOAT) do
      .
      .
      .
    end;
  or
    accept MAKE_DRIVE_UP_DEPOSIT (AMOUNT : in FLOAT)
                                                    do
      .
      .
      .
    end;
  or
    terminate;
  end select;
end loop;
```

CONDITIONAL ENTRY CALL

T-39

```
conditional_entry_call ::=
  select
    entry_call_statement
    [ sequence_of_statements ]
  else
    sequence_of_statements
  end select;
```

Student Notes:

```
select
  BANK_TELLER.MAKE_DEPOSIT (20.00);
else
  GIVE_UP;
end SELECT;
```

T-40

TIMED ENTRY CALL

```
timed_entry_call ::=
  select
    entry_call_statement
    [ sequence_of_statements ]
  or
    delay_alternative
  end select;
```

```
select
  BANK_TELLER.MAKE_DEPOSIT (1_000.00);
or
  delay DURATION (10.0 * MINUTES);
  TAKE_A_HIKE;
end select;
```

T-42

Student Notes:

T-43

ABORT STATEMENT

- abort_statement ::= abort task_name { , task_name }
- Causes a task and all dependent tasks to become ABNORMAL thus preventing any further rendezvous with the task
- An abnormal task becomes completed in certain circumstances

  — accept statement
  — select statement
  — delay statement
  — entry call
  — activating

- Calling an ABNORMAL task or if a call has been made to an entry and is queued raises the exception TASKING_ERROR

T-44

ABORT STATEMENT

"An abort statement should be used only in extremely severe situations requiring unconditional termination"

STUDY GUIDE/WORKBOOK

E30AR4924 004
E40ST4924 020
90P 893

Technical Training

FUNDAMENTALS OF Ada PROGRAMMING/ SOFTWARE ENGINEERING

DECEMBER 1987

USAF TECHNICAL TRAINING SCHOOL
3390th Technical Training Group
Keesler Air Force Base, Mississippi

Designed For ATC Course Use

DO NOT USE ON THE JOB

Comm-Comp Systems Advanced Training Division
Keesler Air Force Base, Mississippi 39534-5000

HO E30AR4924 004
E40ST4924 020
1 Oct 1987

Philosophy

The philosophy of the wing emerges from a deep concern for individual Air Force men and women and the need to provide highly trained and motivated personnel to sustain the mission of the Air Force. We believe the abilities, worth, self-respect, and dignity of each student must be fully recognized. We believe each must be provided the opportunity to pursue and master an occupational specialty to the full extent of the individual's capabilities and aspirations, for the immediate and continuing benefit of the individual, the Air Force, DoD and the country. To these ends, we provide opportunities for individual development of initial technical proficiencies, on-the-job training in challenging job assignments, and follow-on growth as supervisors. In support of this individual development, and to facilitate maximum growth of its students, the wing encourages and supports the professional development of its faculty and administrators, and actively promotes innovation through research and the sharing of concepts and material with other educational institutions.

CONTENTS

Chapter 1

ORIENTATION

WELCOME

Welcome to the Fundamentals of Ada Programming/Software Engineering course. This class will give you knowledge of the fundamentals of engineering good Ada systems. It's a challenging class with time split between lecture and hands on exercises. It is our intention to make this course as informative and interesting as possible; however, we cannot accomplish this without your assistance. You are encouraged to participate in discussions and contribute as much as possible to enhance your learning and make the course more meaningful and enjoyable.

In this chapter, we will cover the student critique program, energy conservation, fraud, waste and abuse, administrative policies, and a course overview.

STUDENT CRITIQUE PROGRAM

To critique something is to express your opinion about the subject. The Student Critique Program exists for all ATC MTT Courses and at all Technical Training Centers because we are interested in your welfare and the effectiveness of our training. The purpose of the program is based upon the assumption that whatever bothers or distracts you will adversely affect your learning.

Although critiques are administered at the end of the course, you may critique training at any time during this course of instruction. Critique forms (ATC Form 736) are readily accessible in every classroom. Should you recognize a problem or a deficiency, do not hesitate to critique it. Likewise, you may submit critiques recognizing outstanding units of instruction, instructors, facilities, equipment, etc. We do ask you to critique training and facilities on a separate form. Your critique will be given careful consideration; it will provide us with valuable ideas which may improve training, as well as facilities and services.

Your sincere cooperation in the Critique Program can be beneficial to all students that follow you.

All critiques can be submitted without fear of reprisal or prejudice.

FRAUD, WASTE AND ABUSE (FW&A)

The Air Force policy on fraud, waste and abuse is to use all available means to prevent, detect, correct and discipline, as warranted, perpetrators involved in FW&A.

Definitions

1. FRAUD: Intentional misleading or deceitful conduct that deprives the Government of its resources or rights.

2. WASTE: Extravagant, careless or needless expenditures of Government resources from improper or deficient practices, systems, controls or decisions.

3. ABUSE: Intentional wrongful or improper use of Government

resources, i.e. misuse of rank, position or authority.

Any Person who knows of fraud, waste or abuse has a duty to report it to his or her supervisor, commander, inspector general, Air Force Audit Agency (AFAA), AFOSI, the security police or other proper authority. Each member of the Air Force, military or civilian, has the right to file a disclosure without fear of reprisal. The following are examples of FW&A that students should avoid:

1. Abusing equipment, whether intentional or not.

2. Wrongful destruction of student literature.

3. Willful waste of janitorial supplies.

4. Facilities abuse.

5. Unauthorized use of Government telephone services.

6. Intentional lack of personal commitment in doing a duty or task for which a salary is being paid.

7. Intentional practice to avoid making corrections to known deficiencies in order to prevent fraud, waste and abuse.

8. Waste/unauthorized distribution of Government supplies.

ADMINISTRATIVE POLICIES

INSTRUCTOR: ______________________

Duty hours are ______ to ______ during this course. Ten minute breaks are provided each hour with one hour for lunch.

You are asked to reschedule any appointments during the length of this course. If an appointment cannot be rescheduled, inform the instructor as soon as possible. If you miss a portion of a class it is your responsibility to make arrangements with the instructor to find out what material was missed and how it can be made up. If for any reason you miss more than 10 percent of the class time you can be removed from training and asked to reschedule.

A class leader will be appointed by the instructor during the first hour of class. The class leader acts as your representative and is tasked with the following responsibilities:

1. Assist the instructor in maintaining order at all times during the class period.

2. Supervise classroom clean-up.

3. Assume control of the class in the absence of the instructor, or as directed.

4. Act as spokesman for the class in any matter which the class members deem necessary, usually matters which require supervisory attention.

5. Encourage military students in the class to maintain high standards IAW AFR 35-10.

Facilities Available

Room _____ Break Area

Room _____ Female Latrine

Room _____ Male Latrine

Room _____ Administration Offices

Phone Number ____________________

COURSE OVERVIEW

Unit 1: Introduction

Unit 2: Training Evaluation Feedback System

Unit 3: Fundamentals of Ada Systems

Unit 4: Basic Ada Types

Unit 5: Control Structures

Unit 6: Subprograms

Unit 7: Packages

Unit 8: Exceptions

Unit 9: Generics

Unit 10: Tasks

Unit 11: Program Design Using Ada

Unit 12: Develop Software Using Ada

CHAPTER 2

THE TRAINING EVALUATION FEEDBACK SYSTEM

OBJECTIVE

Using the student handout as a reference, briefly describe the purpose of the training evaluation program.

INTRODUCTION

The training evaluation feedback system is a useful tool to keep our courses up to date with the requirements of the Air Force.

INFORMATION

PURPOSE OF EVALUATION

The purpose of the training evaluation program is to obtain the information necessary to determine the:

1. Ability of graduates to perform their assigned task to the level of proficiency specified in the applicable training standard.

2. Extent to which skills acquired in training are used by graduates in the field.

3. Extent to which knowledge attained in training is retained by graduates in the field.

4. Need for revisions in the training standards and courses to improve training effectiveness and responsiveness to the needs of the using commands.

The evaluation includes the collection, collation, analysis, and interpretation of feedback information to assess the effectiveness of training and the extent to which course graduates satisfy field performance requirements.

RESPONSIBILITIES

Commands conducting formal courses are required to conduct evaluations to determine the adequacy and relevance of training and to make revisions as needed.

Using commands are required to participate in the evaluation program by furnishing information to representatives of training activities during:

1. Field visits

2. Completing and returning field survey questionnaires

3. Completing Training Quality Reports to identify training deficiencies and recommending changes to training standard tasks, knowledge or proficiency levels that are not meeting command requirements.

SUMMARY

The program provides a means whereby supervisors and graduates can help training activities develop and conduct training programs that are best suited to their needs.

Chapter 3

FUNDAMENTALS OF Ada SYSTEMS

OBJECTIVE

Given a simple program specification, student instructional materials, and student notes, engineer a program in Ada that correctly implements the problem. Program must conform to course software engineering standards. Instructor may provide up to 4 assists.

INTRODUCTION

You may have heard the claim that "Ada is just another programming language." Well, that depends on your point of view. Any programming language is a tool to transform a software design into the actual machine language instructions that a computer performs. In that respect, Ada is another computer language, just as a hand shovel and diesel powered shovel are both tools to dig a basement. However, when digging a basement, you should choose the tool that best supports the job. When developing software, you should choose the tool that best supports the goals and principles of software engineering.

INFORMATION

SOFTWARE ENGINEERING

What is software engineering? For the purposes of this course, we view it as an orderly application of tools to develop software that is reliable, maintainable, efficient, and understandable. Using this definition, a programming language is just one of a number of tools that is used when called for in applying some methodology to develop software.

We can identify a number of principles to keep in mind while developing software that supports the goals of reliable, maintainable, efficient, and understandable software. These principles are:

- o Abstraction - Considering only the important features at this level and ignoring the unimportant details.
- o Information Hiding - Making underlying details inaccessible.
- o Modularity - Breaking up a large system into manageable pieces.
- o Localization - Physically grouping together logically related entities.
- o Completeness - Ensuring that all required features are present.
- o Confirmability - Ensuring that the system can be tested to make sure it's complete and meets the requirements.
- o Uniformity - Ensuring that there are no unnecessary differences in notation that can be confusing.

Throughout the course we will relate the features of Ada to these goals and principles of software engineering.

Ada LANGUAGE FEATURES

Anyone who has looked through

the Ada reference manual can tell you that Ada is a complex language. The features of Ada are integrated--in other words, to write even a simple Ada program, you need at least a shallow knowledge of a number of language features.

Data Typing

One of these features is Ada's use of strong data typing. Strong typing means that every object (variables and constants are objects) has to be associated with some type. This type defines the set of values and the set of operations for that object. Ada also doesn't allow you to mix apples and oranges; if you have two objects of different types, you can't implicitly mix them in an operation. In other languages, you may declare an object to be an integer, character, etc. This is similar to declaring an object in Ada, although Ada takes this one step further in that it allows you to declare your own distinct types. This helps you model the data structure of the real world problem.

Program Units

Program units are structures used to break up large software systems into smaller, more manageable parts. An Ada program should therefore use program units to break the code up into easily understandable segments. Each program unit has two parts: a specification and a body. The specification is the logical view of this program unit which defines the interface to other program units (Abstraction). The body defines how the details of the program unit are implemented. These details that are in the body are inaccessible to other program units (Information Hiding). The separation of specification and body allow you to view these program units more like black boxes with the communication requirements specified by the specification.

There are four different kinds of program units that we can use to break up our system. These are:

- o Subprograms - Program units that perform an operation or calculation.
- o Packages - Program units that allow you to group together logically related entities.
- o Generics - Program units that generalize subprograms or packages.
- o Tasks - Program units that run in parallel with other program units.

In class, we'll explore each of these different program units.

Program Structure

An Ada program is simply a main subprogram. The body to this main subprogram contains two parts: the declarative part and the executable part. The declarative part is where we declare our types, objects, or even other program units. The executable part contains the statements to be performed during execution. A simple Ada program would then look like this:

```
procedure MAIN is

  -- Declarative Part

begin
  null;     -- Executable Part
end MAIN;
```

The declarative and executable parts are separated by the word 'begin'.

Everything to the right of a double dash '--' would be a comment. We'll add more to this basic structure later in class.

PROGRAM LIBRARY

The concept of a program library is very important in Ada. The program library is simply a collection of information on all compilation units, or program parts, that have been compiled into a library. This is important in Ada because the language allows you to separately compile parts of the system that are in different files. When compiling a program unit, the compiler has access to a record of everything else that has been compiled up to this point. This powerful feature enables the compiler to enforce its visibility and strong typing rules across program unit boundaries.

SIMPLE CONTROL STRUCTURES

Ada has a number of control structures. During this lecture, we'll cover some simple control structures such as assignment, if, and loop statements. These will allow you to begin writing simple Ada programs.

SIMPLE INPUT/OUTPUT

When looking at the Ada language, you have to keep in mind the application for which it was designed: embedded computer systems. These are systems where the computer is only a small part that controls the rest of the system, such as the computer that controls the ignition system in your car. These embedded systems typically have small memory space and unique input/output requirements. Because of these requirements, the language designers chose not to make textual input/output intrinsic to Ada. This way, those embedded systems programs that don't require input or output of text don't have to suffer with the overhead of these routines.

For programs that need text input/output, there is a predefined package in the Ada program library that contains a set of input/output routines. The routines in this package TEXT_IO are only accessible to those programs that explicitly tie into this package. In class, you'll see how we do this.

SUMMARY

Ada is a language designed for engineering software systems. It directly supports the goals and principles identified for software engineering. It's a complex language with many integrated features—features that you will find very useful by the end of the course.

EXERCISE 3-1

Note: Some of the material needed to answer the following questions is only covered in the lecture.

1. What is "abstraction"?

2. What is "information hiding"?

3. Name the two parts of a program unit.

4. Types and objects are declared in the ________________ part of a program unit body.

5. What is a package?

6. How do generic program units aid in reusability?

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                      %
--%  Abstract:  This program computes the area of a triangle.  It first %
--%             prompts you for the length of the base and the height of %
--%             the triangle, then prints out the area.                  %
--%                                                                      %
--%  Author:    John Doe                                                 %
--%                                                                      %
--%  Date:      19 Oct 87                                                %
--%                                                                      %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

with TEXT_IO;
procedure COMPUTE_AREA_OF_TRIANGLE is

   MAXIMUM_LENGTH : constant := 50;
   ONE_HALF : constant := 0.5;
   MAXIMUM_AREA : constant := ONE_HALF *
                              (MAXIMUM_LENGTH * MAXIMUM_LENGTH);

--

   type LENGTH_TYPE is range 0 .. MAXIMUM_LENGTH;

   type AREA_TYPE is digits 10 range 0.0 .. MAXIMUM_AREA;

--

   LENGTH_OF_BASE : LENGTH_TYPE := 0;
   HEIGHT         : LENGTH_TYPE := 0;
   AREA           : AREA_TYPE   := 0.0;
   ANSWER         : CHARACTER   := 'Y';

--

   package AREA_IO   is new TEXT_IO.FLOAT_IO( AREA_TYPE );
   package LENGTH_IO is new TEXT_IO.INTEGER_IO( LENGTH_TYPE );

--                    (Continued on next page)
```

```
begin  -- COMPUTE_AREA_OF_TRIANGLE

   loop

      TEXT_IO.PUT_LINE("This program calculates the area of a");
      TEXT_IO.PUT_LINE(" triangle given the length of its base")·
      TEXT_IO.PUT_LINE(" and its height.");
      TEXT_IO.NEW_LINE(2);

      TEXT_IO.PUT("ENTER THE LENGTH OF THE BASE (MUST BE AN INTEGER BETWEEN");
      LENGTH_IO.PUT( LENGTH_TYPE'FIRST);
      TEXT_IO.PUT(" AND");
      LENGTH_IO.PUT( LENGTH_TYPE'LAST );
      TEXT_IO.PUT(": ");
      LENGTH_IO.GET( LENGTH_OF_BASE );

      TEXT_IO.PUT("ENTER THE HEIGHT (MUST BE AN INTEGER BETWEEN");
      LENGTH_IO.PUT( LENGTH_TYPE'FIRST);
      TEXT_IO.PUT(" AND");
      LENGTH_IO.PUT( LENGTH_TYPE'LAST );
      TEXT_IO.PUT(": ");
      LENGTH_IO.GET( HEIGHT );

      AREA := ONE_HALF * AREA_TYPE(LENGTH_OF_BASE) * AREA_TYPE(HEIGHT);

      TEXT_IO.NEW_LINE;
      TEXT_IO.PUT("THE AREA OF THE TRIANGLE IS: ");
      AREA_IO.PUT( AREA );

      TEXT_IO.NEW_LINE;
      TEXT_IO.PUT("DO YOU WANT TO TRY ANOTHER? (Y OR N)");
      TEXT_IO.GET( ANSWER );

      exit when ANSWER = 'N' or ANSWER = 'n';

   end loop;

end COMPUTE_AREA_OF_TRIANGLE;
```

EXERCISE 3-2

1. Log on to the computer with your correct user name.
2. Enter the program below using the editor.
3. Compile using the Ada compiler.
4. Make any corrections needed to fix errors.
5. Run your program.
6. When you have finished, call the instructor to evaluate your program.

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                    %
--%  Abstract:  This program computes all of the prime numbers up to some %
--%             value MAXIMUM_NUMBERS.                                 %
--%                                                                    %
--%  Author:    John Doe                                               %
--%                                                                    %
--%  Date:      18 Sep 87                                              %
--%                                                                    %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

with TEXT_IO;
procedure SIEVE is
   MAXIMUM_NUMBERS : constant := 500;
--
   type NUMBER_TYPE is range 1 .. (MAXIMUM_NUMBERS + MAXIMUM_NUMBERS/2);
   subtype PRIME_RANGE is NUMBER_TYPE range 1..MAXIMUM_NUMBER;
   type BOOLEAN_ARRAY_TYPE is array (PRIME_RANGE) of BOOLEAN;
--
   NUMBER    : NUMBER_TYPE := NUMBER_TYPE'FIRST;
   PRIMES    : BOOLEAN_ARRAY_TYPE := (others => TRUE);
--
   package INT_IO is new TEXT_IO.INTEGER_IO (NUMBER_TYPE);

begin
   for COUNTER in NUMBER_TYPE range 2..PRIMES'LAST / 2 loop
      NUMBER := COUNTER + COUNTER;
      while NUMBER <= PRIMES'LAST and PRIMES (COUNTER) loop
         PRIMES (NUMBER) := FALSE;
         NUMBER := NUMBER + COUNTER;
      end loop;                  -- while NUM
   end loop;                     -- for COUNTER

   TEXT_IO.PUT ("The prime numbers from 1 to ");
   INT_IO.PUT (PRIMES'LAST);
   TEXT_IO.PUT_LINE (" are:");
   for INDEX in PRIMES'RANGE loop
      if PRIMES (INDEX) then
         INT_IO.PUT (INDEX);
      end if;
   end loop;                     -- for INDEX
   TEXT_IO.NEW_LINE;
end SIEVE;
```

Chapter 4

BASIC Ada TYPES

OBJECTIVE

Given a simple program specification, an incomplete Ada program, student instructional materials, and student notes, add the correct types-objects to the program to correctly implement the problem. Program must conform to course software engineering standards. Instructor may provide up to 3 assists.

INTRODUCTION

What purpose does it serve to say that a variable is of a certain type, or class of objects? First, it tells the compiler how to treat a series of bits. For example, adding two integers is different from adding two floating point numbers.

But data typing helps software engineers also. It allows them to assign logical properties to an object to model the real world. This data abstraction can greatly increase the understandability of a program. For example, if we need to keep track of the days of the week, it is much more understandable to refer to the days as MONDAY, TUESDAY, WEDNESDAY, etc. as opposed to numbers from 1 to 7 (or was it 0 to 6?).

INFORMATION

STRONG TYPING

In the last chapter, we defined strong typing to mean that every object has an associated type; this type defines the set of values and set of operations available for objects of that type. Also, we can't implicitly mix objects of different types. Ada supplies some predefined types such as INTEGER, CHARACTER, and FLOAT, but more importantly, it gives us the capability to declare new data types to model our own abstractions.

Type Declarations

Type and object declarations can be declared in the declarative part of any program unit:

```
procedure MAIN is
  -- Declarative part

  type AIRCRAFT is (B1,B52,F16);

begin
  null;
end MAIN;
```

In the above example, the type declaration begins with:

```
type AIRCRAFT is ....
```

ALL new type declarations begin this way, with the word 'type' followed by the type name followed by the word 'is'. Whatever comes after the word 'is' defines what class of type we are declaring.

Object Declarations

The above type declaration only defines the characteristics for objects of that type--the set of values (B1, B52, F16) and a set of operations. In order to get any use from a type declaration, we need to declare an object of that type:

```
procedure MAIN is
  -- Declarative part

  type AIRCRAFT is (B1,B52,F16);
  PLANE : AIRCRAFT;

begin
  null;
end MAIN;
```

Now we have an object called PLANE whose value can be B1, B52, or F16. The operations we can perform on PLANE are those available for the class of enumeration types which we'll discuss in class.

CLASSES OF TYPES

The types we can declare in Ada fall into one of the following classes: scalar, composite, access, private, or task.

Scalar Types

The objects of a scalar type only contain one value at a time. Type AIRCRAFT in the previous example is a scalar type because at any point in time PLANE can contain only one of the values B1, B52 or F16.

We can break the class of scalar types into integer, enumeration, floating point, and fixed point types, which we'll cover during class.

Composite Types

Unlike objects of a scalar type that can only contain one value, objects of a composite type can contain collections of values. Composite types can be arrays, where all of the components in the collection are of the same type, and records, where the components can be of different types.

Other Types

The other kinds of types we can declare in Ada are:

- o Access Types: The objects are pointers to other objects.
- o Private Types: The operations on objects of the type are only those that are explicitly stated.
- o Task Types: The objects define a parallel process.

SUMMARY

The strong typing rules in Ada require that every object be associated with a type. The type defines the set of values and the set of operations available to the objects of the type. Ada allows you to declare your own types to set up abstractions of the real world and make the solution more understandable.

EXERCISE 4-1

Note: Some of the material needed to answer the following questions is only covered in the lecture.

1. Define strong typing.

2. The two kinds of composite types are ________ and ________.

3. A type defines a set of ________ and a set of __________.

4. What is the difference between a constrained and an unconstrained array?

5. An array is a collection of ____________ objects while a record is a collection of ____________ objects.

EXERCISE 4-2

1. Add the type and object declarations that are called for in the following program shell:

```
procedure BUY_A_USED_CAR is

  -- Declare an integer type called YEARS that ranges from 1900 to 2500.


  - Declare a type named CAR_MAKES that contains the values DODGE, FORD,
    PONTIAC, PLYMOUTH, MERCURY, and CHEVY.


  -- Declare a type named COLORS that has the values RED, SILVER, BLUE, BLACK,
     and YELLOW.


  -- Declare a record type called CARS with the following components: YEAR of
     type YEARS, COLOR of type COLORS, and MAKE of type CAR_MAKES.


  -- Declare an array type named USED_CAR_LOTS that can contain 50 elements of
     type CARS.


  -- Declare an object named DANS_USED_CARS of the type USED_CAR_LOTS.


begin
  null;
end BUY_A_USED_CAR;
```

EXAMPLE 4-1

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                   %
--%  Abstract:  This program computes the average of student's test  %
--%             scores for an entire class.  Each student has three test  %
--%             scores and the number of students is given by the number  %
--%             declaration.  The average is the total average of all  %
--%             tests by all students.                                %
--%                                                                   %
--%  Author:    Max Programmer                                        %
--%                                                                   %
--%  Date:      19 Oct 87                                             %
--%                                                                   %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

with TEXT_IO;
procedure AVERAGE_SCORES is

   MAX_SCORE          : constant := 100.0;
   NUMBER_OF_STUDENTS : constant := 10;
   NUMBER_OF_TESTS    : constant := 3.0;

--

   type TEST_SCORE_TYPE is digits 5 range 0.0 .. MAX_SCORE;

   type STUDENT_TYPE is range 1 .. NUMBER_OF_STUDENTS;

   type SCORES_RECORD is
     record
       FIRST_TEST  : TEST_SCORE_TYPE;
       SECOND_TEST : TEST_SCORE_TYPE;
       THIRD_TEST  : TEST_SCORE_TYPE;
     end record;

   type SCORE_LIST_TYPE is array( STUDENT_TYPE ) of SCORES_RECORD;

--

   SCORES : SCORE_LIST_TYPE := ( others => ( 0.0, 0.0, 0.0 ) );

   INDIVIDUAL_AVERAGE,
   TOTAL_AVERAGE : TEST_SCORE_TYPE := 0.0;

--

   package STUDENT_IO is new TEXT_IO.INTEGER_IO( STUDENT_TYPE);
   package SCORE_IO is new TEXT_IO.FLOAT_IO( TEST_SCORE_TYPE );

--
```

(Continued on next page)

```
begin  -- AVERAGE_SCORES

   for STUDENT in SCORES'RANGE loop                      -- Get test data

      TEXT_IO.PUT("STUDENT NUMBER: ");
      STUDENT_IO.PUT( STUDENT );
      TEXT_IO.NEW_LINE(2);
      TEXT_IO.PUT("FIRST TEST SCORE: ");
      SCORE_IO.GET( SCORES(STUDENT).FIRST_TEST );
      TEXT_IO.PUT("SECOND TEST SCORE: ");
      SCORE_IO.GET( SCORES(STUDENT).SECOND_TEST );
      TEXT_IO.PUT("THIRD TEST SCORE: ");
      SCORE_IO.GET( SCORES(STUDENT).THIRD_TEST );
      TEXT_IO.NEW_LINE(3);

   end loop;

   for STUDENT in SCORES'RANGE loop                      -- Compute average

      INDIVIDUAL_AVERAGE := SCORES(STUDENT).FIRST_TEST/NUMBER_OF_TESTS +
           SCORES(STUDENT).SECOND_TEST/NUMBER_OF_TESTS +
           SCORES(STUDENT).THIRD_TEST/NUMBER_OF_TESTS;
      TOTAL_AVERAGE := TOTAL_AVERAGE + INDIVIDUAL_AVERAGE/
           TEST_SCORE_TYPE(SCORES'LENGTH);
      TEXT_IO.PUT("STUDENT NUMBER: ");
      STUDENT_IO.PUT( STUDENT );
      TEXT_IO.PUT("AVERAGE IS: ");
      SCORE_IO.PUT(INDIVIDUAL_AVERAGE);
      TEXT_IO.NEW_LINE(2);

   end loop;

   TEXT_IO.PUT("CLASS AVERAGE IS ");                     -- Print average
   SCORE_IO.PUT( TOTAL_AVERAGE );
   TEXT_IO.NEW_LINE;

end AVERAGE_SCORES;
```

Chapter 5

CONTROL STRUCTURES

OBJECTIVE

Given a program specification, an incomplete Ada program, student instructional materials, and student notes, use the appropriate control structures to correctly implement the problem. Program must conform to course software engineering standards. Instructor may provide up to 2 assists.

INTRODUCTION

Control structures , or statements, define the flow of control in the executable part of our program units. These define the steps the program unit goes through to get its job done. Of all the control structures available, we can break them up into three general categories: sequential, conditional, and iterative.

INFORMATION

SEQUENTIAL CONTROL STRUCTURES

Sequential statements are performed one after another. Three sequential statements that we'll talk about in class are the assignment, null, and block statements.

The assignment statement simply assigns a value to an object. It sounds simple, but there's a catch: the objects on both sides of the assignment statement have to be the same type. (Remember strong typing?)

The null statement does nothing. Just like a page that says "THIS PAGE INTENTIONALLY LEFT BLANK", a null statement can add to the readability of a program. It's useful in structures such as a case statement where you want to do nothing for a specific path of control.

The block statement is much more exciting. It allows us to localize declarations, kind of like creating a little declarative part within the sequence of statements of our executable part. The block statement also lets us localize the handling of certain conditions that occur during the program's execution, as we'll see in a later unit on exceptions.

CONDITIONAL CONTROL STRUCTURES

There are two kinds of conditional control structures: the if statement and the case statement. Both of these statements branch to a sequence of statements based on the value of some condition.

The if statement branches on a boolean (TRUE or FALSE) condition. If the condition is true, the enclosed statements will be executed.

```
if STOP_LIGHT = RED then
  STOP;
  WAIT;
  GO;
end if;
```

The if statement can also have an 'else' and/or 'elsif' part to further define the flow of control, as we'll see in class.

The case statement branches based upon the value of some discrete object. Instead of having just two alternatives, as with the if statement, the case statement can branch to a number of places based on the value of that discrete object.

```
type LIGHT is (RED, YELLOW, GREEN);
STOP_LIGHT : LIGHT := GREEN;

begin
  ...
  case STOP_LIGHT is
    when GREEN =>
      KEEP_GOING;
      CHECK_GAS;
    when RED =>
      STOP;
      WAIT;
      GO;
    when YELLOW =>
      GO_FASTER;
  end case;
```

ITERATIVE CONTROL STRUCTURES

Iterative control structures, or loops, are all based on one structure in Ada: the basic loop. This loop is structured to loop forever.

```
loop
  DO_SOMETHING;
  DO_SOMETHING_ELSE;
  --If you want to exit from the loop:
  exit;
end loop;
```

We can exit from this basic loop only through an exit statement, as shown above.

We can change the characteristics of the loop by adding an iteration scheme. A 'for' loop goes through the loop once for every value in a given range.

```
for INDEX in 1..10 loop
  PUT_LINE("Hello!");
end loop;
```

This for loop would print "Hello!" ten times. A 'for' loop should be used whenever you know how many times you want to go through the loop.

A 'while' loop lets you go through the loop while some condition is true.

```
while STATUS = RUNNING loop
  CHECK_TEMPERATURE;
  CHECK_FUEL_FLOW;
end loop;
```

This loop would execute until STATUS is no longer equal to RUNNING.

SUMMARY

Ada, like other languages, provides the three classical kinds of control structures: sequential, conditional, and iterative. With these classes of statements, all algorithms can be written.

EXERCISE 5-1

1. Complete the following subprogram.

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                    %
--%  Abstract:  This exercise requires you to enter the code to perform %
--%             the actions described in the comments below.           %
--%                                                                    %
--%  Author:    Max Programmer                                         %
--%                                                                    %
--%  Date:      19 Oct 87                                              %
--%                                                                    %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

with TEXT_IO;
procedure CONTROL_YOURSELF is

  MAX_NUMBERS : constant := 500;

--

  type NUMBERS is range 0..MAX_NUMBERS;

  type COLORS is ( RED, WHITE, BLUE, GREEN );

  type ARRAY_TYPE is array ( COLORS ) of NUMBERS;

--

  MY_ARR : ARRAY_TYPE := ( 2,46,12,38 );
  TOTAL  : NUMBERS := 0;

--

  package NUM_IO is new TEXT_IO.INTEGER_IO( NUMBERS );

begin  -- CONTROL_YOURSELF

  -- Add the components of the array together and put the result in TOTAL

  -- If TOTAL is between 1 and 50 then add 5 to the TOTAL
  -- If TOTAL is between 51 and 200 then add 10 to the TOTAL
  -- If TOTAL is between 250 and 300 then subtract 10 from the TOTAL
  -- If TOTAL is anything else then set TOTAL to zero

  -- Print out the result

end CONTROL_YOURSELF;
```

Chapter 6

SUBPROGRAMS

OBJECTIVE

Given a program specification, student instructional materials, and student notes, use subprograms to correctly implement the problem. Program must conform to course software engineering standards. Instructor may provide up to 3 assists.

INTRODUCTION

Subprograms are the primary means of defining abstract actions that take place in our system. For example, when we call the addition routine to add two numbers, we don't concern ourselves with the steps that take place to add the numbers--only that the result is correct. The same applies to routines that we design: someone who uses that routine can concentrate on what the function does rather than how it works.

Subprograms aid our design effort in that we can break up the large system into smaller, more understandable pieces and use subprograms to implement some of the pieces.

INFORMATION

There are two forms of subprograms in Ada: procedures and functions. Procedures are used to invoke some action, while functions are used to compute a value.

STRUCTURE

Both procedures and functions have two parts: the specification which tells WHAT the subprogram does, and the body that tells HOW the subprogram is implemented.

Specification:

```
procedure PRINT_MY_NAME;
```

Body:

```
with TEXT_IO;  -- To gain access to
               -- Input/Output routines
procedure PRINT_MY_NAME is
  -- Declarative Part
begin
  -- Executable Part
  TEXT_IO.PUT("Joe");
end PRINT_MY_NAME;
```

As shown above, the body is also divided into two parts: the declarative part where we can declare local types, variables, or program units; and the executable part where we define the steps to be executed when the procedure is called.

If we had a program that needed this routine, we could call it from that program:

```
with PRINT_MY_NAME;
procedure MAIN is

begin  -- MAIN
  PRINT_MY_NAME;   -- Call to procedure
end PRINT_MY_NAME;
```

PARAMETERS

With the previous procedure, we couldn't tell it which name to print--it would always print "Joe". In order to communicate with this procedure, we need to set up parameters to pass data to it:

```
procedure PRINT_MY_NAME(NAME : in STRING);
```

Now when we call the procedure, we must also pass a value to it of type STRING to match this parameter NAME. The body would look like:

```
with TEXT_IO;
procedure PRINT_MY_NAME(NAME : in
                               STRING) is
begin
  TEXT_IO.PUT(NAME);
end PRINT_MY_NAME;
```

To call this procedure from our main program:

```
with PRINT_MY_NAME;
procedure MAIN is

begin  -- MAIN
  PRINT_MY_NAME("Joe");
  PRINT_MY_NAME("Sally");
end MAIN;
```

In class you'll see the different modes allowed for parameters to pass data into or out from a subprogram.

FUNCTIONS

As we said earlier, a procedure performs some abstract action, while a function computes a value. We reflect this in the syntax by adding a 'return' clause to the end of the specification indicating the type of value returned:

```
function DOUBLE (NUMBER : INTEGER)
                          return INTEGER;
```

When we call this function, it will return the computed value to the point where it was called. Therefore, we can only call a function as part of an expression. We can keep track of the result by assigning it to a variable:

```
with DOUBLE;
procedure DOUBLE_IT is

  RESULT : INTEGER;

begin  -- DOUBLE_IT
  RESULT := DOUBLE(5);
          -- RESULT has a value of 10.
end DOUBLE_IT;
```

SUMMARY

Subprograms provide a tool for defining functional abstractions of our system. Like other program units, we can separate the WHAT (Specification) from the HOW (Body). Ada gives us two forms of subprograms, procedures and functions to represent actions or calculations.

EXERCISE 6-1

Note: Some of the material needed to answer the following questions is only covered in the lecture.

1. What are the three modes for procedure parameters?

2. The only mode allowed for function parameters is ________.

3. Define:
   a. Actual Parameters

   b. Formal Parameters

4. How do subprograms support abstraction?

5. How do subprograms support modularity?

EXERCISE 6-2

1. Rewrite the subprogram below into one main subprogram and three embedded subprograms. Use subunits to place the subprograms in a separate file.

2. The main subprogram will simply call the first subprogram to prompt for and get the name. The second subprogram will count the number of 'S's in the name. The third subprogram will echo the name back to the user.

3. In the body of the main program, call the subprograms in the order listed above to get a name, count the number of 'S's, and print echo back the name and number of 'S's.

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                      %
--%  Abstract:  This program reads in a name, counts the number of upper %
--%             case 'S's, and echos back the name as well as the number %
--%             of 'S's in the name.                                     %
--%                                                                      %
--%  Author:    Sleepy                                                   %
--%                                                                      %
--%  Date:      14 Jan 76                                                %
--%                                                                      %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

with TEXT_IO;
procedure ECHO_NAME is

     MAX_LENGTH : constant := 80;

--

     subtype LINE_TYPE is STRING(1..MAX_LENGTH);

     type NAME_TYPE is record
          CHARACTERS : LINE_TYPE;
          LENGTH     : NATURAL;
     end record;

--

     -- Make your declarations here

begin

     -- Make your subprogram calls here

end ECHO_NAME;
```

EXAMPLE 6-1

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                      %
--%   Abstract:  This program implements the famous Humpty Dumpty        %
--%              algorithm.                                              %
--%                                                                      %
--%   Author:    Sneezy                                                  %
--%                                                                      %
--%   Date:      1 Sep 80                                                %
--%                                                                      %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure MOTHER_GOOSE is

  MAX_EGG_HEADEDNESS : constant := 10;
  MAX_MEN : constant := 50;
  MAX_HORSES : constant := 50;

--

  type EGG_HEAD is range 1..MAX_EGG_HEADDEDNESS;  -- Degree of eggheadedness

  type MEN      is range 1..MAX_MEN;

  type HORSES   is range 1..MAX_HORSES;

--

  HUMPTY_DUMPTY     : EGG_HEAD := 7;
  ALL_KINGS_MEN     : MEN      := 15;
  ALL_KINGS_HORSES  : HORSES   := 15;

--
  procedure SAT_ON_WALL ( PERSON : in out EGG_HEAD );

  procedure HAD_GREAT_FALL ( PERSON : in out EGG_HEAD );

  procedure GET_OFF_WALL ( PERSON : in out EGG_HEAD );

  function  CAN_PUT_TOGETHER_AGAIN ( PERSON      : in EGG_HEAD;
                                     HOW_MANY    : in MEN;
                                     HOW_MANY    : in HORSES )
                                     return BOOLEAN;

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                      %
--%   Abstract:  This subprogram sits a person on a wall.                %
--%                                                                      %
--%   Author:    Sneezy                                                  %
--%                                                                      %
--%   Date:      1 Sep 80                                                %
--%                                                                      %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure SAT_ON_WALL ( PERSON : in out EGG_HEAD ) is

begin  -- SAT_ON_WALL
  .
  .
  .
end SAT_ON_WALL;
```

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                 %
--%  Abstract:  This subprogram gives you a great fall.             %
--%                                                                 %
--%  Author:    Dopey                                               %
--%                                                                 %
--%  Date:      1 Sep 80                                            %
--%                                                                 %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure HAD_GREAT_FALL ( PERSON : in out EGG_HEAD ) is

begin -- HAD_GREAT_FALL
   .
   .
   .
end HAD_GREAT_FALL;

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                 %
--%  Abstract:  This subprogram gets a person off of the wall.      %
--%                                                                 %
--%  Author:    Sleepy                                              %
--%                                                                 %
--%  Date:      2 Sep 80                                            %
--%                                                                 %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure GET_OFF_WALL ( PERSON : in out EGG_HEAD ) is separate;

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                 %
--%  Abstract:  This function determines if a person can be put back %
--%             together, given a number of MEN and HORSES.         %
--%                                                                 %
--%  Author:    Sneezy                                              %
--%                                                                 %
--%  Date:      1 Sep 80                                            %
--%                                                                 %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function  CAN_PUT_TOGETHER_AGAIN ( PERSON     : in EGG_HEAD;
                                   HOW_MANY   : in MEN;
                                   HOW_MANY   : in HORSES )
                                   return BOOLEAN is separate;

--%%%%%%%%%%%%%%%%%%%%%%%%%% MOTHER_GOOSE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

begin  --  MOTHER_GOOSE

   SAT_ON_WALL    ( HUMPTY_DUMPTY );
   HAD_GREAT_FALL ( HUMPTY_DUMPTY );
   if CAN_PUT_TOGETHER_AGAIN ( HUMPTY_DUMPTY, ALL_KINGS_MEN,
                                     ALL_KINGS_HORSES ) then
      GET_OFF_WALL ( HUMPTY_DUMPTY );
   end if;

end MOTHER_GOOSE;
```

Chapter 7

PACKAGES

OBJECTIVE

Given a program specification, student instructional materials, and student notes, use packages to correctly implement the problem. Program must conform to course software engineering standards. Instructor may provide up to 3 assists.

INTRODUCTION

A package is one of the most powerful tools in the Ada language. It allows us to define a more meaningful structure to our software. A package is defined as a collection of logically related entities, such as types or subprograms. This tool allows us to directly implement principles of software engineering such as modularity, localization, abstraction and information hiding.

INFORMATION

A package is like the other program units in that it consists of a specification and a body. The specification gives the logical view of WHAT is in the package while the body defines HOW these features are implemented. This separation of specification and body (the WHAT from the HOW) is the key to engineering understandable and maintainable code.

SPECIFICATION

The specification of a package may be placed in its own file and compiled all by itself. Inside the specification we declare the types and program unit specifications of the entities we want to export to other programs.

```
package TELEVISION_SETS is

   type STATUS is (ON, OFF);
   type CHANNEL is range 2..80;

   type TV is
     record
       TV_STATUS  : STATUS  := OFF;
       TV_CHANNEL : CHANNEL := 2;
     end record;

   procedure TURN_ON(SET : in out TV);
   procedure TURN_OFF(SET : in out T');
   procedure SET_CHANNEL(SET : in out TV;
                         TO  : in CHANNEL);
   procedure NEXT_CHANNEL(SET : in out TV);

end TELEVISION_SETS;
```

In this package, we are modeling a television. Our TV is defined as having a TV_STATUS and a current TV_CHANNEL. What can we do with this TV? We can turn it on, turn it off, or change the channel. In this single package, we have defined our logical view of a television set. The package allows us to implement object abstraction by grouping the type TV and all of its operations together in one package.

We can now use this TV model in programs we may write:

```
with TELEVISION_SETS;
procedure LOOK_FOR_SPORTS is

  MY_SONY : TELEVISION_SETS.TV;

  procedure WATCH_TV is separate;
  function IS_SPORTS return BOOLEAN
                          is separate;

begin -- LOOK_FOR_SPORTS

  while not IS_SPORTS loop
    TELEVISION_SETS.NEXT_CHANNEL(MY_SONY);
  end loop;

  WATCH_TV;

end LOOK_FOR_SPORTS;
```

The first line of our program is a "context clause". It gives us access to everything that is declared in the package specification. Notice how everywhere we refer to anything out of the package, we preface it with the name of the package:

```
MY_SONY : TELEVISION_SETS.TV;
```

This helps out the maintenance programmer locate where type TV is located.

BODY

The specification gave us the logical view of what was in the package. The body defines the implementation details of what is declared in the package specification.

When a main program uses a package, it only has access to things declared in the package specification. Therefore, anything defined in the package body that isn't declared in the specification is hidden from the main program. This concept directly implements the principle of information hiding. This tends to make programs more modifiable because changes to the implementation details (body) won't affect other programs, as long as the interface (specification) remains the same.

```
package body TELEVISION_SETS is

  procedure TURN_ON(SET : in out TV) is

  begin -- TURN_ON
    SET.TV_STATUS := ON;
  end TURN_ON;
  ------------------------------------------------

  procedure TURN_OFF(SET : in out TV) is

  begin -- TURN_OFF
    SET.TV_STATUS := OFF;
  end TURN_OFF;
  ------------------------------------------------

  procedure SET_CHANNEL(SET : in out TV;
                        TO  : in CHANNEL) is

  begin -- SET_CHANNEL
    SET.TV_CHANNEL := TO;
  end SET_CHANNEL;
  ------------------------------------------------

  procedure NEXT_CHANNEL
                        (SET : in out TV) is

  begin -- NEXT-CHANNEL
    SET.TV_CHANNEL := SET.TV_CHANNEL + 1;
  end NEXT_CHANNEL;

end TELEVISION_SETS;
```

PRIVATE TYPES

If we look back at our package specification, you'll notice that programs that use this package have access to the details of type TV. In many cases we may not want this. To support the principle of information hiding, we would like the ability to hide the implementation of this type as well. Ada allows us to hide these details through the use of a private type:

```
package TELEVISION_SETS is

  type CHANNEL is range 2..80;

  type TV is limited private;

  procedure TURN_ON(SET : in out TV);
  procedure TURN_OFF(SET : in out TV);
  procedure SET_CHANNEL(SET : in out TV;
                        TO  : in CHANNEL);
  procedure NEXT_CHANNEL(SET : in out TV);

private

  type STATUS is (ON, OFF);

  type TV is
    record
      TV_STATUS  : STATUS  := OFF;
      TV_CHANNEL : CHANNEL := 2;
    end record;

end TELEVISION_SETS;
```

Now other programs can only access anything in the VISIBLE part of the package: that part before the word private. Between the word private and the end of the package specification is the private part where we define the full type declaration.

When another program uses this package, the only allowed operations for type TV are those also defined in the package specification: TURN_ON, TURN_OFF, SET_CHANNEL, NEXT_CHANNEL. He can't even assign one object of type TV to another. By making type TV private, we deny other program units the ability to access the components of a TV. Maybe these components may change. If so, other program units won't be affected because they are still forced to manipulate the TV only through the operations listed in the package specification.

Inside the package body however, the coder has full access to EVERYTHING defined in the specification, including private types. Therefore he can refer to the components of objects of type TV. In essence, our package body would remain the same as it was when type TV wasn't private.

SUMMARY

Packages are a very powerful tool that directly support many of the principles of software engineering. The specification provides the abstract view of the collection of resources, whole the body hides the details of their implementation. We can use the package to break up our software system into logically related, localized routines. This added feature allows us to define new types of abstraction, such as object abstraction, that aren't available in languages where a subprogram is the primary structuring tool.

EXERCISE 7-1

1. Modify the following program so that all the constant, type and subprogram declarations are in a separately compiled package. The main subprogram should declare the array object and make calls to the routines in the package.

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                      %
--%  Abstract:  This program gets a word from the keyboard, inverts the  %
--%             characters in the word, and prints it back out in its    %
--%             inverted form.                                           %
--%                                                                      %
--%  Author:    Joe Dynamite                                             %
--%                                                                      %
--%  Date:      4 Jul 85                                                 %
--%                                                                      %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

with TEXT_IO;
procedure INVERT_ARRAY is

  MAX_NUM : constant := 5;

--
  subtype CAPITALS is CHARACTER range 'A'..'Z';
  type NUM_ITEMS is range 1..MAX_NUM;
  type CAP_ARR is array ( NUM_ITEMS ) of CAPITALS;

--
  WORD : CAP_ARR;
--
  procedure GET_WORD ( NEW_WORD : out CAP_ARR ) is
  begin
    for INDEX in NEW_WORD'RANGE loop
       TEXT_IO.PUT( " INPUT A CAPITAL LETTER " );
       TEXT_IO.GET( NEW_WORD(INDEX) );
    end loop;
  end GET_WORD;

  procedure INVERT_WORD ( BACK_WORDS : in out CAP_ARR ) is
    TEMP_WORD : CAP_ARR;
  begin
    for INDEX in reverse BACK_WORDS'RANGE loop
      TEMP_WORD( BACK_WORDS'LAST - INDEX +1 ) := BACK_WORDS(INDEX);
    end loop;
    BACK_WORDS := TEMP_WORD;
  end INVERT_WORD;

  procedure PRINT_WORD ( FOR_WORD : in CAP_ARR ) is
  begin
    for INDEX in FOR_WORD'RANGE loop
      TEXT_IO.PUT( FOR_WORD( INDEX ) );
    end loop;
  end PRINT_WORD;

begin  -- INVERT_ARRAY

  GET_WORD( WORD );
  INVERT_WORD( WORD );
  PRINT_WORD( WORD );

end INVERT_ARRAY;
```

EXAMPLE 7-1

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                     %
--%  Abstract:  This package contains trig functions that work on the   %
--%             predefined type FLOAT.  It contains the traditional trig %
--%             functions, arc trig functions, and hyperbolic trig       %
--%             functions.                                              %
--%                                                                     %
--%  Authors:   W A WHITAKER  AFATL EGLIN AFB FL 32542                  %
--%             T C EICHOLTZ  USAFA                                     %
--%                                                                     %
--%  Date:      16 JULY 1982                                            %
--%                                                                     %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

package TRIG_LIB is

   function SIN(X : FLOAT) return FLOAT;
   function COS(X : FLOAT) return FLOAT;
   function TAN(X : FLOAT) return FLOAT;
   function COT(X : FLOAT) return FLOAT;

   function ASIN(X : FLOAT) return FLOAT;
   function ACOS(X : FLOAT) return FLOAT;
   function ATAN(X : FLOAT) return FLOAT;
   function ATAN2(V, U : FLOAT) return FLOAT;

   function SINH(X : FLOAT) return FLOAT;
   function COSH(X : FLOAT) return FLOAT;
   function TANH(X : FLOAT) return FLOAT;

end TRIG_LIB;

-----------------------------------------------------------------------------

package body TRIG_LIB is

   function SIN(X : FLOAT) return FLOAT is separate;
   function COS(X : FLOAT) return FLOAT is separate;
   function TAN(X : FLOAT) return FLOAT is separate;
   function COT(X : FLOAT) return FLOAT is separate;
   function ASIN(X : FLOAT) return FLOAT is separate;
   function ACOS(X : FLOAT) return FLOAT is separate;
   function ATAN(X : FLOAT) return FLOAT is separate;
   function ATAN2(V, U : FLOAT) return FLOAT is separate;
   function SINH(X : FLOAT) return FLOAT is separate;
   function COSH(X : FLOAT) return FLOAT is separate;
   function TANH(X : FLOAT) return FLOAT is separate;

end TRIG_LIB;
```

```
--!!!!!!!!!!!!!!!!!!!!!!!!!!!!! PRELIMINARY VERSION !!!!!!!!!!!!!!!!!!!!!!!!!!!
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                          %
--%  Abstract:  The following routine is coded with reference to the        %
--%             algorithms and coeficients given in "Software Manual for    %
--%             the Elementry Functions" by William J. Cody, Jr. and        %
--%             William Waite, Prentice_Hall, 1980.  This particular        %
--%             version is stripped to work with FLOAT and INTEGER and      %
--%             uses a mantissa represented as a FLOAT.  A more general     %
--%             formulation uses MANTISSA_TYPE, etc.                        %
--%                                                                          %
--%  Authors:   W A WHITAKER  AFATL EGLIN AFB FL 32542                      %
--%             T C EICHOLTZ  USAFA                                         %
--%                                                                          %
--%  Date:      16 JULY 1982                                                %
--%                                                                          %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

separate (TRIG_LIB)
with CORE_FUNCTIONS;
function SIN(X : FLOAT) return FLOAT is

   C1        :  constant FLOAT := 3.140625;
   C2        :  constant FLOAT := 9.6765_35897_93E-4;
   SGN, Y    : FLOAT;
   N         : INTEGER;
   XN        : FLOAT;
   F, G,
   X1, X2    : FLOAT;
   RESULT    : FLOAT;
   YMAX      : FLOAT := FLOAT(INTEGER(CORE_FUNCTIONS.PI *
                        CORE_FUNCTIONS.TWO**(CORE_FUNCTIONS.IT/2)));
   BETA      : FLOAT := CORE_FUNCTIONS.CONVERT_TO_FLOAT(CORE_FUNCTIONS.IBETA);
   EPSILON : FLOAT := CORE_FUNCTIONS.BETA ** (-CORE_FUNCTIONS.IT/2);

begin  -- SIN

   if X < CORE_FUNCTIONS.ZERO then
      SGN := -CORE_FUNCTIONS.ONE;
      Y := -X;
   else
      SGN := CORE_FUNCTIONS.ONE;
      Y := X;
   end if;

   N := INTEGER(Y * CORE_FUNCTIONS.ONE_OVER_PI);
   XN := CORE_FUNCTIONS.CONVERT_TO_FLOAT(N);
   if N mod 2 /= 0 then
      SGN := -SGN;
   end if;

   X1 := CORE_FUNCTIONS.TRUNCATE(abs(X));
   X2 := abs(X) - X1;
   F := ((X1 - XN*C1) + X2) - XN*C2;
   if abs(F) < CORE_FUNCTIONS.EPSILON then
      RESULT := F;
   else
      G := F * F;
      RESULT := F + F*CORE_FUNCTIONS.R(G);
   end if;
   return (SGN * RESULT);

end SIN;
```

EXAMPLE 7-2

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                  %
--%  Abstract:  This package defines a rational number type.  The    %
--%             following routines are provided to work with these   %
--%             rational numbers:                                     %
--%                                                                  %
--%               NUMERATOR_OF   - Returns the numerator of the number %
--%               DENOMINATOR_OF - Returns the denominator of the number %
--%               MAKE           - Makes a rational number from integers %
--%               "+"            - Adds rational numbers              %
--%               "-"            - Subtracts rational numbers         %
--%               "*"            - Multiplies rational numbers        %
--%               "/"            - Divides rational numbers           %
--%               DISPLAY        - Displays a rational number to terminal%
--%                                                                  %
--%  Author:    Num Burrs                                            %
--%                                                                  %
--%  Date:      23 Nov 85                                            %
--%                                                                  %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

package RATIONAL_NUMBERS is

    MAX_VALUES : constant := INTEGER'LAST;

--

    type VALUES is range -MAX_VALUES .. MAX_VALUES;
    subtype POSITIVE_VALUES is VALUES range 1 .. MAX_VALUES;

    type NUMBER_TYPE is private;

--

    function NUMERATOR_OF( A_NUMBER : NUMBER_TYPE) return POSITIVE_VALUES;

    function DENOMINATOR_OF ( A_NUMBER : NUMBER_TYPE ) return VALUES;

    function MAKE ( TOP    : VALUES;
                    BOTTOM : POSITIVE_VALUES ) return NUMBER_TYPE;

    function "+" ( LEFT, RIGHT : NUMBER_TYPE ) return NUMBER_TYPE;

    function "-" ( LEFT, RIGHT : NUMBER_TYPE ) return NUMBER_TYPE;

    function "*" ( LEFT, RIGHT : NUMBER_TYPE ) return NUMBER_TYPE;

    function "/" ( LEFT, RIGHT : NUMBER_TYPE ) return NUMBER_TYPE;

    procedure DISPLAY ( A_NUMBER : NUMBER_TYPE );

private

    type NUMBER_TYPE is
      record
        NUMERATOR   : VALUES;
        DENOMINATOR : POSITIVE_VALUES;
      end record;

end RATIONAL_NUMBERS;
```

```
with TEXT_IO;
package body RATIONAL_NUMBERS is

  package VALUE_IO is new TEXT_IO.INTEGER_IO( VALUES );

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                          %
--%  Abstract:  This function returns the numerator of the rational         %
--%             number.                                                      %
--%                                                                          %
--%  Author:    Num Burrs                                                    %
--%  Date:      25 Nov 85                                                    %
--%                                                                          %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function NUMERATOR_OF (A_NUMBER : NUMBER_TYPE) return POSITIVE_VALUES is

begin  -- NUMERATOR_OF
  return A_NUMBER.NUMERATOR;
end NUMERATOR_OF;

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                          %
--%  Abstract:  This function returns the denominator of the rational       %
--%             number.                                                      %
--%                                                                          %
--%  Author:    Num Burrs                                                    %
--%  Date:      25 Nov 85                                                    %
--%                                                                          %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function DENOMINATOR_OF (A_NUMBER : NUMBER_TYPE) return VALUES is

begin  -- DENOMINATOR_OF
  return A_NUMBER.DENOMINATOR;
end DENOMINATOR_OF;

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                          %
--%  Abstract:  This function takes a VALUES and a POSITIVE_VALUES and      %
--%             creates a rational number out of them.                       %
--%                                                                          %
--%  Author:    Num Burrs                                                    %
--%  Date:      25 Nov 85                                                    %
--%                                                                          %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function MAKE (TOP    : VALUES;
               BOTTOM : POSITIVE_VALUES) return NUMBER_TYPE is

begin  -- MAKE
  return ( TOP, BOTTOM );
end MAKE;
```

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%  Abstract:  This function adds two rational numbers.                %
--%  Author:    Num Burrs                                               %
--%  Date:      25 Nov 85                                               %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function "+" (LEFT , RIGHT : NUMBER_TYPE) return NUMBER_TYPE is
begin  -- "+"
  return (( LEFT.NUMERATOR * RIGHT.DENOMINATOR) +
             (RIGHT.NUMERATOR * LEFT.DENOMINATOR),
           LEFT.DENOMINATOR * RIGHT.DENOMINATOR);
end "+";

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%  Abstract:  This function subtracts rational numbers.               %
--%  Author:    Num Burrs                                               %
--%  Date:      25 Nov 85                                               %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function "-" (LEFT , RIGHT : NUMBER_TYPE) return NUMBER_TYPE is
begin  -- "-"
   return LEFT + (-RIGHT.NUMERATOR,RIGHT.DENOMINATOR);
end "-";

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%  Abstract:  This function multiplies rational numbers.              %
--%  Author:    Num Burrs                                               %
--%  Date:      25 Nov 85                                               %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function "*" (LEFT , RIGHT : NUMBER_TYPE) return NUMBER_TYPE is
begin  -- "*"
  return (LEFT.NUMERATOR * RIGHT.NUMERATOR,
          LEFT.DENOMINATOR * RIGHT.DENOMINATOR);
end "*";

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%  Abstract:  This function divides rational numbers.                 %
--%  Author:    Num Burrs                                               %
--%  Date:      25 Nov 85                                               %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function "/" (LEFT , RIGHT : NUMBER_TYPE) return NUMBER_TYPE is
begin  -- "/"
   return (LEFT.NUMERATOR * RIGHT.DENOMINATOR,
           LEFT.DENOMINATOR * RIGHT.NUMERATOR);
end "/";

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%  Abstract:  This displays a rational number to the terminal.        %
--%  Author:    Num Burrs                                               %
--%  Date:      25 Nov 85                                               %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure DISPLAY ( A_NUMBER : number_type ) is
begin  -- DISPLAY

   VALUE_IO.PUT( A_NUMBER.NUMERATOR );
   TEXT_IO.PUT("/");
   VALUE_IO.PUT( A_NUMBER.DENOMINATOR );
   TEXT_IO.NEW_LINE;

end DISPLAY;

end RATIONAL_NUMBERS;  -- body
```

Chapter 8

EXCEPTIONS

OBJECTIVE

Given a program specification, student instructional materials, and student notes, add exceptions to correctly implement the program. Program must conform to course software engineering standards. Instructor may provide up to 4 assists.

INTRODUCTION

When working with embedded computer systems, reliability of our software is a major concern. Software is a vital component controlling aircraft or missiles whose failure can have disastrous results. In order to deal with error conditions, Ada defines something called an exception. An exception is the name of a condition that is unusual (an error). We can then specify, via an exception handler, what actions we want to take when this condition occurs.

INFORMATION

There are a number of predefined exceptions in Ada. These are raised automatically whenever the associated condition occurs during the execution of the program. Some examples are CONSTRAINT_ERROR (Raised when a constraint is violated, such as when you assign a value that is out of range to a variable), STORAGE_ERROR (Raised when there is not enough memory left to continue execution) or DATA_ERROR (Raised within TEXT_IO whenever you GET a bad input value). To define what action should be taken when these conditions occur, you can define an exception handler.

HANDLING EXCEPTIONS

When implemented, an exception handler must be placed at the end of the sequence of statements in a frame. A frame can be thought of as any begin-end block, such as a program unit or block statement. The syntax of an exception handler is similar to a case statement and looks like this:

```
with TEXT_IO;
procedure TESTING is

   MAX : constant := 100;
   type SCORES is range 0..MAX;
   MY_SCORE : SCORES;
   package SCORE_IO is new
             TEXT_IO.INTEGER_IO(SCORES);

begin -- TESTING

   TEXT_IO.PUT("Enter test score: ");
   SCORE_IO.GET(MY_SCORE);
   MY_SCORE := MY_SCORE * 2;

exception

   when TEXT_IO.DATA_ERROR =>
           TEXT_IO.PUT("Invalid entry." &
                       " Try again.");
   when CONSTRAINT_ERROR =>
           MY_SCORE := MAX;

end TESTING;
```

In this example, if the person entering data at the keyboard enters a 'b' when asked for a test score, we have an error condition named by DATA_ERROR. We say that the exception DATA_ERROR is raised and we go to the associated exception handler to find out what action to take. In this case, the message to try again will be printed.

Likewise, if the multiplication results in an answer out of the range for SCORES (above 100), CONSTRAINT_ERROR will be raised and MY_SCORE will be assigned MAX.

If either exception is raised, once the statements in the exception handler are finished, control passes out of the frame—you DO NOT return to the point where the exception was raised. You can be creative with block statements to localize exceptions, as you'll see in the lecture.

USER DEFINED EXCEPTIONS

The exceptions in the example above were predefined in the language. The Ada compiler inserted object code into the program to test for the conditions and raise the exception. Ada also gives you the capability to define your own exceptions.

The declaration of an exception looks a lot like an object declaration. Remember though that an exception is not an object that we can assign a value to, it just names some condition. See figure 8-1.

Unlike predefined exceptions where the code was automatically inserted to test for the error condition, user defined exceptions require the programmer to write the code to test for the condition and raise the exception.

SUMMARY

Exceptions can be powerful tools in handling error conditions that occur during the execution of a program. By using exception handlers, you can write code that will never quit abnormally, unless a hardware error kills the main processor! Instead you can retry the operation that cause the error, try a different algorithm, restart the system, or whatever action is necessary in those circumstances. Ada therefore allows you to build in reliability by letting the programmer, not the operating system, decide what action to take in the event of an error condition.

```
with GIVE_EXTRA_INSTRUCTION;
procedure TESTING is

  MAX     : constant := 100;
  PASSING : constant := 70;

  type SCORES is range 0..MAX;
  subtype FAILING_SCORES is SCORES range 0..PASSING;

  MY_SCORE : SCORES;
  FAILING  : exception;

  package SCORE_IO is new INTEGER_IO(SCORES);

begin

  TEXT_IO.PUT("Enter test score: ");
  SCORE_IO.GET(MY_SCORE);
  if MY_SCORE in FAILING_SCORES then
     raise FAILING;
  end if;

exception

  when TEXT_IO.DATA_ERROR =>
          TEXT_IO.PUT("Invalid entry.  Please try again.");
  when FAILING =>
          TEXT_IO.PUT("Student failing");
          GIVE_EXTRA_INSTRUCTION;
end TESTING;
```

Figure 8-1. User Defined Exceptions

EXERCISE 8-1

1. Modify the following procedure to handle DATA_ERROR conditions. Make it keep trying to get numbers until it gets two correct numbers.

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                         %
--%  Abstract:                                                              %
--%                                                                         %
--%  Author:                                                                %
--%                                                                         %
--%  Date:                                                                  %
--%                                                                         %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

with TEXT_IO;
procedure ADD_NUMBERS is

  MAX_NUMBER : constant := 1_000;

--

  type NUMBER_TYPE is range 0..MAX_NUMBER;

--

  FIRST_NUMBER,
  SECOND_NUMBER,
  TOTAL_NUMBER    : NUMBER_TYPE;

--

  package NUMBER_IO is new TEXT_IO.INTEGER_IO(NUMBER_TYPE);

begin  -- ADD_NUMBERS

  NUMBER_IO.GET(FIRST_NUMBER);
  NUMBER_IO.GET(SECOND_NUMBER);
  TOTAL_NUMBER := FIRST_NUMBER + SECOND_NUMBER;

end ADD_NUMBERS;
```

EXAMPLE 8-1

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                     %
--%  Abstract:  This package implements an aircraft auto pilot.        %
--%             The package contains procedures to:                     %
--%                - Get the current altitude                           %
--%                - Disengage the auto pilot                           %
--%                     .                                               %
--%                     .                                               %
--%                     .                                               %
--%                                                                     %
--%  Author:    T. Gunn                                                 %
--%  Date:      22 Jun 88                                               %
--%                                                                     %
--%  Propagated Exceptions:                                             %
--%                                                                     %
--%             INPUT_ERROR    - Raised when incorrect altitude read    %
--%             TOO_HIGH_ERROR - Raised when altitude too high          %
--%             TOO_LOW_ERROR  - Raised when altitude too low           %
--%                                                                     %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

package AUTO_PILOT_PACKAGE is

  MAX_ALTITUDE      : constant := 100_000;
  MIN_SAFE_ALTITUDE : constant := 1_000;
  MAX_SAFE_ALTITUDE : constant := 80_000;

--

  type ALTITUDE_TYPE is range 0..MAX_ALTITUDE;

  subtype TOO_LOW is ALTITUDE_TYPE range 0..MIN_SAFE_ALTITUDE;

  subtype TOO_HIGH is ALTITUDE_TYPE range MAX_SAFE_ALTITUDE..MAX_ALTITUDE;

--

  procedure GET( ALT : out ALTITUDE_TYPE );
  procedure DISENGAGE_AUTO_PILOT;

  --
  -- other subprogams declared here
  --
  -- these subprograms will test for and
  -- raise exceptions declared below when
  -- and where appropriate
  --

--

  INPUT_ERROR    : exception;
  TOO_LOW_ERROR  : exception;
  TOO_HIGH_ERROR : exception;

end AUTO_PILOT_PACKAGE;
```

```
package body AUTO_PILOT_PACKAGE is

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                      %
--%  Abstract:  This procedure gets the altitude from the sensor.        %
--%                                                                      %
--%  Author:    T. Gunn                                                  %
--%                                                                      %
--%  Date:      23 Jun 88                                                %
--%                                                                      %
--%  Propagated Exceptions:                                              %
--%                                                                      %
--%             INPUT_ERROR    - Raised when incorrect altitude read     %
--%             TOO_HIGH_ERROR - Raised when altitude too high           %
--%             TOO_LOW_ERROR  - Raised when altitude too low            %
--%                                                                      %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure GET (ALT : out ALTITUDE_TYPE) is

   TEMP_ALTITUDE : ALTITUDE_TYPE := 0;

begin  -- GET

   -- Code here to get TEMP_ALTITUDE from sensor
   -- INPUT_ERROR is raised of incorrect type of data is received

   if TEMP_ALTITUDE in TOO_LOW then
     raise TOO_LOW_ERROR;
   elsif TEMP_ALTITUDE in TOO_HIGH then
     raise TOO_HIGH_ERROR;
   end if;
   ALT := TEMP_ALTITUDE;
end GET;

   .
   .
   .

end AUTO_PILOT_PACKAGE;
```

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                      %
--%   Abstract:  This procedure is the autopilot that flies the aircraft. %
--%                                                                      %
--%   Author:    T. Gunn                                                  %
--%   Date:      23 Aug 88                                                %
--%                                                                      %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

with AUTO_PILOT_PACKAGE,TEXT_IO;
procedure AUTO_PILOT is

   MAX_TRIES : constant := 3;

   ALTITUDE  : AUTO_PILOT_PACKAGE.ALTITUDE_TYPE := 0;

begin

  for I in 1..MAX_TRIES loop                        -- Get the altitude

     begin
        AUTO_PILOT_PACKAGE.GET( ALTITUDE );
        exit;
     exception
       when AUTO_PILOT_PACKAGE.INPUT_ERROR =>
          if I=3 then                               -- Max of three tries
             raise;
          end if;
     end;  -- Block statement

  end loop;

  --
  -- The meat of AUTO_PILOT will be here
  --

  exception

   when AUTO_PILOT_PACKAGE.INPUT_ERROR =>

            AUTO_PILOT.DISENGAGE_AUTO_PILOT;
            TEXT_IO.PUT(" ***** DISENGAGING AUTO PILOT ***** ");
            TEXT_IO.PUT("         ALTIMETER FAILED           ");

   when AUTO_PILOT_PACKAGE.TOO_HIGH_ERROR =>

            AUTO_PILOT_PACKAGE.DISENGAGE_AUTO_PILOT;
            TEXT_IO.PUT(" ***** DISENGAGING AUTO PILOT ***** ");
            TEXT_IO.PUT(" ALTITUDE TOO HIGH TO USE AUTO PILOT");

   when AUTO_PILOT_PACKAGE.TOO_LOW_ERROR =>

            AUTO_PILOT_PACKAGE.DISENGAGE_AUTO_PILOT;
            TEXT_IO.PUT(" ***** DISENGAGING AUTO PILOT ***** ");
            TEXT_IO.PUT(" ALTITUDE TOO LOW TO USE AUTO PILOT");

 end AUTO_PILOT;
```

Chapter 9

GENERICS

OBJECTIVES

1. Given a program specification, a generic, student instructional materials, and student notes, correctly instantiate a generic to solve the problem. Program must conform to course software engineering standards. Instructor may provide up to 2 assists.

2. Given a program specification, a generic declaration, an incomplete generic body, student instructional materials, and student notes, complete the generic body to correctly solve the problem. Program must conform to course software engineering standards. Instructor may provide up to 4 assists.

INTRODUCTION

Generics. The mere mention of the word makes otherwise gallant programmers tremble. But Ada generics are nothing to be afraid of. Once you understand the significance of strong typing, the concept of a generic program unit is simplified--even natural.

A generic program unit simply makes a subprogram or package more general so we can reuse it in different applications. In order to more clearly see why we need generics, let's imagine the Ada language without generic program units.

INFORMATION

SOFTWARE REUSE

What if we took away the generic INTEGER_IO package and replaced it with a non-generic variety. After all, don't we have non-generic packages for input/output of types CHARACTER and STRING? Why can't we do the same for input/output of integers? Let's call our new package NEW_TEXT_IO and declare in it a PUT routine for integers that looks like this:

```
procedure PUT(ITEM : in INTEGER);
```

If we have an object of type INTEGER, we can then call this procedure to print it out:

```
with NEW_TEXT_IO;
procedure PRINT_NUMBERS is

   NUMBER : INTEGER := 22;

begin  -- PRINT_NUMBERS

   NEW_TEXT_IO.PUT(NUMBER);

end PRINT_NUMBERS;
```

So far, this works fine. But as we want to do more in our PRINT_NUMBERS program, we may have to print out numbers of different types:

```
with NEW_TEXT_IO;
procedure PRINT_NUMBERS is

   type SCORES is range 0 .. 100;
   type LENGTH is range 0 .. 36;

   NUMBER : INTEGER := 22;
   SCORE  : SCORES  := 80;
   SIDE   : LENGTH  := 25;

begin  -- PRINT_NUMBERS

   NEW_TEXT_IO.PUT(NUMBER);
   NEW_TEXT_IO.PUT(SCORE);      --Illegal!
   NEW_TEXT_IO.PUT(SIDE);       --Illegal!

end PRINT_NUMBERS;
```

Remember our strong typing rules!

These rules don't allow us to mix apples and oranges; or INTEGERs, SCORES, AND LENGTHs for that matter. Since our PUT routine has its ITEM parameter declared to be of the predefined type INTEGER, we can't pass it an object of type SCORES or LENGTH. Without generics we would have to declare three PUT procedures in our NEW_TEXT_IO package:

```
procedure PUT(ITEM : in INTEGER);
procedure PUT(ITEM : in SCORES);
procedure PUT(ITEM : in LENGTH);
```

In fact, every time we declared a new integer type, and we need to print out a value of that type, we would have to declare a new procedure.

This solution obviously is unsatisfactory. We have three PUT procedures that do exactly the same thing, yet because of our strong typing rules, all three must be written. Also, types SCORES and LENGTH would have to be visible in package NEW_TEXT_IO, which would, when you think about it, result in forbidding you from declaring new user defined types that need to be PUT or GET. Wouldn't it be nice if we could just take one of these PUT routines and make it general enough so that we don't have to rewrite it two more times!

GENERIC DECLARATIONS

Well, that's exactly what generic program units do. When we compile a generic declaration, we don't define what type that routine will work with—we just define a template. We define the algorithm, but leave a 'dummy' name in place of the type name we want the routine to work with:

```
generic
  type NUM is range <>;
procedure PUT(ITEM : in NUM);
```

Here we've defined a place holder called NUM to be the name of the type of item we can pass to this procedure. We call this place holder a generic formal parameter. Now, when we need to print out an integer value, we can take this template and fill it in by specifying what type we want to take the place of NUM. We do this through a generic instantiation.

GENERIC INSTANTIATION

Once a generic routine is compiled, we can make use of that general routine in different parts of our program or even in different programs. We just have to tell the compiler what type we want to match up with the place holder name we defined when we declared the generic. If we again want to print out numbers as in our previous example, we can use our generic PUT routine that we had defined above, and instantiate it for our types INTEGER, LENGTH, and SCORES:

```
with PUT;
procedure PRINT_NUMBERS is

   type SCORES is range 0 .. 100;
   type LENGTH is range 0 .. 36;

   NUMBER : INTEGER := 22;
   SCORE  : SCORES  := 80;
   SIDE   : LENGTH  := 25;

-- Generic instantiations:

   procedure PUT_INTEGERS is
                        new PUT(INTEGER);
   procedure PUT_SCORES is
                        new PUT(SCORES);
   procedure PUT_LENGTHS is
                        new PUT(LENGTH);

begin -- PRINT_NUMBERS
   PUT_INTEGERS(NUMBER);
   PUT_SCORES(SCORE);
   PUT_LENGTHS(SIDE);
end PRINT_NUMBERS;
```

Logically, the generic instantiation is similar to declaring a brand new subprogram. The difference is that we don't have to rewrite any of the algorithm for the routine. With one line, the instantiation, we can escape from writing many lines of the subprogram body.

SUMMARY

Ada's strong typing rules are such that you must be very specific in passing parameters of the correct type when calling a subprogram. The rules don't allow you to pass a routine a SCORE when it is looking for an INTEGER. Generic program units simply take that subprogram or package and make it more general so that we don't have to rewrite that piece of code many times.

EXERCISE 9-1

Given the following generic function, fill in the main procedure which will instantiate it, as well as I/O for the array component type. It will then make a call to the function and print out the results.

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                        %
--%  Abstract:                                                             %
--%                                                                        %
--%  Author:                                                               %
--%  Date:                                                                 %
--%                                                                        %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

generic

   type INDEX_TYPE is ( <> );
   type INT_TYPE   is range <>;
   type ARR_TYPE   is array ( INDEX_TYPE ) of INT_TYPE;

function GREATEST_VALUE ( LIST : ARR_TYPE ) return INT_TYPE;

------------------------------------------------------------------------------

function GREATEST_VALUE ( LIST : ARR_TYPE ) return INT_TYPE is

  TEMP_INT : INT_TYPE := INT_TYPE'FIRST;

begin

  for I in LIST'RANGE loop
     if LIST(I) > TEMP_INT then
        TEMP_INT := LIST(I);
     end if;
  end loop;

  return TEMP_INT;

end GREATEST_VALUE;
```

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                     %
--%  Abstract:                                                          %
--%                                                                     %
--%  Author:                                                            %
--%  Date:                                                              %
--%                                                                     %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

with TEXT_IO,GREATEST_VALUE;
procedure MAIN is

   --
   -- Declare needed types here
   --

   --
   -- Make needed instantiations here
   --

   --
   -- Declare needed objects here
   --

begin

   --
   -- Fill array and then call function
   --
   -- Print out the results of the function call
   --

end MAIN;
```

GENERIC BODIES

So far we've seen how to instantiate a generic program unit. For the remainder of the chapter, we'll look at writing the bodies of a generic.

Let's consider a generic function that computes the average of two floating point numbers. It may look like this:

```
-- Generic Specification:
generic
  type NUMBERS is digits <>;
function AVERAGE (FIRST,
                  SECOND : NUMBERS)
                  return NUMBERS;
```

```
-- Generic Body:
function AVERAGE (FIRST,
                  SECOND : NUMBERS)
                  return NUMBERS is
begin
  return (FIRST + SECOND) / 2.0;
end AVERAGE;
```

When instantiated, we can logically think of all instances of our generic parameter NUMBERS in the body are replaced by the type name that we instantiated it with. Remember, our generic formal parameter NUMBERS is just a place holder for the name of the type we pass when we instantiate the generic.

Generic Formal Parameters

In order to calculate the average of two floating point numbers, we had to use operations such as addition and division. While these are natural operations for floating point types, it makes no sense at all to add and divide some other types, such as type CHARACTER. There should be some way we can prevent this generic function from being instantiated for CHARACTERs to enforce our limited set of operations allowed by the type definition of enumeration types like CHARACTER. Ada handles this by defining different classes of generic formal type parameters.

Ada allows us to set up generic formal parameters that will match the following classes of types:

- All types
- All but limited private types
- All discrete types
- All integer types
- All floating point types
- All fixed point types
- Array types
- Access types

The language also defines generic formal parameters to pass values, objects, and even subprograms to a generic. These concepts will be covered in the lecture.

Generic Formal Type Parameters

A generic formal type parameter actually defines two things: 1) The types the compiler will allow us to instantiate the generic with and 2) The operations that can be performed on that type within the body of the generic program unit.

There's kind of a trade-off between the types we allow to match during instantiation and the operations that are allowed in the body of the generic. The more types we allow to instantiate the generic with (i.e. the more general it is) the more restricted we are inside the generic as to what we can do to objects of that type. The only predefined operations available inside the generic on an object of a generic formal type parameter are those that are predefined for ALL types that can possibly be matched in an instantiation of the generic. For example, if we set up a generic formal parameter to match all discrete types (integer and enumeration types), we are prevented from using any addition or multiplication operations, since those are not operations defined for ALL discrete types, specifically enumeration types.

SUMMARY

Generic program units are invaluable in building up libraries of reusable code. Fortunately, utilizing existing generic program units in your program through an instantiation is not very difficult--just pick the right generic and pass it the right parameters to instantiate it. Writing a generic is more involved in that you have to decide what operations are needed in the algorithm and how general you want the generic to be when setting up the generic formal parameters.

EXERCISE 9-2

1. This simple function takes in two object of the pre-defined INTEGER type and does a floating point division on them, which returns a value of the pre-defined type FLOAT. Your job is to modify this function so it will take in two objects of any integer type and return a value of any floating point type you choose. (i.e. make it a generic with two generic formal parameters)

2. After the generic is written write a main procedure which tests it using user defined integer and floating point types.

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                         %
--%  Abstract:                                                              %
--%                                                                         %
--%  Author:                                                                %
--%  Date:                                                                  %
--%                                                                         %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function INT_DIVISION ( INT1,INT2 : INTEGER ) return FLOAT is

begin

   return FLOAT(INT1) / FLOAT(INT2);

end;
```

Example 9-1

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                       %
--%  Abstract:  This generic package implements an associative table with %
--%             an abstract state machine.  The package has the following %
--%             subprograms:                                              %
--%                 INSERT   - Places a key and its associated value into %
--%                            the table                                  %
--%                 RETRIEVE - Retrieves the value associated with the    %
--%                            given key                                  %
--%                                                                       %
--%  Generic Parameters:                                                  %
--%             SIZE  - The size of the table                             %
--%             KEY   - The type for the key                              %
--%             VALUE - The type for the associated values                %
--%                                                                       %
--%  Author:    Jimmy Key                                                 %
--%                                                                       %
--%  Date:      7 Mar 87                                                  %
--%                                                                       %
--%  Propagated Exceptions:                                               %
--%             TABLE_IS_FULL  - Raised when the INSERT operation tries   %
--%                              to place an association in a full table. %
--%             ITEM_NOT_FOUND - Raised when the key is not in the table  %
--%                              during the RETRIEVE operation.           %
--%                                                                       %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

generic
      SIZE  :  POSITIVE := 100;
      type KEY is private;
      type VALUE is private;
package TABLE_MAKER is

      procedure INSERT (KEY_ITEM    : KEY;
                        A_VALUE     : VALUE);
      function RETRIEVE (KEY_ITEM : KEY) return VALUE;

--

      TABLE_IS_FULL     :  exception;
      ITEM_NOT_FOUND    :  exception;

end TABLE_MAKER;
```

(Continued on next page)

```
package body TABLE_MAKER is

      type PAIR is record
         A_KEY        :   KEY;
         ITS_VALUE    :   VALUE;
      end record;

      type COUNT is range 0..SIZE;

      subtype INDEX is COUNT range 1..SIZE;

      type TABLE_ARRAY is array (INDEX) of PAIR;

--

      A_TABLE            :   TABLE_ARRAY;
      CURRENT_INDEX      :   COUNT := COUNT'FIRST;

--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                       %
--%  Abstract:  This procedure places a key and its associated value into %
--%             the table.                                                %
--%                                                                       %
--%  Author:    Jimmy Key                                                 %
--%                                                                       %
--%  Date:      7 Mar 87                                                  %
--%                                                                       %
--%  Propagated Exceptions:                                               %
--%             TABLE_IS_FULL  - Raised when the INSERT operation tries   %
--%                              to place an association in a full table. %
--%                                                                       %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      procedure INSERT (KEY_ITEM : in     KEY;
                        A_VALUE  : in     VALUE) is

      begin  -- INSERT

        if CURRENT_INDEX = SIZE then
          raise TABLE_IS_FULL;
        end if;

        CURRENT_INDEX := CURRENT_INDEX + 1;
        A_TABLE(CURRENT_INDEX) := (KEY_ITEM,A_VALUE);

      end INSERT;
```

-- (Continued on next page) --

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                   %
--%  Abstract:  This function returns the value associated with the given %
--%             key.                                                  %
--%                                                                   %
--%  Author:    Jimmy Key                                             %
--%                                                                   %
--%  Date:      7 Mar 87                                              %
--%                                                                   %
--%  Propagated Exceptions:                                           %
--%             ITEM_NOT_FOUND - Raised when the key is not in the table %
--%                              during the RETRIEVE operation.       %
--%                                                                   %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      function RETRIEVE (KEY_ITEM : KEY) return VALUE is

      begin  -- RETRIEVE

                                              --Search table backwards linearly.
        for THIS_INDEX in reverse INDEX'FIRST..CURRENT_INDEX loop
          if A_TABLE(THIS_INDEX).A_KEY = KEY_ITEM then
            return A_TABLE(THIS_INDEX).ITS_VALUE;
          end if;
        end loop;

        raise ITEM_NOT_FOUND;

      end RETRIEVE;

end TABLE_MAKER;
```

-- (Continued on next page) --

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                       %
--%  Abstract:  This program manipulates the generic TABLE_MAKER declared %
--%             previously.  It instantiates two tables, a height table   %
--%             and an amount table.                                      %
--%                                                                       %
--%  Author:    Jimmy Key                                                 %
--%  Date:      9 Mar 87                                                  %
--%                                                                       %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

with TABLE_MAKER,TEXT_IO;
procedure TABLE_INSTANCES is

     MAX_AMOUNT    :   constant := 1_000_000.0;
     MAX_HEIGHT    :   constant := 100;
     MIN_HEIGHT    :   constant := 6;
     MAX_STRING    :   constant := 20;

--

     subtype NAME is String(1..MAX_STRING);

     type HEIGHT is range MIN_HEIGHT..MAX_HEIGHT;   --  inches

     type DOLLAR is digits 6 range 0.0..MAX_AMOUNT;

--

     HOW_TALL   :   HEIGHT;
     AMOUNT     :   DOLLAR;

--

     package HEIGHT_TABLE is new TABLE_MAKER(KEY   => NAME,
                                            VALUE => HEIGHT);

     package AMOUNT_TABLE is new TABLE_MAKER(KEY   => NAME,
                                            VALUE => DOLLAR,
                                            SIZE  => 200);

begin  -- TABLE_INSTANCES

  HEIGHT_TABLE.INSERT("Clyde                ", 69);
  AMOUNT_TABLE.INSERT("Bonnie               ", 10_000.0);

  HOW_TALL := HEIGHT_TABLE.RETRIEVE("Clyde                ");
  AMOUNT   := AMOUNT_TABLE.RETRIEVE("Bonnie               ");

exception

  when HEIGHT_TABLE.TABLE_IS_FULL =>
    TEXT_IO.Put("Height table is full!");
  when AMOUNT_TABLE.TABLE_IS_FULL =>
    TEXT_IO.Put("Amount table is full!");
  when HEIGHT_TABLE.ITEM_NOT_FOUND =>
    TEXT_IO.Put("HEIGHT not found in Height table!");
  when AMOUNT_TABLE.ITEM_NOT_FOUND =>
    TEXT_IO.Put("Amount not found in Amount table!");

end TABLE_INSTANCES;
```

Chapter 10

TASKS

OBJECTIVE

Given a program specification, an incomplete program, student instructional materials, and student notes, add tasks to correctly implement the program. Program must conform to course software engineering standards. Instructor may provide up to 5 assists.

INTRODUCTION

We can define an Ada task as a program unit that logically executes in parallel with other program units. A key word in that definition is "logically"; a program with Ada tasks can not only run on multiple processor machines, but it can run on a machine with a single processor as well. In this case, the execution of a task is somehow interleaved with the execution of the main program and other tasks; many of the decisions as to how this is done is left up to the compiler implementation. Programming with tasks can therefore be pretty tricky, especially if the tasks must communicate with each other a great deal.

In this chapter we'll give you just a brief taste of what tasks look like and how they work.

INFORMATION

SPECIFICATION

A task is like any other program unit in that it has the same two parts, a specification and a body, that other program units have. As you might expect, the specification defines the communication interface between the task and other program units. One difference between tasks and other program units is that a task cannot be a library unit; it is ALWAYS in the declarative part of another program unit.

A simple task that goes off on its own and doesn't talk with other program units can have a simple specification:

```
task CHECK_SENSORS;
```

If, on the other hand, we needed to communicate with a task, we can do so through entries defined in the task specification:

```
task ALTIMETER is
  entry READING (HEIGHT : out NATURAL);
end ALTIMETER;
```

In this case, we initiate communication with this task by issuing an entry call. This entry call can be given from any sequence of statements of any other program unit where entry READING is visible. The entry call would look like this (assuming that ALTITUDE is a variable of the subtype POSITIVE):

```
ALTIMETER.READING(ALTITUDE);
```

When this line is reached in a sequence of statements, the program unit will wait until the ALTIMETER task is ready to accept communication through the READING entry.

BODY

The body of a task contains the sequence of statements to be performed by the task. The syntax is very similar to the syntax of other program unit bodies:

```
task body ALTIMETER is

  LOCAL_ALTITUDE : NATURAL := 0;

begin
  loop
    . -- Perform statements to check
    . -- air pressure and compute
    . -- LOCAL_ALTITUDE.
    accept READING
          (HEIGHT : out NATURAL) do
      HEIGHT := LOCAL_ALTITUDE;
    end READING;
  end loop;
end ALTIMETER;
```

When ALTIMETER reaches the accept statement, it waits until some other program unit calls the READING entry before it moves on. Once someone calls the entry, the statements inside the accept statement are executed and we say the two tasks are in **rendezvous**. This term just names the lifetime of task communication.

TASKING STATEMENTS

We said in the previous example that with the accept statement for the READING entry, task ALTIMETER would wait until an entry call is made to the entry. This is not a good situation to be in if the task should be off doing some critical work. For example, in the ALTIMETER task, we probably wouldn't want to just wait around at the accept statement for another program unit to inquire about the altitude. This would be like a newsstand owner not selling today's papers until yesterday's were all sold out. Ideally, if nobody is waiting to get the altitude, the task should go back and compute an updated value.

Ada allows this capability through various forms of the **select** statement. The select statement allows a task to select between accepting an entry or performing some other action. Your instructor will be covering the details during the lecture.

SUMMARY

Ada tasks allow multiple threads of control to be set up in a program. This can make the software more efficient if working with multiple processors. Even with single processor machines, tasks are a good tool to break up the solution to be more understandable. Real world processes that operate in parallel can be coded with tasks to reflect that parallel nature in the software solution.

EXERCISE 10-1

1. In the following program write the task specification for QUEUE_TASK (the task body is provided). Make sure you include the required entries to PUT a value into the queue and TAKE a value off of the queue.

2. Modify the select statement of the task body to do the following:

   a) The task will attempt to rendezvous with a caller, but only if it can do so immediately. If no immediate rendezvous is possible, it will execute an else part, which prints out an appropriate message to the terminal.

   b) The task will wait for a caller, but it will wait no longer than 60 seconds. If 60 seconds elapse and rendezvous does not occur, print out an appropriate message to the terminal.

3. The main subprogram should make calls to the task entries. Make the entry calls to do each the following:

   a) The task makes the call, but withdraws it if rendezvous does not ocur within the 60 seconds (timed entry call).

   b) The task will attempt an entry call, but withdraws it if the rendezvous is not immediately possible (conditional entry call). If no rendezvous can occur, it executes the else part of the statement, which prints out an appropriate message.

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                      %
--%  Abstract:  This program places values into a queue and retrieves    %
--%             them later.                                              %
--%                                                                      %
--%  Author:    Andrew Asynchronous                                      %
--%                                                                      %
--%  Date:      20 Oct 86                                                %
--%                                                                      %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure START_QUEUE_TASK is

  MIN_NUMBER   :   constant := 0;
  MAX_NUMBER   :   constant := 10_000;

--

  type NUMBERS is range MIN_NUMBER..MAX_NUMBER;

--

  A_NUMBER   :   NUMBERS := MIN_NUMBER;

--

  --
  -- Write QUEUE_TASK specification here.
  --
```

-- (Continued on next page) --

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                     %
--%  Abstract:  This task is the implementation of the queue.  It has   %
--%             the following entries:                                  %
--%                  PUT  - Place a value into the front of the queue.  %
--%                  TAKE - Retrieve a value from the rear of the queue. %
--%                                                                     %
--%  Author:    Andrew Asynchronous                                     %
--%  Date:      20 Oct 86                                               %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

task body QUEUE_TASK is

  SIZE   :   constant := 10;
--
  subtype THE_COUNT is NUMBERS range 0..SIZE;
  subtype INDEX is NUMBERS range 1..SIZE;
  type SPACE is array (INDEX) of NUMBERS;

  type QUEUE_TYPE is record
    BUFFER    :    SPACE;
    HEAD      :    INDEX := 1;    -- Next value to be removed.
    TAIL      :    INDEX := 1;    -- Next available slot.
    COUNT     :    THE_COUNT := 0;
  end record;

--

  QUEUE   :   QUEUE_TYPE;

begin  -- QUEUE_TASK

    loop

      select
        when QUEUE.COUNT /= SIZE =>
          accept Put(THE_NUMBER   :  in  NUMBERS) do
            QUEUE.BUFFER(QUEUE.TAIL) := THE_NUMBER;
          end Put;

          QUEUE.TAIL  := QUEUE.TAIL  + 1;
          QUEUE.COUNT := QUEUE.COUNT + 1;
      or
        when QUEUE.COUNT /= 0 =>
          accept Take(THE_NUMBER   :  out  NUMBERS) do
            THE_NUMBER := QUEUE.BUFFER(QUEUE.HEAD);
          end Take;

          QUEUE.HEAD  := QUEUE.HEAD  + 1;
          QUEUE.COUNT := QUEUE.COUNT - 1;
      or
        terminate;
      end select;

    end loop;

end QUEUE_TASK;

--%%%%%%%%%%%%%%%%%%%%%%%%% START_QUEUE_TASK %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

begin  --START_QUEUE_TASK

  -- Make calls to task.

end START_QUEUE_TASK;
```

EXAMPLE 10-1

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                         %
--%  Abstract:  This program counts the number of each character from      %
--%             strings that have been entered from the keyboard.  The     %
--%             total count is the total of each character since the       %
--%             start of the program.                                       %
--%                                                                         %
--%  Author:    Count Chars                                                 %
--%  Date:      20 Oct 85                                                   %
--%                                                                         %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

with TEXT_IO;
procedure TASK_EXAMPLE is

  MAX_NUM            : constant := 100;
  CHARACTERS_IN_LINE : constant := 20;

--

  subtype CHARS_TO_COUNT is CHARACTER range 'a'..'z';

  type COUNT_NUM is range 0..MAX_NUM;

  type CHAR_COUNT is array ( CHARS_TO_COUNT ) of COUNT_NUM;

  subtype LINE is STRING(1..CHARACTERS_IN_LINE);

--

  MY_LINE : LINE;
  CHAR    : CHARACTER := 'Y';
  LAST    : NATURAL := 0;

--

  task COUNT_CHARS is
    entry SEND_LINE ( A_LINE : in LINE );
    entry PRINT_COUNT;
  end COUNT_CHARS;

  package COUNT_IO is new TEXT_IO.INTEGER_IO(COUNT_NUM);


--                          (Continued on next page)
```

```
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                                                                       %
--%  Abstract:  This task counts the number of each character in the     %
--%             string passed in and maintains the running total until   %
--%             told to print it out.  The entries to the task are:      %
--%                SEND LINE    - Call to give task the string to count  %
--%                PRINT_COUNT  - Call to print out number of each       %
--%                                   character counted                  %
--%                                                                       %
--%  Author:    Count Chars                                              %
--%  Date:      20 Oct 85                                                %
--%                                                                       %
--%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

task body COUNT_CHARS is

  LOCAL_LINE : LINE;
  COUNTER : CHAR_COUNT := ( others => 0 );

begin  -- COUNT_CHARS

  loop
    select

      accept SEND_LINE ( A_LINE : in LINE ) do
         LOCAL_LINE := A_LINE;
      end;

      for I in LOCAL_LINE'RANGE loop
        if LOCAL_LINE(I) in CHARS_TO_COUNT then
          COUNTER(I) := COUNTER(I) + 1;
        end if;
      end loop;

    or

      accept PRINT_COUNT;

      TEXT_IO.PUT_LINE(" THE COUNT OF THE CHARACTERS IS => ");
      for I in COUNTER'RANGE loop
        TEXT_IO.PUT(" NUMBER OF ");
        TEXT_IO.PUT(I);
        TEXT_IO.PUT("'S => ");
        COUNT_IO.PUT(COUNTER(I));
        TEXT_IO.NEW_LINE;
      end loop;

    or

      terminate;

    end select;
  end loop;

end COUNT_CHARS;

--
```

(Continued on next page)

```
--XXXXXXXXXXXXXXXXXXXXXXXXXX TASK_EXAMPLE XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

begin  -- TASK_EXAMPLE

  loop

    TEXT_IO.PUT_LINE(" ENTER UP TO 20 CHARACTERS => ");
    TEXT_IO.GET_LINE( MY_LINE,LAST );
    COUNT_CHARS.SEND_LINE(MY_LINE);
    TEXT_IO.PUT(" DO YOU WISH TO CONTINUE (Y or N) : ");
    TEXT_IO.GET(CHAR);
    TEXT_IO.SKIP_LINE;

    exit when ( CHAR = 'N' or CHAR = 'n' );

  end loop;

  COUNT_CHARS.PRINT_COUNT;

end TASK_EXAMPLE;
```

Chapter 11

PROGRAM DESIGN USING Ada

OBJECTIVES

1. Given a problem specification, student instructional materials, and student notes, student teams will develop a complete informal strategy for the problem. Instructor may provide up to 3 assists.

2. Given a problem specification, informal strategy, student instructional materials, and student notes, student teams will correctly formalize the informal strategy. Instructor may provide up to 4 assists.

3. Given a problem specification, an object oriented design, student instructional materials, and student notes, student teams will correctly transform an object oriented design into Ada Program Design Language. The design language must conform to course software engineering standards. Instructor may provide up to 4 assists.

INTRODUCTION

As the saying goes, there is more than one way to skin a cat. There is also more than one way to design software. These methods may vary anywhere from very structured full life-cycle methodologies with automated tools, to what can best be described as "ad hoc" coding or "hacking".

As one wise man once said, no one tool is always best; we should use the best tool for the job at hand. Software continues to grow more and more complex as we tackle larger projects. Tools that deal with this complexity are evolving also. In older languages, the primary construct for structuring programs is the subprogram. This leads the software designers to structure the software based on the functions to be performed in the system. Ada has other tools besides subprograms to aid in dealing with the complexity of software. Packages and tasks can be used to give a more organized and understandable layout to the code. Using these tools, we are no longer constrained into structuring our software based on function; we now can break it up using objects or overall processes in the system as the structuring criteria.

INFORMATION

Object Oriented Design (OOD) is one design tool that has become popular with Ada. As its name implies, OOD breaks the software up into the abstract objects that exist in the system. It utilizes the package structure as the main building block of the design. The cornerstone of OOD is that a package groups together the definition of the class of objects with the operations that can be performed on objects of this class.

OOD PROCESS

The steps in Object Oriented Design vary somewhat, depending on who you talk to and when. Some organizations use a very general approach with only a few steps while others use an approach that tries to provide more guidance by breaking down the major steps into more detail. There are arguments on both sides of the issue; ask your instructor if you want a more detailed discussion. The general steps to OOD, adapted from Grady Booch in the second edition of his book Software Engineering with Ada, are:

1. Identify the objects that exist in the system and their characteristics.

2. Identify the operations that are performed on those objects.

3. Establish the visibility of each object in relation to the others.

4. Establish the interface of each object.

5. Implement each object.

OOD is an iterative process. First perform the above steps at the highest level of abstraction, encompassing the entire system. In order to perform the final step of implementing the objects, it's likely that you'll have to repeat the steps on the next lower level of abstraction. This will identify secondary objects that are needed in the system, but that didn't show up in your consideration of the overall process. Repeat the steps for each level of abstraction until each object is simple enough to be easily understood and implemented.

There are a few ways to identify the objects and operations in the system. The end product of the analysis of the problem should give you a starting point. Your instructor will show you one way of identifying the objects and operations that makes use of a written paragraph that defines what the system will do.

SUMMARY

Ada, with features such as packages, generics and tasks, has added structuring capabilities over traditional languages. These added capabilities require the use of different design methods if they are to be used to their full advantage. Object Oriented Design attempts to use more of the features of Ada and result in a design that produces more understandable and maintainable code.

EXERCISE 11-1

Develop an Object Oriented Design for a system that will count the change in your pocket. Your solution should use the following steps as its algorithm:

1. Zero out the counter for the total value of the change.

2. (While the pocket is not empty) Take a coin out of the pocket.

3. Determine the value of the coin.

4. Add the value of the coin to the total value.

5. Finally, display the total value after the pocket is empty.

a. Type all Ada reserved words (LRM 2.9) in lower case.

b. User defined names/identifiers should be typed in all upper case; i.e. AIRCRAFT. However, many automated tools and some authors of Ada textbooks suggest capitalizing only the first letter of user defined names/identifiers; i.e. Aircraft. This is acceptable but not as readable.

c. Use the embedded underscore to separate multiple-word identifiers; i.e. MY_AIRCRAFT_TYPE. Also, use embedded underscores to add readability to numbers; i.e. 1_000_000.00.

d. Properly aligned code is much more readable. Try indenting two or three spaces for each level. In addition, align begins and ends with their appropriate parent.

e. Add blank lines to aid readability. Use blank lines to set logically related code apart. Specifically, spacing will depend on what's best to make your program easier to read. Some hints would be to add spaces to offset subprograms and their associated begin, declarations, loop structures, if statements, etc.

f. Identify the name of the subprogram or package with it's end statement; i.e.: end MY_PROGRAM;. In addition, it's a good practice to associate the subprogram name with its begin statement; i.e.: begin -- MY_PROGRAM. Notice you must type in the '--' first since this is only a comment to help the maintenance programmer.

```
|..|..|..| -- < indent levels 2 Or 3 spaces

---------------------------------------------------
-- Abstract: Comment lines outlining the purpose
--           of the program unit.
-- Author:   Name of programmer
-- Date:     Date program unit was written.
---------------------------------------------------

with TEXT_IO; -- context clauses
procedure CODING_FORMAT_DEMONSTRATION is

   -- Declare named numbers
   MAX_VALUE  : constant := 10_000.0;   -- Align throughout to
   LOOP_LIMIT : constant := 5;          -- improve readability.

   -- Declare your various types
   type FIXED_TYPE is delta 0.01 range 0.0 .. MAX_VALUE;
   type LOOP_TYPE  is range 1 .. LOOP_LIMIT;

   -- Group object declarations in one location, one to a line
   MY_FIXED,
   HIS_FIXED_NUMBER,
   YOUR_FIXED_NUMBER : FIXED_TYPE := 0.01;

   package MY_FIXED_IO is new TEXT_IO.FIXED_IO (FIXED_TYPE);
```

```
begin  -- CODING_FORMAT_DEMONSTRATION

   MY_FIXED := 10.0;
   HIS_FIXED_NUMBER := MY_FIXED + YOUR_FIXED_NUMBER;

   for  INDEX  in LOOP_TYPE loop          -- notice the blank lines added
      MY_FIXED  :=  MY_FIXED  +  0.01;    -- to  group the 'for' statement
   end loop;                              -- and improve readability

   if MY_FIXED = YOUR_FIXED_NUMBER then
      -- A series of;
      -- statements if the;
      -- above was true;                -- Use blank lines to
   else                                 -- group any related code.
      -- A series of;
      -- statements if the;
      -- above was false;
   end if;
   MY_FIXED_IO.PUT (MY_FIXED);
end CODING_FORMAT_DEMONSTRATION;
```

g. Yes, documentation is a necessary evil. Much of our code today is not documented well and is a nightmare to maintain. Documentation up front adds to program readability, understandability, maintainability... ... need we say more.

h. As a minimum, we will expect your code to be commented as outlined below. Other comments should be added at your (or the instructors) discretion to enhance understandability of your code:

```
---------------------------------------------------
-- Abstract: Comment lines outlining the purpose
--           of the program unit.
-- Author:   Name of programmer
-- Date:     Date program unit was written.
---------------------------------------------------

procedure DOCUMENTATION is

begin -- DOCUMENTATION

   CODE_STATEMENT;

   loop

      if NO_DOCUMENTATION then          -- what will happen if students
         INSTRUCTOR_WILL_NOT_ACCEPT;    -- don't document their code
         STUDENT_WILL_RE_DO_EXERCISE;   -- is explained here
         UNTIL_IT_IS_RIGHT;
         exit;
      end if;

   end loop;

end DOCUMENTATION;
```

a. When practical, organize the declarative region of a subprogram or a package specification as follows:

NAMED NUMBERS
TYPES
OBJECT DECLARATIONS
TEXT_IO INSTANTIATIONS

b. Separate logical groupings of types by a blank line.

c. Declarations of records should follow this format:

```
type MY_PERSONNEL_FILE is
   record
      .....    -- various record fields
   end record;
```

d. Use names that are descriptive in nature to enhance program readability. Put some thought into this. A meaningful name will greatly enhance the maintenance programmer's job.

e. Don't forget to use meaningful object names also. Your code will be judged for a great part at how readable it is. Your instructor will probably highlight ambiguities wherever possible.

f. Always use 'named numbers' when placing range constraints to your types. This will add a degree of understandability and modifiability to your code by eliminating those "MAGIC NUMBERS" we're used to using. (Exceptions are allowed when range of '0' or '1' are used)

g. As a general rule, objects should be initialized when declared since the language does not implicitly do so.

```
------------------------------------------------------------
-- Abstract: Comment lines outlining the purpose
--           of the program unit.
-- Author:   Name of programmer
-- Date:     Date program unit was written.
------------------------------------------------------------

procedure DEMO_TYPING_STANDARDS is

   MAX_SIZE  :  constant := 100;  -- Named number

   type NBR_OF_ITEMS is range 1 .. MAX_SIZE;

   type AIRCRAFT      is (FIGHTER, BOMBER, TANKER, NONE);
   type CARS          is (FORD, LINCOLN, MERCURY, NONE);
   type BOATS         is (ROW, MOTOR, PADDLE, NONE);

   type BASE_AIRCRAFT is array (NBR_OF_ITEMS) of AIRCRAFT;
   type LOT_CARS      is array (NBR_OF_ITEMS) of CARS;
   type MARINA_BOATS  is array (NBR_OF_ITEMS) of BOATS;
```

-- (Continued on next page)

```
   type TRANSPORTATION_FILE is
      record
         LAND : CARS      := FORD;
         SEA  : BOATS     := ROW;
         AIR  : AIRCRAFT  := FIGHTER ;
      end record;

   THE_AIRCRAFT        : BASE_AIRCRAFT  := ( others => NONE ) ;
   THE_CAR             : LOT_CARS       := ( others => NONE ) ;
   THE_BOAT            : MARINA_BOATS   := ( others => NONE ) ;
   TRANS_HISTORY       : TRANSPORTATION_FILE ;

begin  -- DEMO_TYPING_STANDARDS

   ...

end DEMO_TYPING_STANDARDS;
```

a. Avoid 'HARD CODING' the loop parameter specification. The use of attributes will greatly enhance maintainability of your code.

b. In a 'for loop' statement, use a meaningful name by which to index the loop. Single character names are permitted as the index but are discouraged and are not acceptable during this course:

This is not good:
```
for I in 1 .. 3 loop
   ...
end loop;
```

This is OK:
```
for I in FIGHTER .. TANKER loop
   ...
end loop;
```

But this is better:
```
for AIRCRAFT in AIRCRAFT_TYPE'range loop
   ...
end loop;
```

c. Structuring case statements is important for enhancing readability of your code:

```
case THE_AIRCRAFT is

   when FIGHTER =>
      DO_A_SEQUENCE;
      OF_STATEMENTS;

   when BOMBER =>
      DO_A_SEQUENCE;
      OF_STATEMENTS;

   when TANKER =>
      DO_A_SEQUENCE;
      OF_STATEMENTS;

   when NONE =>
      exit;

end case;
```

a. Now, organize the declarative region of program units containing embedded subprograms as follows:

NAMED NUMBERS
TYPES
OBJECT DECLARATIONS
SUBPROGRAM SPECIFICATIONS (when needed)
TEXT_IO INSTANTIATIONS
SUBPROGRAM BODIES

b. However, as a general rule, don't embed subprograms. Embedded subprograms should be used when their utility is only applicable to the local code. Once embedded, the subprogram is not reuseable. If you do embed subprograms, group the subprogram specifications together, then place the subprogram bodies after any I/O instantiations. This will add to program readability and understandability.

c. When subprograms are not embedded, compile subprogram specification and body to separate files.

d. When prudent to do so, use NAMED_NOTATION for parameters when calling subprograms to aid understandability and future modifiability:

```
------------------------------------------------
-- Abstract: Comment lines outlining the purpose
--           of the program unit.
-- Author:   Name of programmer
-- Date:     Date program unit was written.
------------------------------------------------

procedure STACK_UTILITIES is

   INDEX_SIZE       : constant := 20;
   MAX_NUM_OF_ITEMS : constant := 50;

   type ITEMS  is range 0 .. MAX_NUM_OF_ITEMS;
   type STACKS is array(INDEX) of ITEM;

   THE_STACK : STACKS := ( others => 0 );
   THE_ITEM  : ITEMS  := 0;

   procedure PUSH ( STACK : in out STACKS;
                    ITEM  : in     ITEMS );

   procedure POP ( STACK : in out STACKS;
                   ITEM  :    out ITEMS );

   package ITEMS_IO is new INTEGER_IO (ITEMS);

                        -- (Continued on next page)
```

```
   -------------------------------------------------------
   -- Abstract: Comment lines outlining the
   --           purpose of the program unit.
   -------------------------------------------------------
   procedure PUSH ( STACK : in out STACKS;
                    ITEM  : in     ITEMS ) is

       ...          -- Local declarations for PUSH
   begin -- PUSH
       ...          -- Code for procedure PUSH
   end PUSH;


   -------------------------------------------------------
   -- Abstract: Comment lines outlining the
   -- purpose of the program unit.
   -------------------------------------------------------
   procedure POP ( STACK : in out STACKS;
                   ITEM  :    out ITEMS ) is

       ...          -- Local declarations for POP
   begin  --  POP
       ...          -- Code for procedure POP
   end POP;

begin  --  STACK_UTILITIES

POP ( STACK => THE_STACK,     -- procedure call using named
      ITEM  => THE_ITEM );    -- notation for parameters.

end STACK_UTILITIES;
```

a. Compile the package specification and the package body in separate files.

b. Do not 'use' any package 'withed in' to your program. This will help in tracing program resources. The 'use' clause with TEXT_IO is acceptable.

c. As a general rule, don't declare objects in package specifications. These become global and can cause problems when more than one program unit accesses the package. Named numbers or constant objects are permitted since their value can't change.

d. Only 'with' packages and subprograms where their utility is needed; i.e. you probably don't need TEXT_IO for the package specification but may need it for the package body (so only 'with' it into the body). This is in keeping with the principle of LOCALIZATION.

e. Organize the package specification as follows:

```
------------------------------------------------------
-- Abstract: Comment lines outlining the purpose of
--           each of the program units in the package.
-- Author:   Name of programmer
-- Date:     Date program unit was written.
------------------------------------------------------

with ....  -- context clauses
package PACKAGE_CONTENTS is
   NAMED NUMBERS
   TYPES
   SUBPROGRAM SPECIFICATIONSend PACKAGE_CONTENTS;
```

e. Organize the package body as follows:

```
with TEXT_IO, .....  -- And other context clauses needed
package body PACKAGE_CONTENTS is

   -- local declarations needed by body subprograms
   -- including any TEXT_IO instantiations.

   ----------------------------------------------------------
   -- Abstract: comment lines outlining the
   --           purpose of the program unit.
   -- Author:   Name of programmer (if different from author
   --           of the package)
   -- Date:     Date program unit was written.
   ----------------------------------------------------------
   LOCAL SUBPROGRAMS (NOT DECLARED IN PACKAGE SPECIFICATION)

   ----------------------------------------------
   -- Abstract: Comment lines outlining the
   --           purpose of the program unit.
   ----------------------------------------------
   SUBPROGRAM BODIES TO CORRESPOND WITH THE PACKAGE SPECIFICATION

end PACKAGE_CONTENTS;
```

EXCEPTIONS

a. Exception handlers are designed to handle erroneous conditions. DO NOT use exception handlers with user-defined exceptions, or predefined exceptions to take the place of checks (for situations that will occur normally) that should be handled by the program's executable code.

b. Use a block statement to localize an exception when appropriate. Remember though, overuse of block statements can cause confusion in code readability. If you find this situation, it may be better to create a subprogram for that section of code.

c. If a subprogram is in a package, and if that subprogram propagates a user-defined exception, the name of that exception will be declared in the package specification; and the subprogram documentation will outline the conditions which would result in propagation of the exception. This allows the user of the package to correctly write the main program to provide for the erroneous situation if it occurs. If a subprogram raises a predefined error, that should also be addressed in the subprogram documentation that appears in the package specification.

d. Always align the reserved word exception with its 'begin' and 'end'.

e. Don't rely on, or overuse 'others' as a means of handling exceptions.

```
------------------------------------------------------------------
-- Abstract: Comment lines outlining the purpose of
--           each subprogram in the package.
-- Author:   Name of programmer
-- Date:     Date program unit was written.
-- Propagated Exceptions: must specify any exceptions (i.e.;
--           STACK_OVERFLOW) that will be propagated by
--           a subprogram.
------------------------------------------------------------------
package STACK_PACKAGE is

  type ITEMS  is range 0 .. MAX_NUM_OF_ITEMS;
  type STACKS is array(INDEX) of ITEM;

  procedure PUSH ( STACK : in out STACKS;
                   ITEM  : in     ITEMS );
  procedure POP ( STACK : in out STACKS;
                  ITEM  :    out ITEMS );

  STACK_UNDERFLOW,
  STACK_OVERFLOW   : exception;

end STACK_PACKAGE;
```

```
package body STACK_PACKAGE is

    --------------------------------------------------------------------
    -- Abstract: Comment lines outlining the purpose of
    --           the program unit.
    -- Propagated Exceptions: must specify any exceptions (i.e.;
    --           STACK_OVERFLOW) that will be propagated by
    --           this subprogram.
    --------------------------------------------------------------------
    procedure PUSH ( STACK : in out STACKS;
                     ITEM  : in     ITEMS ) is

    begin -- PUSH

       if SOME_CONDITION then        -- some sequence of statements
          raise STACK_OVERFLOW;      -- that may raise STACK_OVERFLOW
       end if;

    exception

       when STACK_OVERFLOW =>
          raise STACK_OVERFLOW;

    end PUSH;

    --------------------------------------------------------------------
    -- Abstract: Comment lines outlining the purpose of
    --           the program unit.
    -- Propagated Exceptions: must specify any exceptions (i.e.;
    --           STACK_UNDERFLOW) that will be propagated by
    --           this subprogram.
    --------------------------------------------------------------------
    procedure POP ( STACK : in out STACKS;
                    ITEM  :    out ITEMS ) is

    begin -- POP

       if SOME_CONDITION then        -- some suquence of statements
          raise STACK_UNDERFLOW;     -- that may raise STACK_UNDERFLOW
       end if;

    exception

       when STACK_UNDERFLOW =>
          RAISE stack_underflow;

end STACK_PACKAGE;
```

GENERICS

a. The format for a generic unit/specification should be as follows:

```
-----------------------------------------------------------------
-- Abstract:  Comment lines outlining the purpose of the
--            program unit.
-- Author:    Name of the programmer.
-- Date:      Date program unit was written.
-- Propagated Exceptions:  must specify any exceptions that
--            will be propagated by this subprogram.
-----------------------------------------------------------------
generic
   VALUE_PARAMETER  : in SOME_TYPE;
   type GENERAL_PURPOSE is ......;  -- Some generic type declaration
   with procedure NEED_RESOURCE (VALUE : in GENERAL_PURPOSE);
procedure GENERIC_STANDARDS (SOME_OBJECT : in out GENERAL_PURPOSE);

procedure GENERIC_STANDARDS (SOME_OBJECT : in out GENERAL_PURPOSE) is

   -- local declarations

begin -- GENERIC_STANDARDS

   ...; -- sequence of statements that
   ...; -- need the above generic parameters

exception

   when (some condition) =>
      ....; -- some sequence of statements
      ....; -- to handle the condition

end GENERIC_STANDARDS;
```

b. Place generic instantiations within the declarative region at a location that still allows you to group object declarations.

c. Instantiate a generic unit as follows:

```
-----------------------------------------------------------------
-- Abstract: Comment lines outlining the purpose of
--           the program unit.
-- Author:   Name of programmer
-- Date:     Date program unit was written.
-- Propagated Exceptions:  must specify any exceptions that
--           will be propagated by this subprogram.
-----------------------------------------------------------------
with TEXT_IO, GENERIC_STANDARDS, A_PK;
use  TEXT_IO;
procedure DEMO_INSTANTIATION is

   type MATCHING_TYPE is ...........  -- Whatever.
   type A_GENERAL_TYPE is ..........
```

-- (Continued on next page) --

```
   THE_MATCH,
   NON_SPECIFIC : A_GENERAL_TYPE := ?;

   package MATCH_IO is new INTEGER_IO (MATCHING_TYPE);
   use     MATCH_IO;

   procedure GENERIC_INSTANCE is new GENERIC_STANDARDS
                        ( VALUE_PARAMETER => THE_MATCH;
                          GENERAL_PURPOSE => A_GENERAL_TYPE;
                          NEED_RESOURCE   => A_PK.LIKE_PROCDURE);

begin -- DEMO_INSTANTIATION

   ...;
   GENERIC_INSTANCE(NON_SPECIFIC);    -- Procedure call to instantiated
   ....;                              -- procedure

exception

   when (some condition) =>
      ....; -- some sequence of statements
      ....; -- to handle the condition

end DEMO_INSTANTIATION;
```

a. It is best to use entries to communicate with tasks to avoid use of global objects.

b. Locate task specifications after localized subprograms (if any).

c. Group task specifications together when declaring more than one task.

```
----------------------------------------------------------
-- Abstract: Comment lines outlining the purpose of
--           the program unit
-- Author:   Name of programmer
-- Date:     Date program unit was written.
-- Propagated Exceptions:  must specify any exceptions that
--           will be propagated by this subprogram.
------------------------------------------------------------
procedure MAIN is

    -- Named number definitions
    -- Local type definitions
    -- Local object definitions
    -- Any I/O instantiotions

    task SCREEN_CONTROL is
       entry SIEZE;
       entry RELEASE;
    end SCREEN_CONTROL;

    task PRINT is
       entry PRINT1;
    end PRINT;

    ------------------------------------------------------------
    -- Abstract: Comment lines outlining the purpose of
    --    the program unit.
    -- Propagated Exceptions:  must specify any exceptions that
    --           will be propagated by this subprogram.
    ------------------------------------------------------------

    task body SCREEN_CONTROL is

    begin  -- SCREEN_CONTROL
       ...     -- Some code
       select
          accept SIEZE;
             ...  -- More code if required
          accept RELEASE;
          ...
       end select;
    end SCREEN_CONTROL;
```

```
-----------------------------------------------------------------
-- Abstract: Comment lines outlining the purpose of the
--           program unit.
-- Propagated Exceptions:  must specify any exceptions
--            that will be propagated by this subprogram.
-----------------------------------------------------------------
   task body PRINT is

   begin  -- PRINT
      ...      -- Some code
      select
         accept PRINT1;
         ...
      end select;
   end PRINT;

begin   -- MAIN

   PRINT.PRINT1;

exception

   when SOME_CONDITION =>
      ...

end MAIN;
```

Appendix B
Ada GLOSSARY

Abstraction – A principle of Software Engineering. Abstraction is the process of extracting essential information relating to a problem while filtering out the unnecessary (lower level) details that tend to cloud our understanding of the problem.

Access Type – An access type is used in conjunction with the "allocator" statement to dynamically create objects during execution. Keyword: *access*.

Access Value – An access value provides the location of, or "points to", an object which has been created by the evaluation of an allocator. Keyword: *access*.

Accuracy Constraint – An accuracy constraint specifies the relative or absolute error bound of values of a real type. Keyword: *delta*, *digits*.

Ada – The new High Order Language developed under the sponsorship of the United States Department of Defense (DOD) to obtain the benefits of language commonality across a wide variety of computer systems. Ada has been designated by the DOD as the official language for all future embedded computer application programs.

Ada Compiler Validation Capability (ACVC) – An integrated set of tests, procedures, software tools, documentation developed by SofTech, Inc. for conducting validation tests of Ada compilers. The ACVC will be used by the Ada Validation Organization (AVO) to perform formal Ada validation tests.

Ada Integrated Environment (AIE) – The Ada language implementation system, being developed by Intermetrics, Inc., under contract to the U.S. Air Force, to enable the development of programs written in the Ada Language for military computer systems. (See *APSE*)

Ada Language System (ALS) – The Ada language implementation system, developed by SofTech, Inc., under contract to the U.S. Army, that will enable programs in the Ada language for execution on advanced, embedded military target computer systems. The ALS represents the first full Ada Programming Support Environment (APSE) to be supplied to the DOD. (See *APSE*)

Ada Joint Program Office (AJPO) – The DOD office responsible for the encouragement and control of the development of the Ada language and its implementation in DOD computer systems.

Ada Programming Support Environment (APSE) – A full Ada programming environment that enables programmers to write programs in the Ada language, using a standard set of development tools, that can be executed on wide variety of target computers. The Ada language system is a friendly, efficient, flexible, portable, easy to use programming environment.

Ada Software Engineering Education and Training Task Team (ASEET) – The purpose of the ASEET is to provide a detailed and organized approach to the task of identifying the Ada education and training needs of the DOD community, including methodologies and materials to fill those needs.

Ada Validation Organization (AVO) – The component of the AJPO responsible for conducting formal Ada compiler validation tests and for encouraging the correct implementation of the Ada language.

Aggregate – An aggregate is a written form denoting a composite value. An array aggregate denotes a value of an array type; a record aggregate denotes a value of a record type. The components of an aggregate may be specified using either positional or named association.

Allocator – The allocator statement creates a new object of a type designated by an access type, and returns an access value designating the location of the created object.

Ancestor - An ancestor compilation unit of a compilation unit currently being compiled is a member of the following set:

a. A unit mentioned in a *with* clause of the compilation unit currently being compiled;

b. An outer textually-nested unit containing the unit currently being compiled, if that unit is a subunit;

c. The specification part of a subprogram or package body currently being compiled;

d. One of the units mentioned in a *with* clause of the ancestor compilation undefined in parts (b) and (c) above; and

e. Package STANDARD.

In short, it is any compilation unit which is made visible to a compilation unit currently being compiled, not including the unit currently being compiled itself.

Attribute – An attribute is a predefined characteristic pertaining to the definition of a type or an object

Body – A body is a program unit defining the executable portion or implementation of a subprogram, package, or task.

Body Stub – A body stub is a replacement for a body that is compiled separately in a subunit.

Code Generator – The component of a compiler back end that generates the machine language for a specified target computer. Typically, a separate code generator is required for each type of target computer.

Collection – A collection is the entire set of allocated objects designated by an access type.

Compilation Unit – A compilation unit is a program unit which can be compiled independently from any other text. It is optionally preceded by a context clause naming other compilation units upon which it may depend. A compilation unit may be the specification or the body of a subprogram or package.

Compiler – A compiler is a computer program that can translate source programs written in a High Order Language (such as Ada) into machine language programs that can be executed on specified target computers.

Compiler Back End – The portion of the compiler that contains the components which depend upon the characteristics of the target computer, and therefore must be designed specifically for each target computer. (See *Code Generator*)

Compiler Front End – See *Machine Independent Portion*.

Complete program – A program with no unresolved external reference is a complete program.

Completeness – A principle of Software Engineering. Completeness refers to the properties of modules with a system, i.e., the module should be small enough to be understood as a whole, and its interfaces should be clearly defined and strictly enforced. If these conditions are met, it is a trivial matter to ensure that no details are missing from the module in question.

Component – A component is an object that is a part of a larger composite object or a value that is a part of a larger composite value. An indexed component is a name containing expressions denoting indices, and names a component in an array or an entry in a family of entries. A selected component is the identifier of the component prefixed by the name of the entity of which it is a component (such as a record type).

Composite type – An object of a composite type is comprised of one or more components. There are two kinds of composite type: arrays and records. All of the components of an array are of the same subtype; individual components can be selected by their indices. The components of a record may be of different types; individual components can be selected by their identifiers.

Confirmability – A principle of Software Engineering. Confirmability refers to the organization of a system, insofar as it is organized in such a fashion as to promote the efficient and reliable testing of the system.

Constant – See *Object*.

Constraint – A constraint determines a subset of the legal values of a type. A value within that subset is said to satisfy the constraint.

Context Clause – A context clause identifies additional library units upon which a following compilation unit may depend.

Cross Compiler – A compiler that is able to generate machine code for a computer system other than the computer system hosting the compiler.

Declarative Part – A declarative part is a sequence of declarations and related information such as subprogram bodies and representation specifications that apply over a region of a program text.

Delimiter – A separator, such as a comma, semicolon, colon, or parenthesis is called a delimiter.

Derived Type – A derived type is a type whose operations and set of values are taken from those of an existing 'parent' type. Objects of a derived type are not compatible with objects of the parent type.

Discrete Type – The set of values associated with a discrete type is an ordered set of distinct, exact values. Discrete types and values may be used as array or entry indices, loop control parameters, and as choices in case statements and record variants. All integer and enumeration types are discrete.

Discriminant – A discriminant is a specially designated component of a record which allows the structure of a record to take on a variety of different forms. The variations of the record may depend on the value of the discriminant.

Discriminant Constraint – A discriminant constraint specifies a value for each discriminant component in a discriminated record type or object.

DOD – The United States Department of Defense.

Efficiency – A goal of Software Engineering. Efficiency refers to the optimal use of available resources, which, in a computational environment, appear primarily as time and space resources.

Elaboration – The elaboration of declaration is the process by which the declaration achieves its effect (such as the allocation of memory to an object declaration); this process occurs during the execution phase.

Embedded Computer – A computer that is included within, as an integral part, a larger operational system or item of equipment. An embedded computer is typically a small, dedicated, special purpose machine designed to perform specific functions (often control functions) of a larger system. Examples are computers in industrial robotics equipment, navigation systems, and process control devices.

Entity – Anything that may be referred to by name is an Ada entity; objects, types, values and all program units are all entities.

Entry – Entries are communications paths between tasks. Entries within a task are called just as subprograms are called (from outside the task containing the *entry*) and may have parameters associated with them. At least one matching *accept* statement appears in the task body for each entry declared in the task specification.

Enumeration Type – An enumeration type describes a set of discrete values which are specified in the type declaration. These values must be either valid identifiers or character literals.

Evaluation and Validation (E & V) Team – The E & V team is responsible for developing the techniques and tools which will provide a capability to perform assessment of APSEs and determine conformance of APSEs to the Common APSE Interface Set (CAIS).

Exception – An exception names an event that causes normal program execution to terminate. Users can define exceptions meaningful to their application, detect the occurrence of the exception condition, and handle the exception by executing a section of program text in response. (See *Exception Handler*)

Exception Handler – An exception handler is that part of a program that will be executed when an exception condition occurs. If no exception handler is provided and an exception condition occurs, the program will be abnormally terminated.

Expression – Any entity that has a value (including a function call) is considered to be an expression. The term is most often applied to formulas that have a numeric or logical value.

Generic Unit – A generic unit is a non-executable template for a subprogram or a package. A generic unit can accept matching parameters that are either types, objects, and/or subprograms, as specified in the generic formal part. An executable instance of this generic template can be created by the process of generic instantiation.

High Order Language (HOL) – A programming language that enables a programmer to write computer instructions in a English-like, readable form, rather than in a complex machine language. Ada, COBOL, and FORTRAN are examples of high order languages.

Host Computer – A computer system upon which a programming environment is installed to enable the efficient development of programs to be executed on specified target computers. Host computers are typically large; flexible, multiprogramming computers.

Index Constraint – An index constraint specifies the upper and lower bounds for each index range of an array type.

Indexed Component – An indexed component names a component in a array or an entry in a family of task entries.

Information Hiding – A principle of Software Engineering. Information hiding refers to the process of making certain implementation details inaccessible, while allowing the interface to remain visible. Its purpose, allied with the principle of abstraction, is to prevent high-level decisions from being based on low-level characteristics.

Instantiation – The process of causing an executable program unit to be created from a generic template by supplying a matching actual parameter for each generic formal parameter that appears in the formal part of the generic unit.

Integer Type – An integer type is a discrete type whose values represent all integer numbers within a specified range.

KAPSE Interface Team (KIT) – A team of military and DOD contractor personnel, the KIT was organized by the AJPO to identify, examine, and set standardization policies for Kernel Ada Progamming Support Environment (KAPSE) interfaces. The KIT is responsible for defining a standard set of KAPSE interfaces to ensure the interoperability of data and the transportability of tools between conforming APSEs. (See *KAPSE*)

Kernel Ada Programming Support Environment (KAPSE) – A core group of programs that provides basic functions in support of the balance of the Ada Programming Support Environment, and permits the transfer of the APSE to different host computer systems without modification to the KAPSE package bodies.

KAPSE Interface Team from Industry and Academia (KITIA) – The counterpart to the KIT from industry and academia.

Lexical Unit – A lexical unit (or lexical element) is an identifier, a number, a character or string literal, a delimiter, or a comment. Basically, it is the smallest meaningful unit in the Ada language.

Library Unit — A library unit is a separately compilable member of a program library — either the declaration of a generic unit, package or subprogram, a subprogram body, or an instantiation of a generic unit. Within a given program library, the names of all library units must be distinct identifiers.

Limited Type — A limited type is a type for which no predefined operations are implicitly declared. A private type may be limited by the inclusion of the reserved word "limited" in the type declaration. All task types are limited.

Literal — A literal states a value literally, that is, by means of letters and digits. A literal is either a numeric, enumeration, string or character literal.

Localization — A principle of Software Engineering. Localization refers to the grouping of logically related entities in the same physical module, thereby localizing possible error.

Machine Language — The binary language used to communicate with a computer system. Each computer uses its own, unique machine language.

Main Program — The subprogram (usually a parameterless procedure) which initially executes in an Ada system.

Minimal Ada Programming Support Environment (MAPSE) — A minimal group of software tools sufficient to enable programmers to develop programs in Ada.

MI (Machine Independent Portion) — The part of a compiler that contains components which are independent of the characteristics of the target computer, and so can be used in common for many different target computers — often called the compiler "front end".

Model Number — A model number is an exactly representable value of a floating point type. Arithmetic operations on floating point numbers are defined in terms of operations on the model numbers of the type. These operations will be the same on all implementations of Ada.

Modifiability — A goal of Software Engineering. Modifiability refers to a process of controlled change, whether in response to an error or a change in requirements, in which introduced changes do not increase the complexity of the system. Preservation of the original design structure should be an important consideration in achieving modifiability.

Modularity — A principle of Software Engineering. Modularity can be defined as a purposeful structuring of resources. The ideal module is small, has a single purpose, and has a well-defined interface.

Name — A name is a symbol that stands for an entity; the name denotes the entity.

Named Association — A named association specifies the association of an item with one or more positions in a list, by naming the positions.

Object — An object contains a value. A program creates an object by elaborating an object declaration or by evaluating an allocator. In either case, a type is specified for the object, and the object can contain values only of that specified type. An object can be either a variable or a constant.

Object Program — The machine language output of a compiler when a source program is input.

Operation — An operation is an elementary action directly associated with one or more types. The operation is either implicitly declared along with the type declaration, or it is an explicitly declared subprogram that has a parameter or result of the type.

Operator — An operator is an operation that has one or two operands. A unary operator is written before a single operand; a binary operator is written between two operands. This notation, called "infix" notation, is a special kind of function call.

Overloading — Overloading allows operators, subprograms, identifiers, and literals to have more than one meaning at different points within the program text. An overloaded operator or subprogram is one which a user has defined to have a different meaning depending upon the type of parameter it can accept, allowing the definition of several subprograms with the same name. An overloaded enumeration literal is an identifier that appears in the definition of more than one enumeration type. Ada uses type information to select the correct literal or subprogram.

Package — A package is a separately compilable program unit (consisting of a specification and a body) that may contain related types, objects, and subprograms that operate on objects of types defined in the same package specification. The visible part of a package (the part of the specification that appears before the reserved word "private") defines names that may be referenced external to the package by means of a context clause; the private part contains internal declarations of types, objects, and program units that are hidden from the user. The body of a package contains the implementations of subprograms which have been specified in the visible part of the package.

Parameter — A parameter is associated with a subprogram, task entry, or generic unit and is used to communicate with the corresponding program unit body. A *formal* parameter is an identifier used to denote the parameter within the subprogram body, task body, or generic unit body. An *actual* parameter is the entity associated with the corresponding formal parameter at invocation or instantiation time. The *mode* of a parameter specifies whether the associated parameter may be used for input, output, or both. The association of actual parameters with formal parameters can be specified by named association, by positional association, or by a combination of these methods.

Program Design Language (PDL) – An English-like artificial language, sometimes called pseudo-code, used in documenting the design of program unit bodies. The PDL used in the design of the Ada Language System (ALS) uses constructs similar to those in the Ada language, thereby facilitating the transition to final implementation.

Positional Association – A positional association specifies the association of an item with a position in a list, by using the same position in the list.

Pragma – A pragma is an instruction to the compiler to perform actions outside the scope of program logic, such as interfaces with other languages or compiler optimization.

Private Type – A private type is a type which may be used outside the package in which it is declared without knowing its internal data structure. A private type, which may only be declared in a package, is known only by its discriminants (if any) and by the set of operations defined for it (in the same package specification). The only implicitly defined operations applicable to a private type are the tests for equality and inequality and the assignment operation, unless the type is limited, in which case no operations are implicitly defined.

Procedure – (See *Subprogram*)

Program – A program is a collection of one or more compilation units which have all been compiled relative to each other. One of these compilation units must be a subprogram designated as the main program, which invokes other subprograms that are declared in other compilation units.

Program Unit – A program unit is a generic unit, a package, a subprogram, or a task unit.

Programming Environment – An integrated collection of programs that provide a wide variety of program development, configuration management, project control, and maintenance functions. The Ada Programming Support Environment (APSE) is an example of a specialized programming environment.

Program Library – The compilation units that make up a program belong to a program library. A "library unit" from the program library may be specified in a context clause at the start of another compilation unit.

Qualified Expression – A qualified expression further specifies the type of an expression by preceding the expression by an indication of its type or subtype. Qualification is necessary when, in its absence, the expression is ambiguous (perhaps as a result of overloading).

Range – A range is a contiguous set of values of a scalar type. A range is specified by giving the lower and upper bounds of the set of values.

Range Constraint – A range constraint of a type specifies a range, and thereby determines the set of values applicable to the type or subtype.

Real Type – A real type is a type whose values represent approximations to the real numbers. There are two kinds: *fixed point* types are specified with absolute precision by specifying a maximum interval (delta) between values of the type; *floating point* types are specified with relative precision expressed as a number of significant decimal digits.

Rehostability – The capability of a programming environment, such as an APSE, to be moved to a different host computer without major modification. Rehostability is achieved by the concentration of all host dependencies in the KAPSE and in the runtime support libraries. (See *Runtime Support Libraries*)

Reliability – This goal of Software Engineering refers to the ability of a system to operate without human intervention for long periods of time. Reliability must be a prime consideration early in the design; it may not be added at a later time.

Renaming Declaration – A renaming declaration declares another name for an entity.

Rendezvous – A rendezvous is the interaction that occurs between two parallel tasks when one task has called an *entry* of the other task, and a corresponding *accept* statement is being executed by the other ask on behalf of the calling task.

Representation Clause – A representation clause optionally specifies the underlying representation and/or addresses for data and program units.

Retargetability – The capability of a programming environment, such as an APSE, to be made to produce programs for different target environments without major modification. Retargetability is enhanced by designing its basic functions so as machine independent as possible.

Runtime Support Library (RSL) — The component of a compiler back end that provides the additional support functions required for the execution of programs on a specified target computer. Since each type of target computer requires its own supporting functions, a unique runtime support library is required for each type of target computer.

Scalar Type — A scalar type is a type whose values have no components. Integer, real and enumeration types are scalar. Further, the values of a scalar type are ordered.

Scope — The scope of a declaration is that region of text over which the declaration has effect.

Selected Component — A selected component is composed of the name of the component, preceded by the name of the structure of which it is a component. Selected components are used to denote record components, task entries, and objects designated by access values.

Software Engineering — The methods and techniques used in the development of efficient, reliable, and maintainable computer software.

Software Portability — The capability of a program to be moved between different computer systems without modification. Software portability is one of the major goals of the Ada language implementation.

Source Program — A program written in a high order language (such as Ada) for input to a compiler. (See *Object Program*)

Statement — A statement specifies one or more actions to be performed during the execution of a program.

Static Expression — A static expression is an expression whose value does not depend on the execution of the program in which it is contained.

Steelman — The DOD document that specifies the technical and qualitative requirements for the Ada language.

Stoneman — The DOD document that specifies the technical and qualitative requirements for implementating an APSE.

Subprogram — A subprogram is an executable program unit that may have parameters for communication between the subprogram and its invoking program unit. A subprogram declaration specifies the name of the subprogram and lists its formal parameters. The body of a subprogram specifies its execution. A subprogram can be either a *procedure*, which performs a sequence of statements and is invoked by a procedure call statement, or a *function*, which returns a value (called the result), and so a function call is not a statement, but an expression. The subprogram call specifies the actual parameters that are to be associated with the formal parameters.

Subtype — A subtype of a type (called the parent type) characterizes a subset of the values of the type. The bounds of the subset are determined by the constraint on the type. The set of operations applicable to a subtype are the same as that applicable to the parent type. Objects of a subtype are compatible with objects of the parent type.

Software Life Cycle — The span of time over which a software system is in existence, starting with its first conception, and ending with its last use. The software life cycle is usually divided into phases, such as Analysis, Requirements Definition, Design, Code, Validation, and Operation and Maintenance.

Target Computer — A computer, usually embedded in an operational system, that is designated to receive programs in its native machine language from one or more host computers. Target computers are typically small, special-purpose machines.

Task — A task is a program unit that operates in parallel with other program units. It consists of a task specification (which specifies the name of the task and the names and formal parameters of its entries), and a task body, which defines its execution.

Task Type — A task type declaration is a type declaration similar in form to a task specification that permits the subsequent declaration of any number of identical task units. A value of a task type designates a task. All task types are limited types.

Type — A type characterizes a set of values and a set of operations applicable to those values. A type definition is a language construct that defines a type. A particular type is either an access type, an array type, a private type, a record type, a scalar type, or a task type.

Understandability — This goal of Software Engineering must be met in order for any of the other goals to be achieved. The understandability of a system is a measure of how well it reflects a natural view of the world.

Uniformity — A principle of Software Engineering that refers to the consistency of notation within a given system. In order to be understandable, modules should be free from unnecessary differences.

Use Clause — A use clause is a context clause that allows direct reference to declarations that appear in the visible parts of packages named in a *with* clause.

Variant Part — A variant part of a record specifies alternative record components, depending on a discriminant of the record. Each value of the discriminant establishes a particular alternative of the variant part.

Visibility — At a given point in the program text, the declaration of an entity is directly visible if it can be referenced by its simple name. The declaration is "visible by selection" is it can be referenced in a named association or as a component.

With Clause — A with clause is a context clause that allows reference (by expanded name) to declarations that appear in the visible parts of named packages. A with clause also allows direct reference to other named library units, such as generic units and subprograms.

STUDENT HANDOUT

E3OAR4916 003
E3OAR4924 004
E3OAR4924 005
E4OST4916 003
E4OST4924 020
E4OST4924 021
90P 886

Technical Training

OBJECT ORIENTED DESIGN

JANUARY 1987

USAF TECHNICAL TRAINING SCHOOL
3390th Technical Training Group
Keesler Air Force Base, Mississippi

Designed For ATC Course Use RGL - N/A

DO NOT USE ON THE JOB

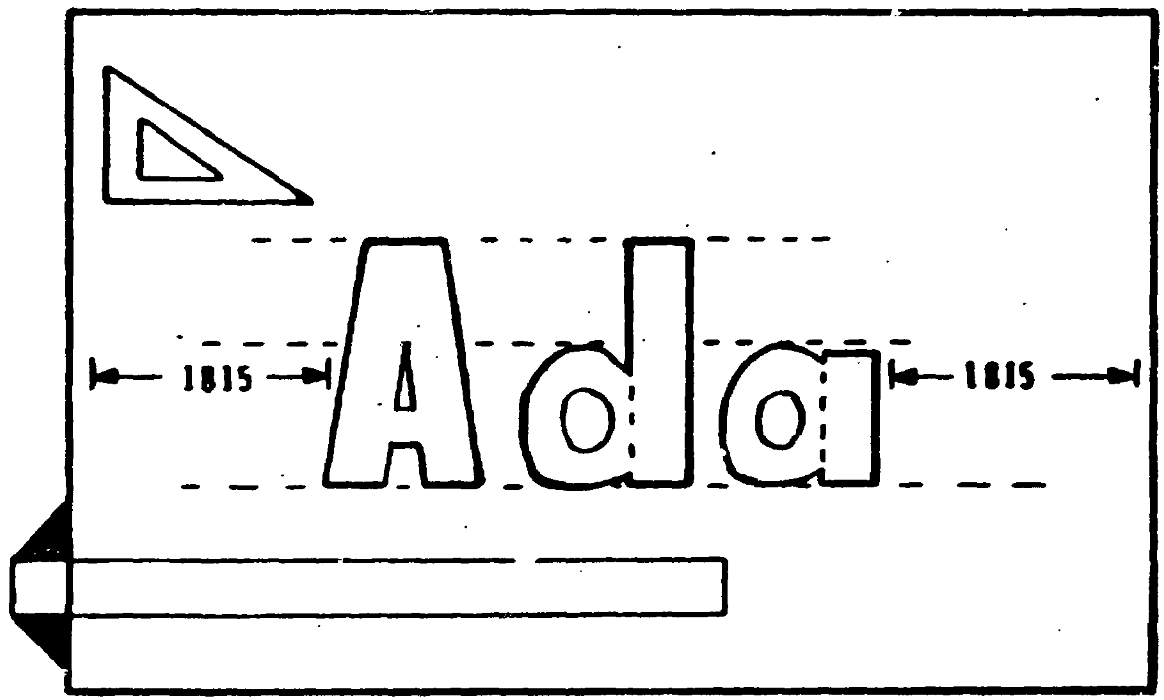
3300 TECHNICAL TRAINING WING
3390 TECHNICAL TRAINING GROUP
KEESLER AIR FORCE BASE, MISSISSIPPI

PHILOSOPHY:

The philosophy of the wing emerges from a deep concern for individual Air Force men and women and the need to provide highly trained and motivated personnel to sustain the mission of the Air Force. We believe the abilities, worth, self-respect, and dignity of each student must be fully recognized; we believe each must be provided the opportunity for the pursuit and mastery of an occupational specialty to the full extent of his or her capabilities and aspirations, and is of immediate and continuing benefit to the individual, the Air Force, and the country. To these ends, we provide opportunities for individual development of initial technical proficiencies, on-the-job training in challenging job assignments, and follow-on growth as supervisors. In support of this individual development, and to facilitate maximum growth of its students, the wing encourages and supports the professional development of its faculty and administrators, and actively promotes innovation through research and the sharing of concepts and materials with other educational institutions.

Supersedes ATC 9OP 886, June 1986

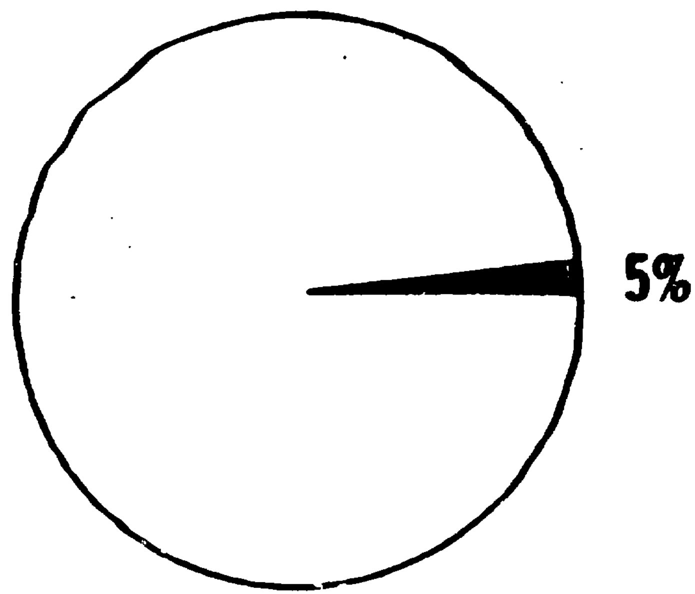
DESIGN



3000

TRADITIONAL DESIGN

- PRELIMINARY DESIGN
- DETAILED DESIGN



SOFTWARE LIFE CYCLE

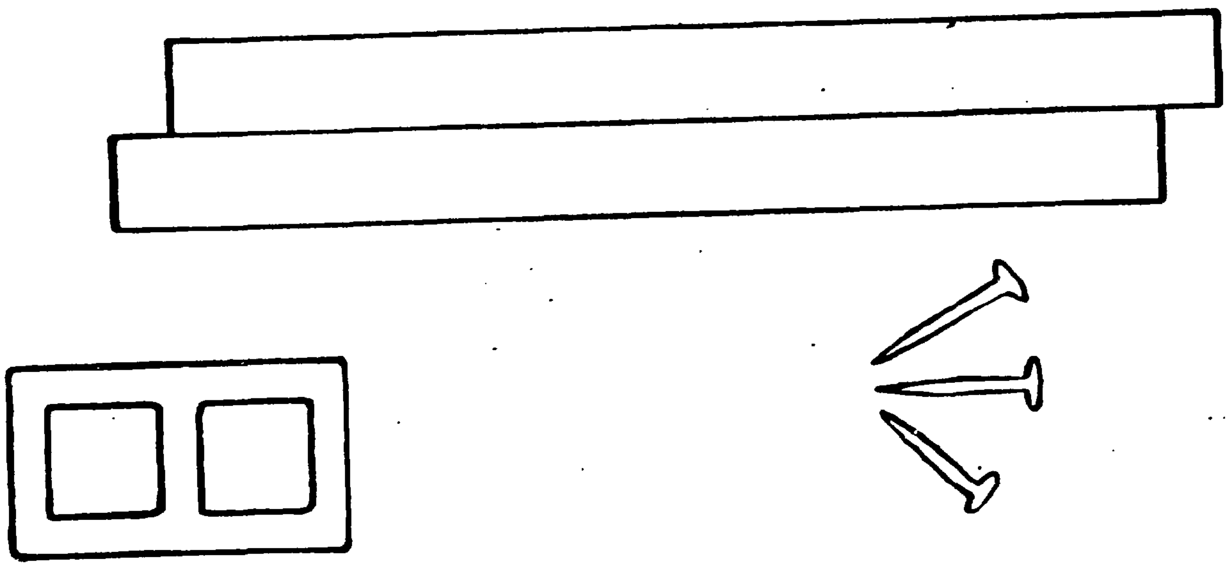
PRELIMINARY DESIGN

- System Flowcharts
- Job Steps
- Program Narratives

DETAILED DESIGN

- Program Flowcharts
- File Layouts
- Data Descriptions

TRADITIONAL APPROACH TO DESIGN



2185

PROBLEMS WITH TRADITIONAL DESIGN

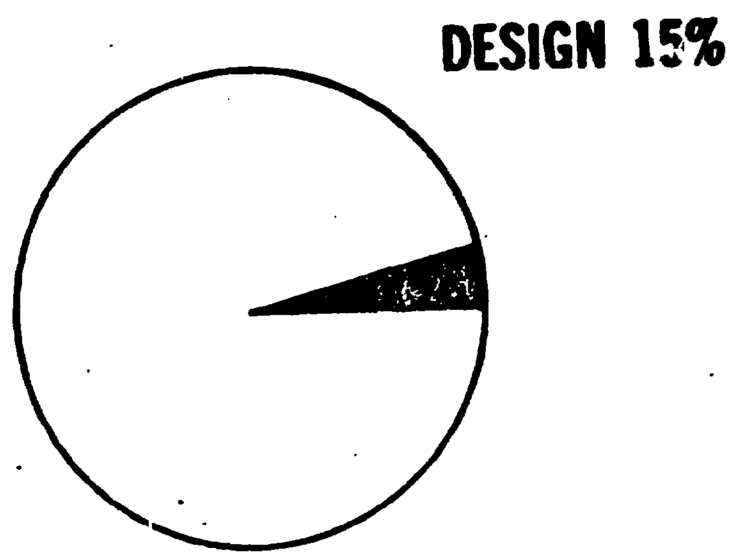
- TIME
- INTERFACES
- DESIGN STRUCTURE
- METHODOLOGY
- DESIGNER
- REQUIREMENTS/DESIGN

2235

PROBLEMS WITH TRADITIONAL DESIGN

— TIME

TOTAL DEVELOPMENT TIME



2195

TIME RAMIFICATIONS

– ERRORS

2200

TIME RAMIFICATIONS
(POOR TESTING)

- Majority of time is spent debugging design errors rather than testing system
- Decreases reliability
- Increases chance that errors will not be identified
- Increases integration time

TIME RAMIFICATIONS
(DESIGN VALIDATION)

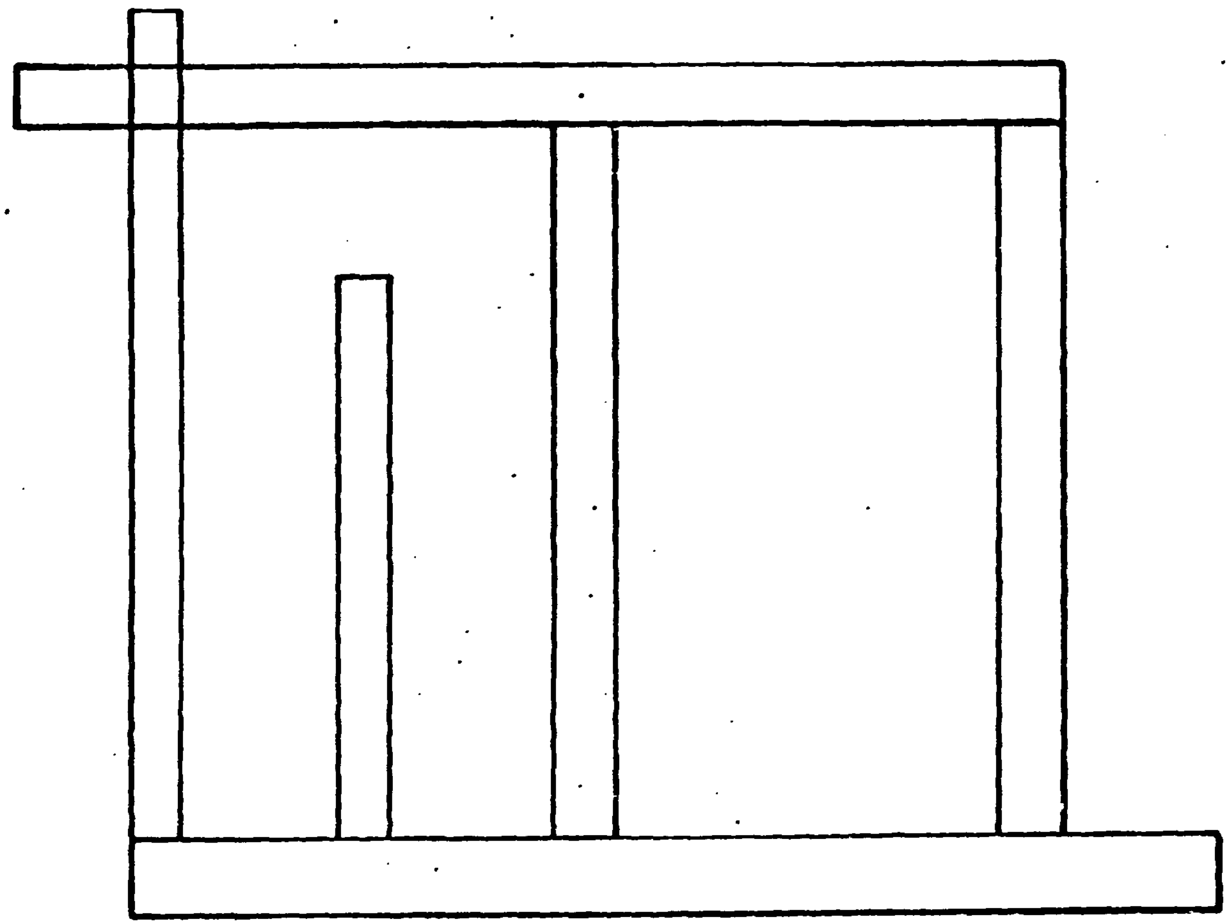
- Too little time is spent validating design

- Commitment to a particular design is made too early

PROBLEMS WITH TRADITIONAL DESIGN

– TIME

– INTERFACES

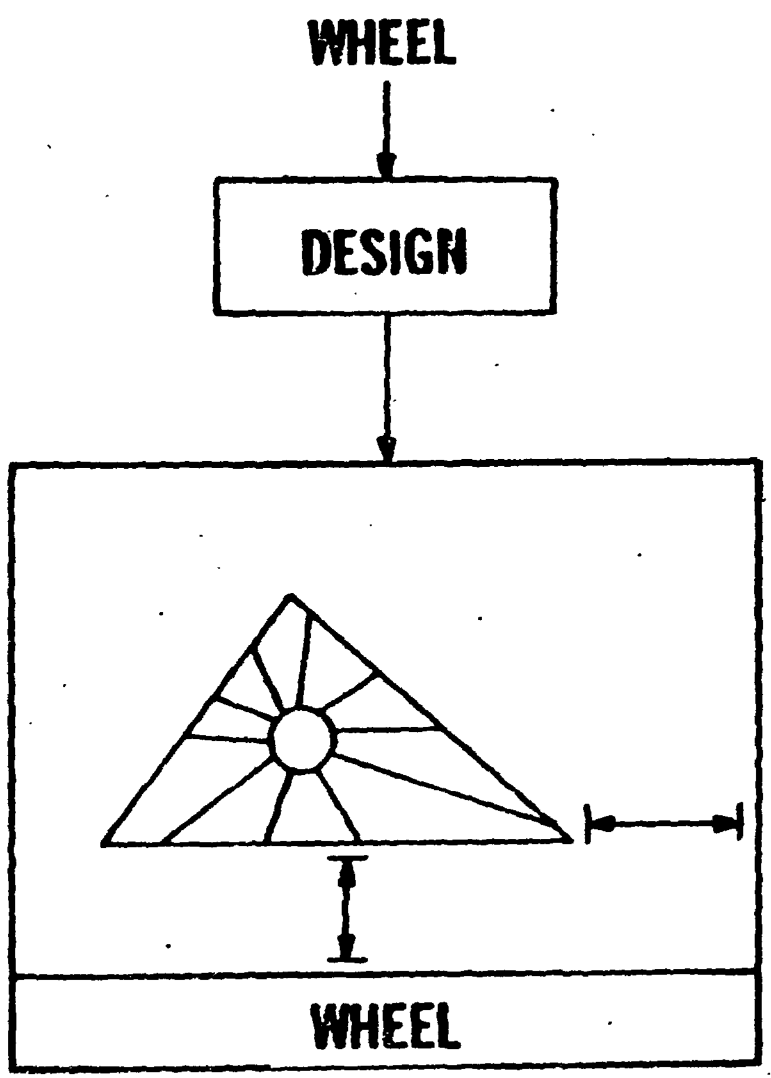


2215

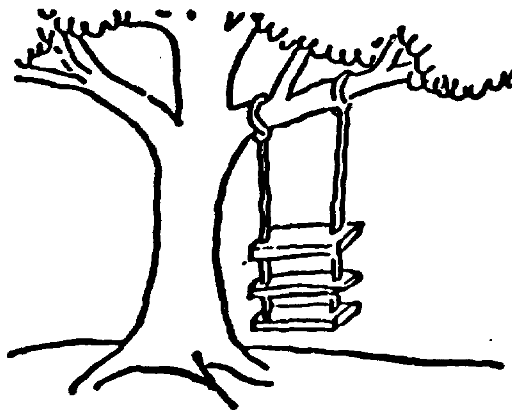
PROBLEMS WITH TRADITIONAL DESIGN

DESIGN STRUCTURE DOESN'T REFLECT PROBLEM; THEREFORE, IT IS

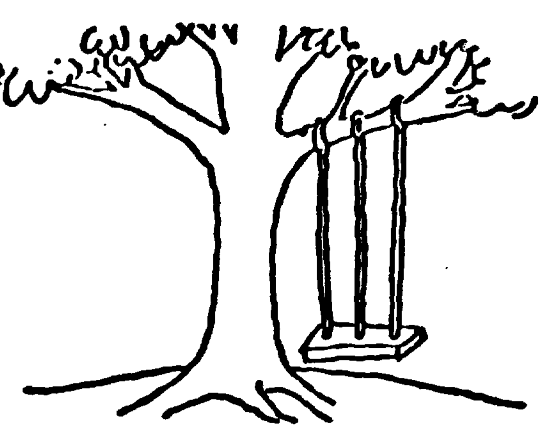
- NOT UNDERSTANDABLE
- NOT RELIABLE
- NOT EFFICIENT
- NOT MODIFIABLE



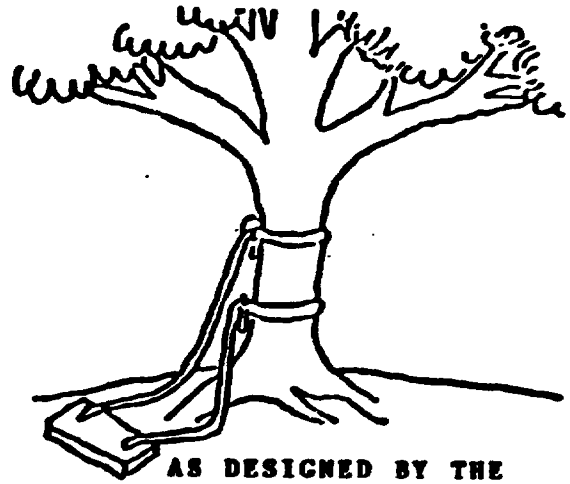
2985



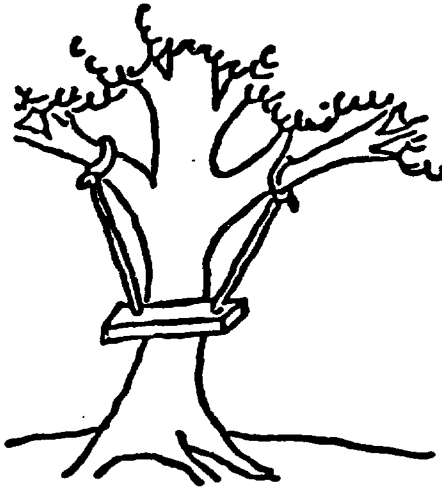
AS SPECIFIED IN THE PROJECT REQUEST



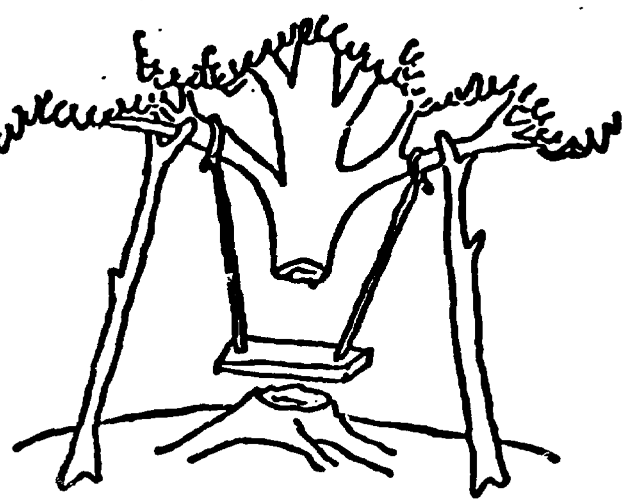
AS PROPOSED BY THE PROJECT SPONSOR



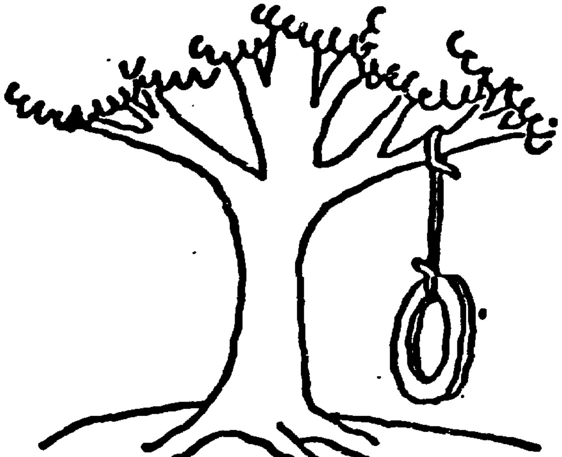
AS DESIGNED BY THE SENIOR ANALYST



AS PRODUCED BY THE PROGRAMMERS



AS INSTALLED AT THE USERS' SITE



WHAT THE USER WANTED

PROBLEMS WITH TRADITIONAL DESIGN

METHODOLOGY

- Well–disciplined methodology is not used or adhered to
- Entrenchment in functional methodologies

FUNCTIONAL METHODOLOGIES

- Do not adequately address data abstraction and information hiding
- Do not express natural concurrency
- Not responsive to changes
- Impose an artificial structure upon system
- Not appropriate for Ada systems

PROBLEMS WITH TRADITIONAL DESIGN

DESIGNER

- Role is well–defined
- Person that does design is usually analyst and/or coder
- Results in a design that looks like analysis or a design that looks like code

PROBLEMS WITH TRADITIONAL DESIGN
REQUIREMENTS/DESIGN DILEMMA

Requirements

N Design Alternatives

Standardization reduces the number of design alternatives that need to be considered and results in requirements that can be truly validated

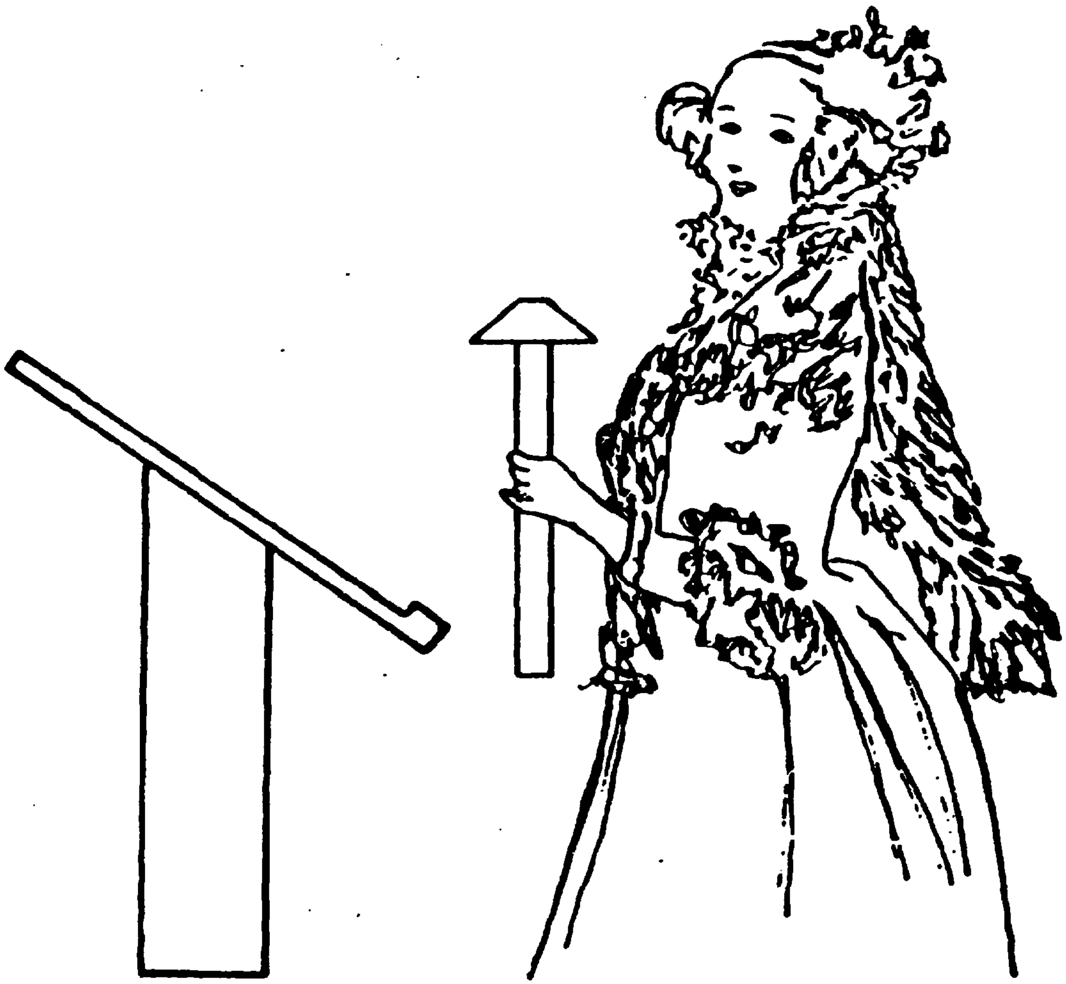
DESIGN PROCESS

METHODMAN

ARCHITECTURAL: Recognition of the overall software structure and interconnection of pieces

DETAILED: Selection of algorithms and data structures appropriate to fulfillment of specific system functions

PROGRAM DESIGN IN Ada



AUTOMATED TOOLS FOR DESIGN PROCESS

- Improve productivity of individual and of development team

- Part of the support environment

  * Documentation System
  * Project Control System
  * Configuration Control System

USING A LANGUAGE DURING DESIGN

– Tendency is to code rather than design

– Allows for automated checking of

  * Software Structure
  * Interfaces
  * "High–level" Logic

– Traditional languages do not contain sufficient structuring features

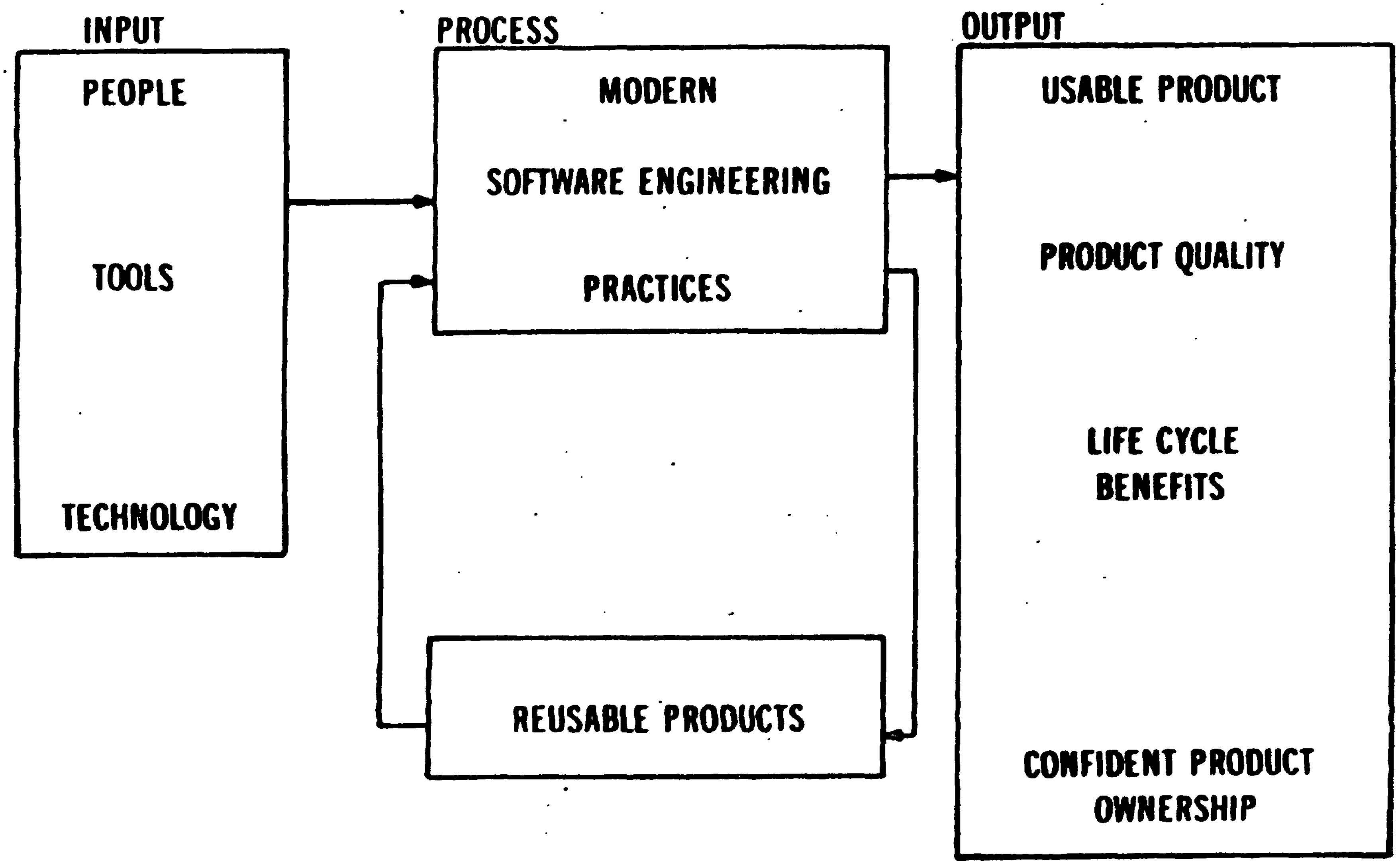
Ada During Design

- Emphasis of the language is on structuring
- Can use compiler as a design tool

PROGRAM DESIGN IN Ada

– Ada PDL

2330



SOFTWARE ENGINEERING

3295

	PURPOSE	OUTER SYNTAX	INNER SYNTAX	
			FUNCTION	DATA
LEVEL 1	USER CONTRACT	ORGANIZING	COMMENTARY	COMMENTARY
LEVEL 2	DESIGN PARTS AND RELATIONSHIPS	STRUCTURING, TASKING	ABSTRACTION	ABSTRACTION
LEVEL 3	DETAILED FUNCTIONAL DESIGNS INDEPENDENT OF TARGET	-	EXPRESSIONS	REFINEMENT, Ada DATA TYPES
LEVEL 4	DETAILED DESIGNS FULLY TARGETED	-	REFINEMENT	REFINEMENT

	PURPOSE	OUTER SYNTAX	INNER SYNTAX FUNCTION	INNER SYNTAX DATA
LEVEL 1	USER CONTRACT	package	COMMENTARY	COMMENTARY

LEVEL 2	DESIGN PARTS AND RELATIONSHIPS	body
		is separate
		begin
		if then
		case
		loop (while, for, exit when)
		function
		task
		entry
		accept, do
			PROCEDURAL CALLS	ABSTRACT DATA STRUCTURES
				PRIVATE DATA TYPES

	PURPOSE	OUTER SYNTAX	INNER SYNTAX: FUNCTION	INNER SYNTAX: DATA
LEVEL 3	DETAILED FUNCTIONAL DESIGNS INDEPENDENT OF TARGET	-	:=, +, -, /, *, ** range rem, mod or, and, xor abs and then or else	Ada DATA TYPES record array range
LEVEL 4	DETAILED DESIGNS FULLY TARGETED, READY FOR IMPLEMENTATION	select	exception, raise abort terminate delay pragma	subtype derived type access type delta, digits constant for, use at

Level 1: (User Contract)

1. CPCI Organizing Syntax

```
package - is
  -- Intended State Machine:
     -- Transition Functions:
          ...
     -- State Data:
          ...
end - ;
```

2. Functional Specification

e.g.,

-- <Receive AUTODIN Segments>

3. CPC Designations

```
procedure - (....) ;
with - ; use - ;
```

4. Commented Abstract Declarations of State Data

e.g.,

```
-- AAAA : ABSTRACT_FILE;
-- XXXX : STACK;
```

Level 2: (Design Parts and Relationships)

1. CPC Organizing Syntax

```
package body - is ...................... end - ;
procedure - (....) is separate;
procedure - (....) is ...... begin .... end - ;
task - entry - (....) ; .... end - ;
task body - is ..............end - ;
accept - (....) do ..........end - ;
```

2. CPC Structuring Syntax (single level)

```
begin .......... end ;
if ....... then .... <else> .... end if ;
case ..... is .... when .... => .... end case ;
for ...... loop ........... end loop ;
while .... loop ........... end loop ;
loop ..... <exit when> .... end loop ;
```

3. Intended Function Commentary

e.g.,

```
--<Send message to user>
```

4. Functional Abstraction

e.g.,

```
SORT (A_TABLE);
GET_HEAD (A_LIST);
```

5. Procedural Calls to Lower Units

e.g.,

MRX_RECEIVING (A_AUTORECS);

6. Data Abstractions and Anonymous Data Structures

e.g.,

package - **is** **new** STRING_FACILITY (type name);

(string, stack, queue, sequence, set, list)

Level 3: (Detailed Design, Independent of Target)

1. Data Tests and Operations

Logical expressions (and, or, xor)

Relational expressions (=, /=, <, <=, >, >=)

Numerical expressions

- o adding (+, -, &)
- o unary (+, -, not)
- o multiplying (*, /, mod, rem)
- o exponentiation (**)

Set membership expressions (in, not in)

2. Data Definition

Predefined Ada data types (INTEGER, BOOLEAN, CHARACTER, FLOAT)

enumerated types (type - is (-,-,-) ;)

array types (type - is array (....) of - ;)

record types (type - is record .... end record ;)

3. Predefined Array Attributes

First

Last

Length

Range

Level 4: (Detailed, Concrete Designs, Fully Targeted to Ada)

1. Exception Handling

exception
when
raise

2. Data Refinement

subtype
derived type
access type
constant
delta
digits
range
renames
all
array slice

3. Tasking Refinements

terminate
select
abort
delay

4. Representation Specification

for
use
at

5. Pragmas (Special Directives)

```
procedure ........... (........: in; ........: out) is
begin
    ....
end ...........;
```

```
procedure GET_TEXT (A_MESSAGE : in STRING) is
--<Build AREA from incoming Message>
begin
    ....
end GET_TEXT;
```

```
procedure GET_TEXT (A_MESSAGE : in STRING) is
--<Build AREA from incoming Message>
    type TEXT is array (1..100) of CHARACTER;
    AREA : TEXT;
begin
    ....
end GET_TEXT;
```

```
procedure GET_TEXT (A_MESSAGE : in STRING) is
--<Build AREA from incoming Message>
    type TEXT is array (1..100) of CHARACTER;
    AREA : TEXT;
begin
    for
        ....
    loop
        .....
    end loop;
    ....
end GET_TEXT;
```

```
procedure GET_TEXT (A_MESSAGE : in STRING) is
--<Build AREA from incoming Message>
    type TEXT is array (1..100) of CHARACTER;
    AREA : TEXT;
begin
    for
        INDEX in AREA'RANGE
    loop
        --<Move message into AREA, blank line feeds>
        if
         CD --<............................>
        then
         ....
        else
         ....
        end if;
    end loop;
    ....
end GET_TEXT;
```

```
procedure GET_TEXT (A_MESSAGE : in STRING) is
--<Build AREA from incoming Message>
    type TEXT is array (1..100) of CHARACTER;
    AREA : TEXT;
begin
    for
        INDEX in AREA'RANGE
    loop
        --<Move message into AREA, blank line feeds>
        if
         CD --<line feed character>
        then
         AREA (INDEX) := '  ';
        else
         AREA (INDEX) := MESSAGE (INDEX);
        end if;
    end loop;
end GET_TEXT;
```

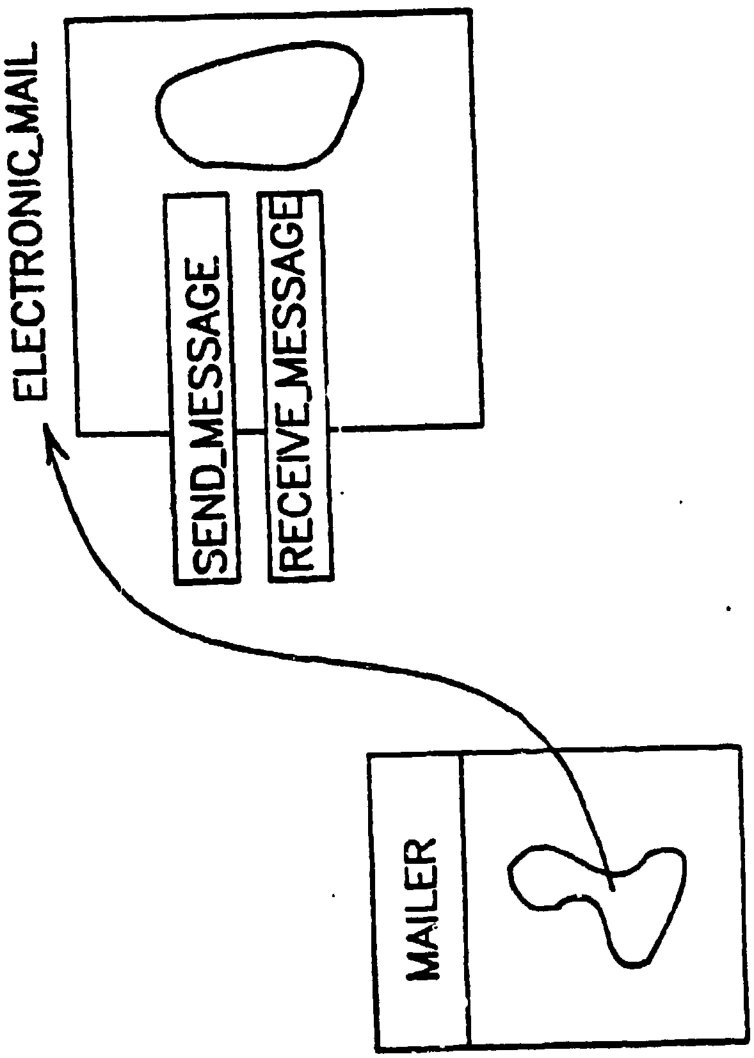
ARCHITECTURAL DESIGN

Ada Software System Structuring Tools

- Packages
- Tasks
- Subprograms
- Separate compilation of specification and body

```
package ELECTRONIC_MAIL is

  procedure SEND_MESSAGE (MESSAGE : in STRING);
  procedure RECEIVE_MESSAGE (MESSAGE : out STRING);

end ELECTRONIC_MAIL;
```

```
with ELECTRONIC_MAIL;
use  ELECTRONIC_MAIL;
procedure MAILER is

  MY_REPLY : STRING (1..17);

begin

  SEND_MESSAGE ("Here is a message");
  RECEIVE_MESSAGE (MY_REPLY);

end MAILER;
```

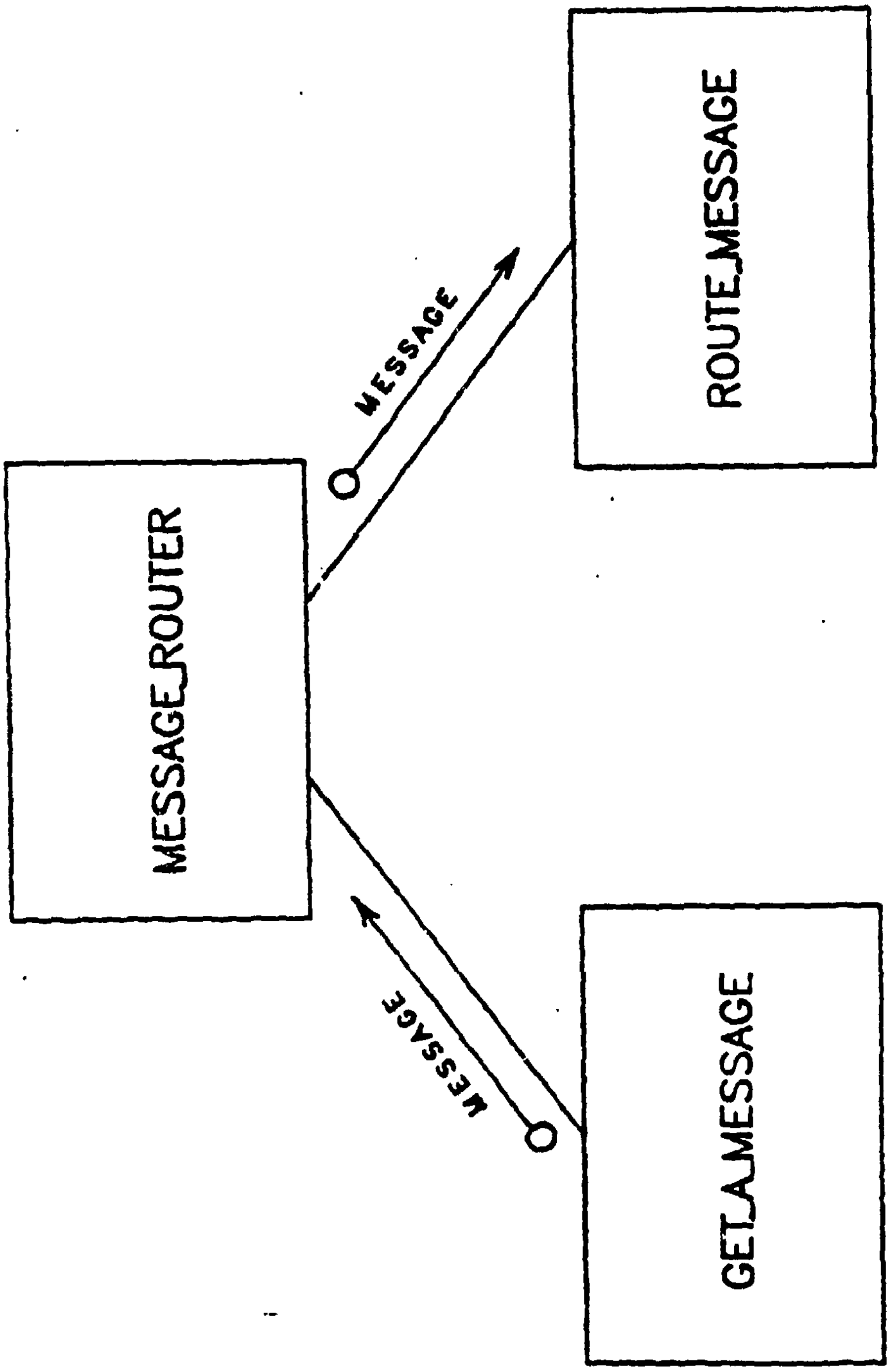

ELECTRONIC_MAIL
SEND_MESSAGE
RECEIVE_MESSAGE
MAILER

ARCHITECTURAL DESIGN

Ada Software System Structuring Tools

- Packages
- Tasks
- Subprograms
- Separate compilation of specification and body
- Subunits

```
procedure MESSAGE_ROUTER is

  MESSAGE : STRING (1 .. 17);

  procedure GET_A_MESSAGE (MESSAGE : out STRING) is separate;
  procedure ROUTE_MESSAGE (MESSAGE : in STRING) is separate;

begin

  loop

    GET_A_MESSAGE (MESSAGE);
    ROUTE_MESSAGE (MESSAGE);

  end loop;

end MESSAGE_ROUTER;
```


MESSAGE_ROUTER
ROUTE_MESSAGE
GET_A_MESSAGE
MESSAGE
MESSAGE

```
separate (MESSAGE_ROUTER)
procedure GET_A_MESSAGE (MESSAGE : out STRING) is

begin

   MESSAGE (1 .. 12) := "Test Message";
   MESSAGE (13 .. MESSAGE'LAST) :=
                        (13 .. MESSAGE'LAST => ' ');

end GET_A_MESSAGE;
```

```
separate (MESSAGE_ROUTER)
procedure ROUTE_MESSAGE (MESSAGE : in STRING) is

begin

    -- Implementation of subprogram goes here
    --     (currently stubbed)

    null;

end ROUTE_MESSAGE;
```

ARCHITECTURAL DESIGN

Interface Definition

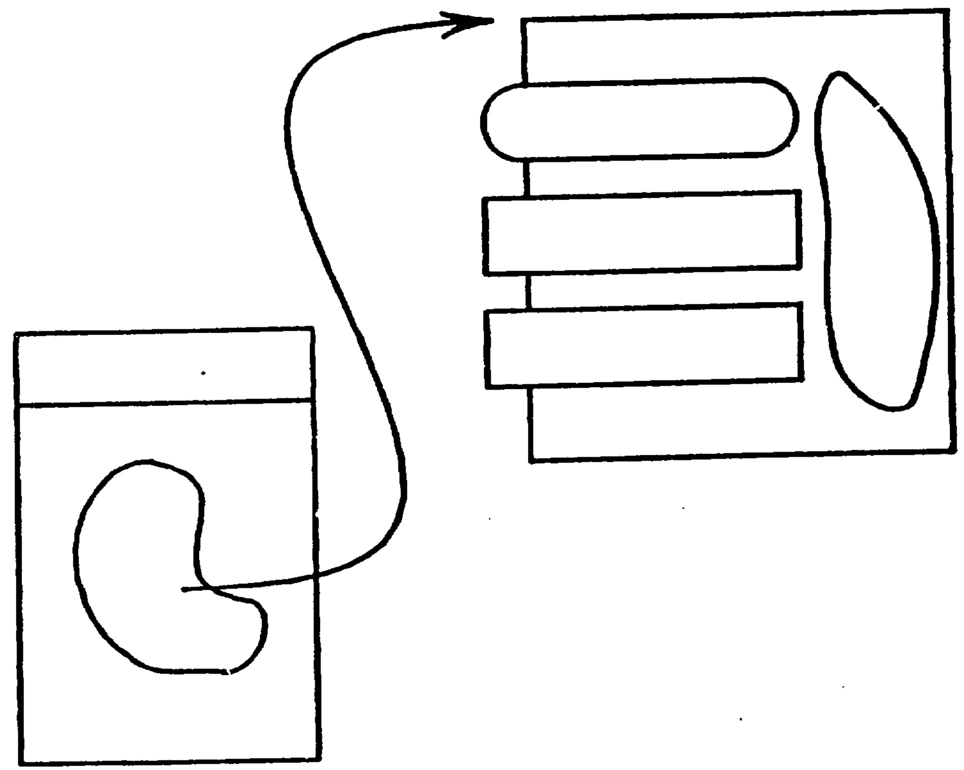
- Strong typing
- Parameters

DETAILED DESIGN

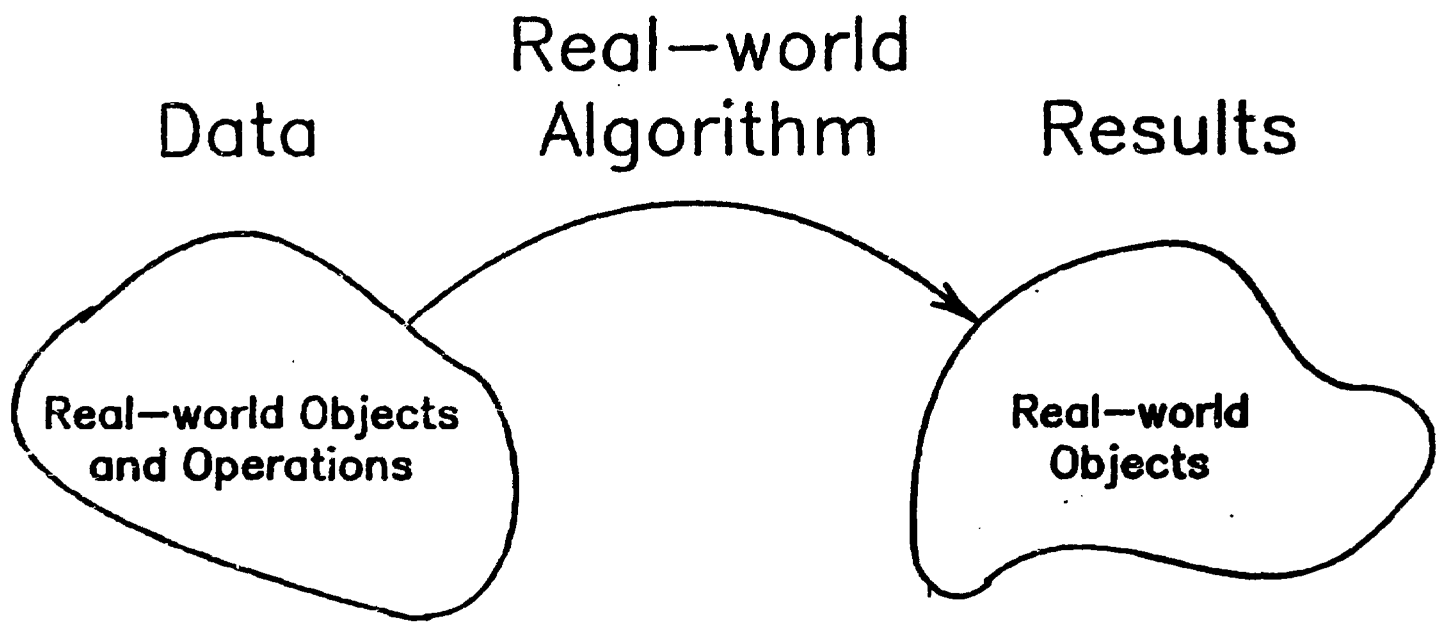
- Rich typing structures
- Control structures

OBJECT ORIENTED DESIGN

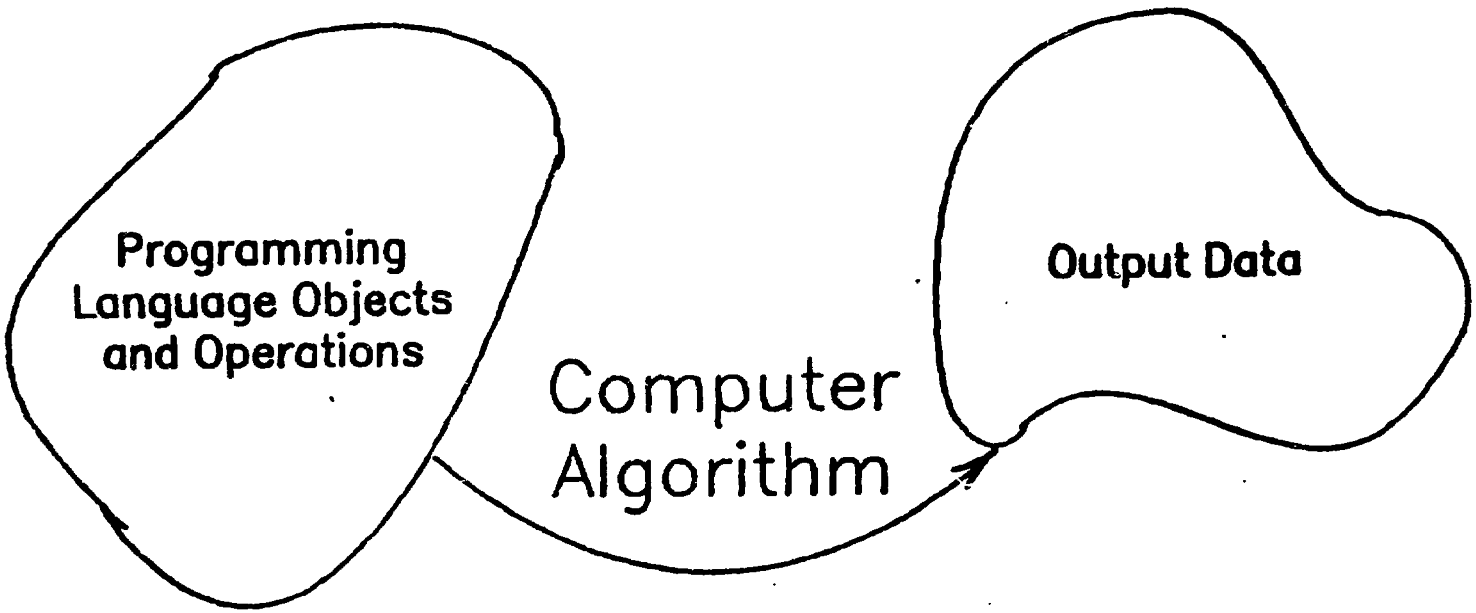
O O D

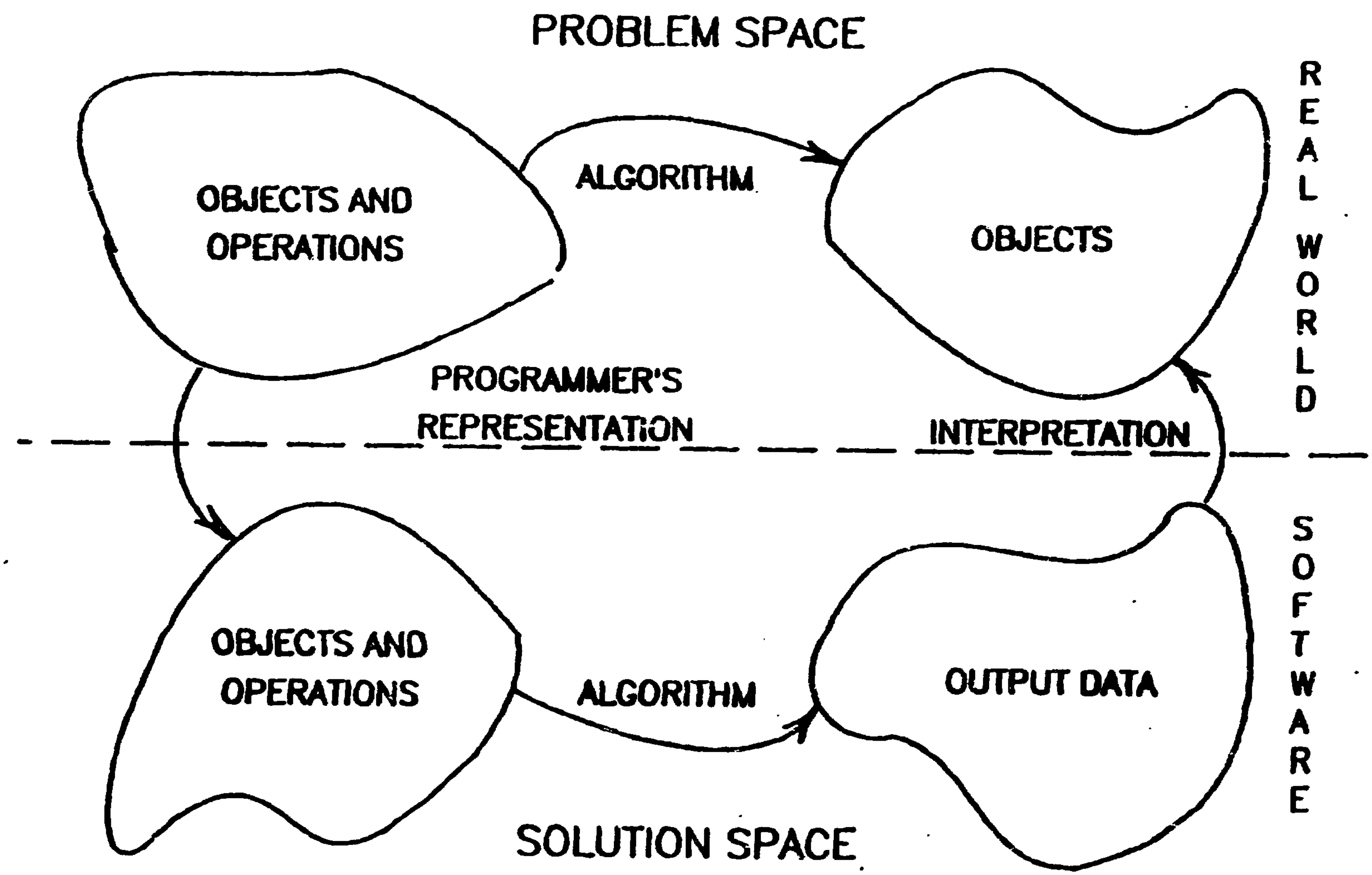


PROBLEM SPACE

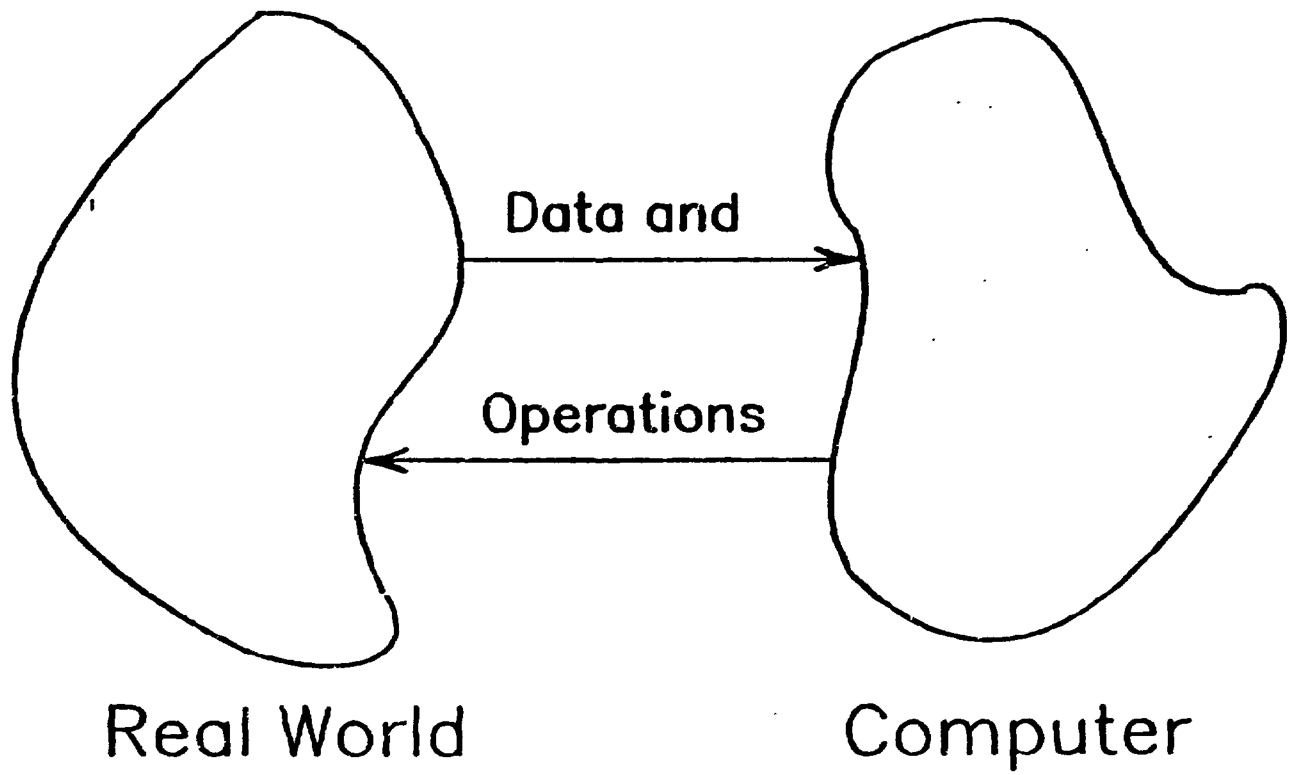


SOLUTION SPACE




PROBLEM SPACE
OBJECTS AND OPERATIONS
ALGORITHM
OBJECTS
REAL WORLD
PROGRAMMER'S REPRESENTATION
INTERPRETATION
SOFTWARE
OBJECTS AND OPERATIONS
ALGORITHM
OUTPUT DATA
SOLUTION SPACE

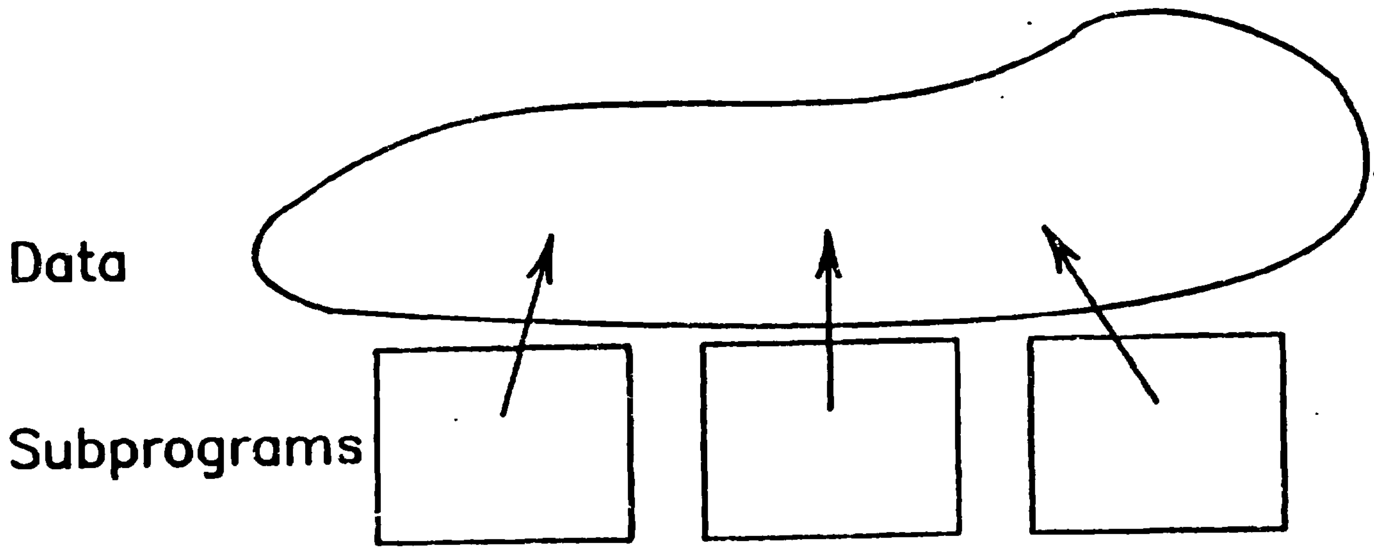
NEED



THE TOPOLOGY OF PROGRAMMING LANGUAGES AFFECTS THE DESIGN OF SOFTWARE SYSTEMS

* Developed prior to modern design methodologies
* Simple structures for simple problems
* Generally action oriented rather than architecture oriented

TOPOLOGY OF FIRST AND SECOND GENERATION LANGUAGES

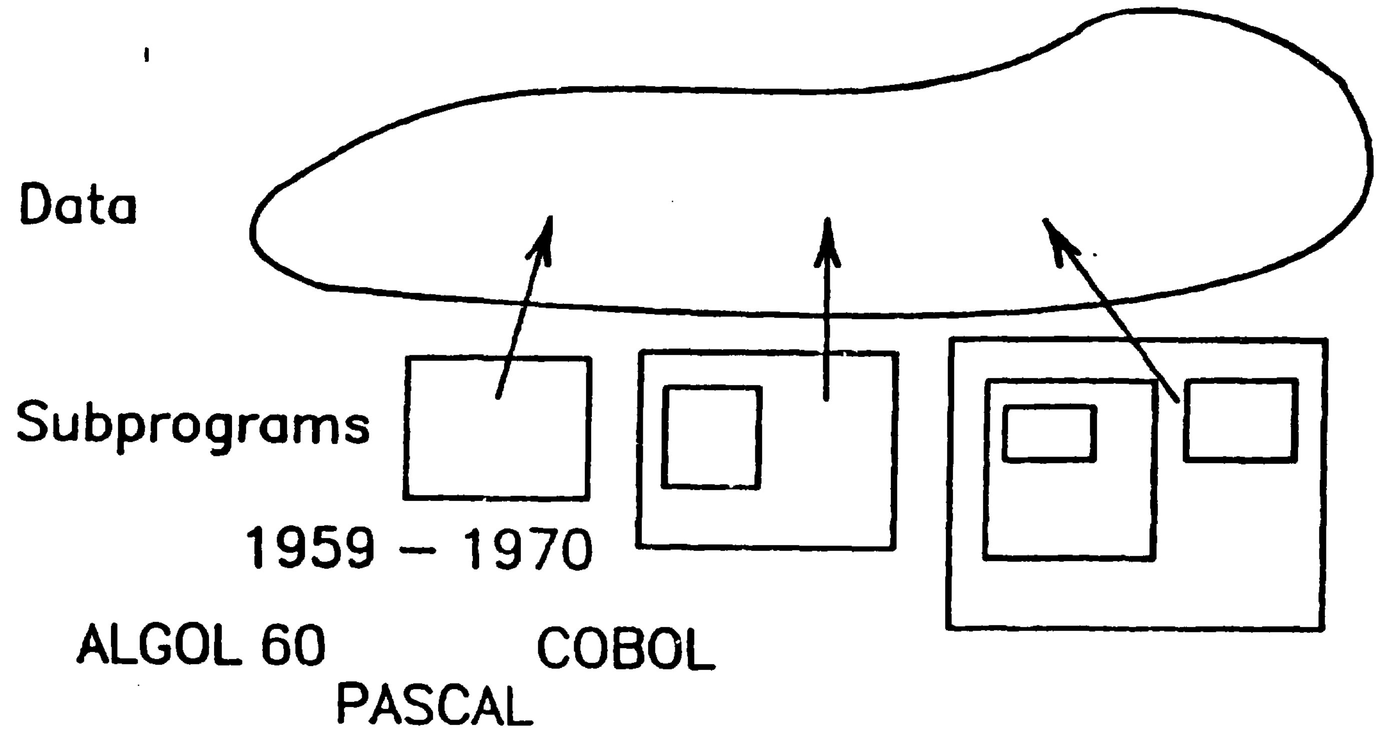


1954 – 1958

FORTRAN I
FLOWMATIC

ALGOL 58
IPL V

TOPOLOGY OF SECOND AND THIRD GENERATION LANGUAGES

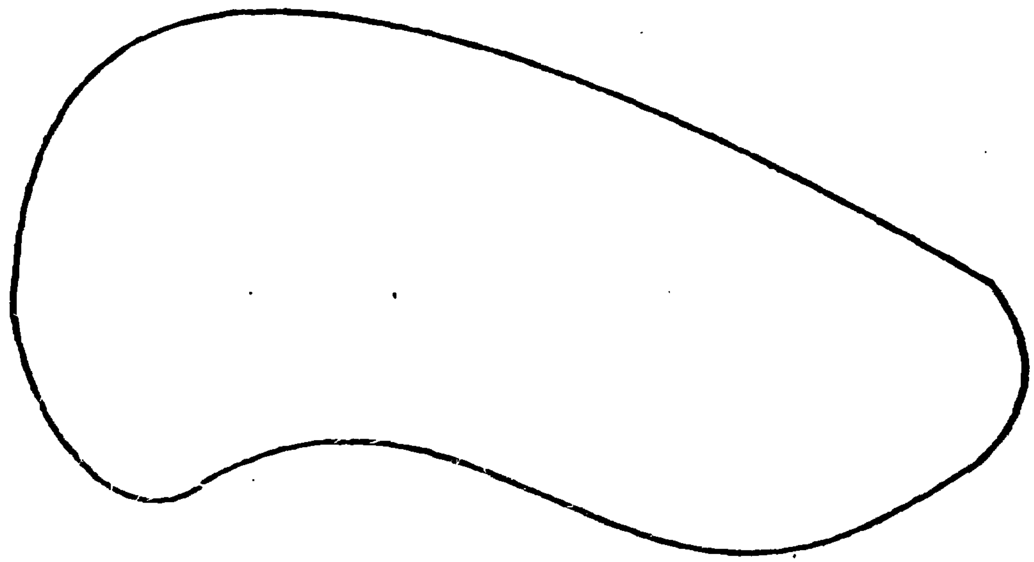


EMBEDDED SYSTEM CHARACTERISTICS

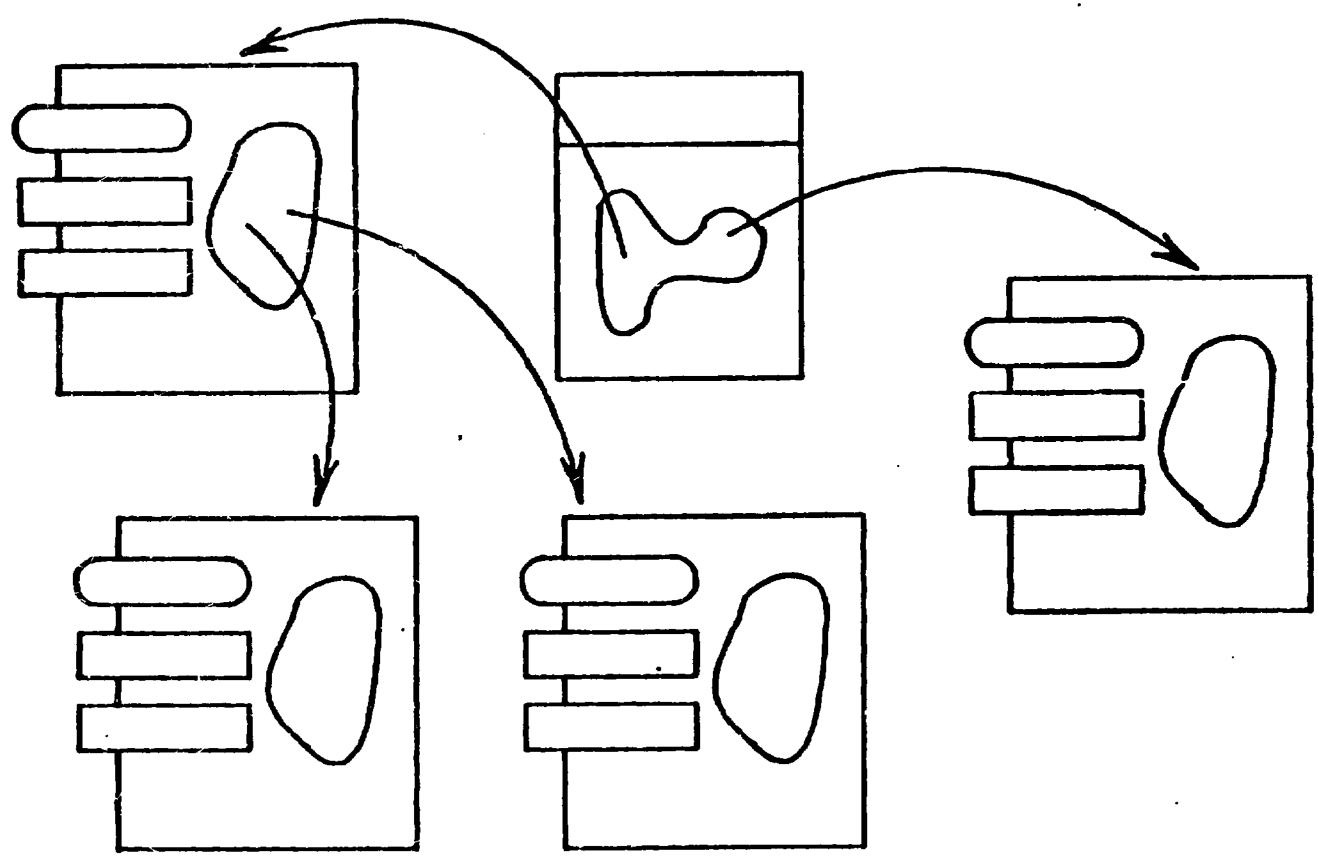
* LARGE
* LONG—LIVED
* CONTINUOUS CHANGE
* PHYSICAL CONSTRAINTS
* HIGH RELIABILITY

Use mostly assembly languages ...

TOPOLOGY OF ASSEMBLY LANGUAGES



Topology of Ada



OBJECT ORIENTED DESIGN

– Supports Software Engineering Principles

* Abstraction
* Information Hiding
* Modularity
* Localization

OBJECT ORIENTED DESIGN

– Helps Students:

* Think Ada
* Learn SE Principles
* Overcome Syntax Limitations

Criteria for Decomposing a System into Modules

* **Top Down Structured Design**

  Each module denotes a major step in the overall process

* **Data Structure Design**

  Input data structure mapped to output data structure

* **Parnas Decomposition**

  Each module hides a design decision

* **Object Oriented Design**

  Each module in the system denotes an object or class of objects from problem space

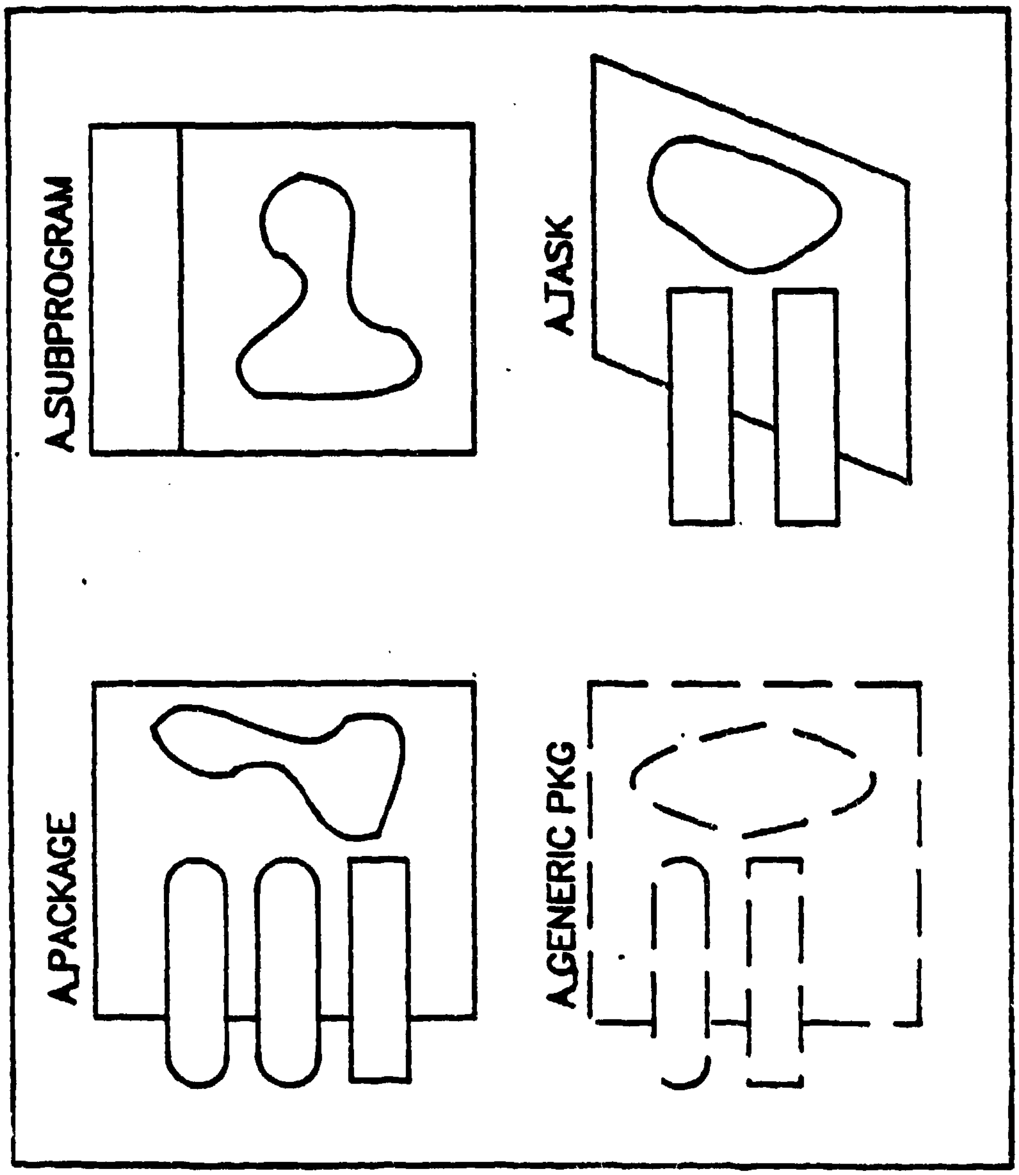
OBJECT ORIENTED DESIGN

GENERAL APPROACH

* Identify objects and their attributes
* Identify operations on and operations required of each object
* Establish the interfaces of each object and its visibility in relation to other objects
* Implement each object

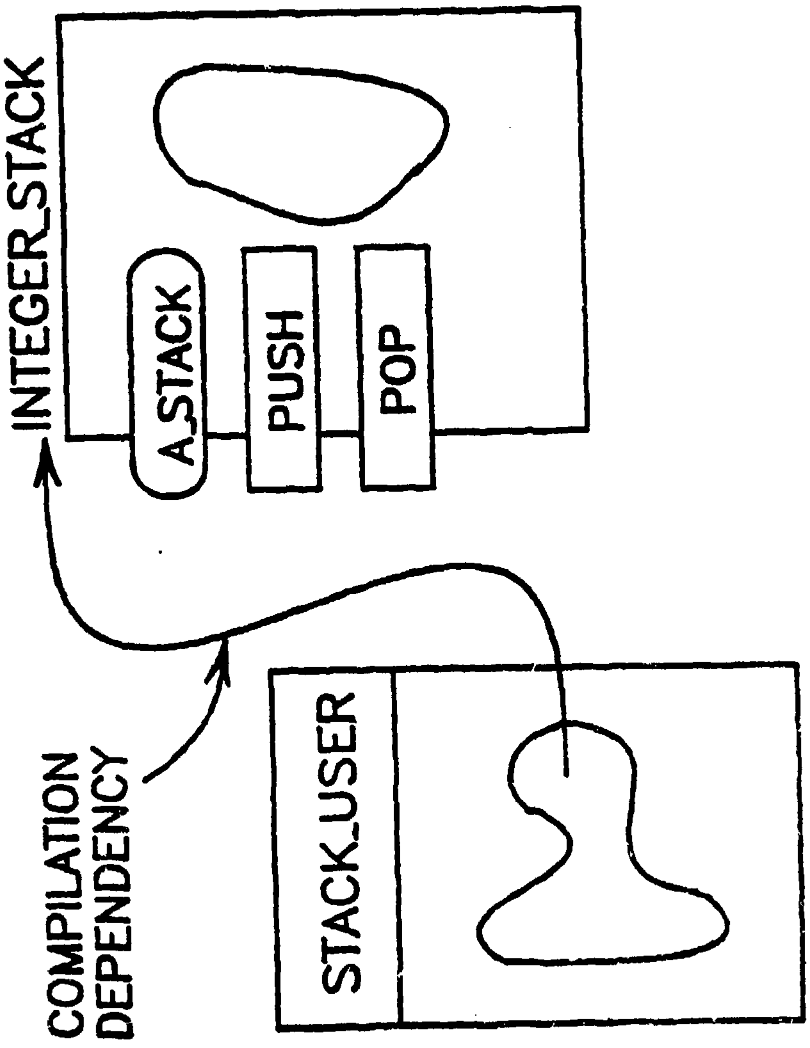
CHARACTERISTICS OF AN OBJECT

* Has a state
* Has a set of operations defined for it and used by it
* Instance of some class of objects
* Has a name
* Has restricted visibility of and by other objects


A_SUBPROGRAM
A_TASK
A_PACKAGE
A_GENERIC PKG

Expressing OOD in Ada

* Objects are denoted by instances of private or limited private types
* Classes of objects are denoted by packages which export private or limited private types
* Variables serve as names of objects
* Visibility is statically defined


INTEGER_STACK
A_STACK
PUSH
POP
COMPILATION DEPENDENCY
STACK_USER

```
package INTEGER_STACK is

   -- This package defines an INTEGER stack
   -- resource

   type A_STACK is limited private;

   procedure PUSH (ITEM : in INTEGER;
                     ON : in out A_STACK);

   procedure POP (ITEM : out INTEGER;
                OFF_OF : in out A_STACK);
private

   -- Implementation of A_STACK is
   -- defined here
   . . .

end INTEGER_STACK;
```

```
with INTEGER_STACK;
use  INTEGER_STACK;
procedure STACK_USER is

   MY_STACK     : A_STACK;
   YOUR_STACK : A_STACK;
   A_NUMBER     : INTEGER

begin

   PUSH (50, ON => MY_STACK);
   PUSH (100, ON => YOUR_STACK);
   POP (A_NUMBER, OFF_OF => YOUR_STACK);

end STACK_USER;
```

```
package B_R is

    type NUMBERS is range 0 .. 99;

    procedure TAKE (A_NUMBER : out NUMBERS);
    procedure SERVE (NUMBER : in NUMBERS);
    function  NOW_SERVING return NUMBERS;

end B_R;

package body B_R is

    SERV_A_MATIC : NUMBERS := 1;

    procedure TAKE (A_NUMBER : out NUMBERS) is
    begin
        A_NUMBER     := SERV_A_MATIC;
        SERV_A_MATIC := SERV_A_MATIC + 1;
    end TAKE;

    procedure SERVE (NUMBER : in NUMBERS) is separate;

    function  NOW_SERVING return NUMBERS is separate;

end B_R;
```

```
with B_R;
use  B_R;
procedure ICE_CREAM is

    YOUR_NUMBER : NUMBERS;

begin

    TAKE (YOUR_NUMBER);
    loop

        if NOW_SERVING = YOUR_NUMBER then
            SERVE (YOUR_NUMBER);
            exit;
        end if;

    end loop;

end ICE_CREAM;
```

```
with B_R;
use  B_R;
procedure ICE_CREAM is

    YOUR_NUMBER : NUMBERS;

begin

    TAKE (YOUR_NUMBER);
    loop

        if NOW_SERVING = YOUR_NUMBER then
            SERVE (YOUR_NUMBER);
            exit;
        else
            YOUR_NUMBER := YOUR_NUMBER - 1;
        end if;

    end loop;

end ICE_CREAM;
```

```
package B_R is

    type NUMBERS is private;

    procedure TAKE (A_NUMBER : out NUMBERS);
    procedure SERVE (NUMBER : in NUMBERS);
    function  NOW_SERVING return NUMBERS;

private

    type NUMBERS is range 0 .. 99;

end B_R;
```

```
with B_R;
use  B_R;
procedure ICE_CREAM is

    YOUR_NUMBER : NUMBERS;

begin

    TAKE (YOUR_NUMBER);
    loop

        if NOW_SERVING = YOUR_NUMBER then
            SERVE (YOUR_NUMBER);
            exit;
        else
            YOUR_NUMBER := NOW_SERVING;
        end if;

    end loop;

end ICE_CREAM;
```

```
package B_R is

    type NUMBERS is limited private;

    procedure TAKE (A_NUMBER : out NUMBERS);
    procedure SERVE (NUMBER : in NUMBERS);
    function  NOW_SERVING return NUMBERS;
    function  "=" (LEFT, RIGHT : in NUMBERS)
                   return BOOLEAN;
    function  CLOSE_ENOUGH (A_NUMBER : in NUMBERS)
                   return BOOLEAN;
private

    type NUMBERS is range 0 .. 99;

end B_R;
```

```
with B_R;
use  B_R;
procedure ICE_CREAM is

     YOUR_NUMBER : NUMBERS;

     procedure GO_TO_DQ is separate;

begin

     TAKE (YOUR_NUMBER);
     if NOW_SERVING = YOUR_NUMBER then
          SERVE (YOUR_NUMBER);
     elsif CLOSE_ENOUGH (YOUR_NUMBER) then

          while NOW_SERVING /= YOUR_NUMBER loop
               null;                          -- wait your turn
          end loop;
          SERVE (YOUR_NUMBER);

     else
          GO_TO_DQ;
     end if;

end ICE_CREAM;
```

OBJECT ORIENTED DESIGN STEPS

1.0 Define the problem

2.0 Develop an informal strategy

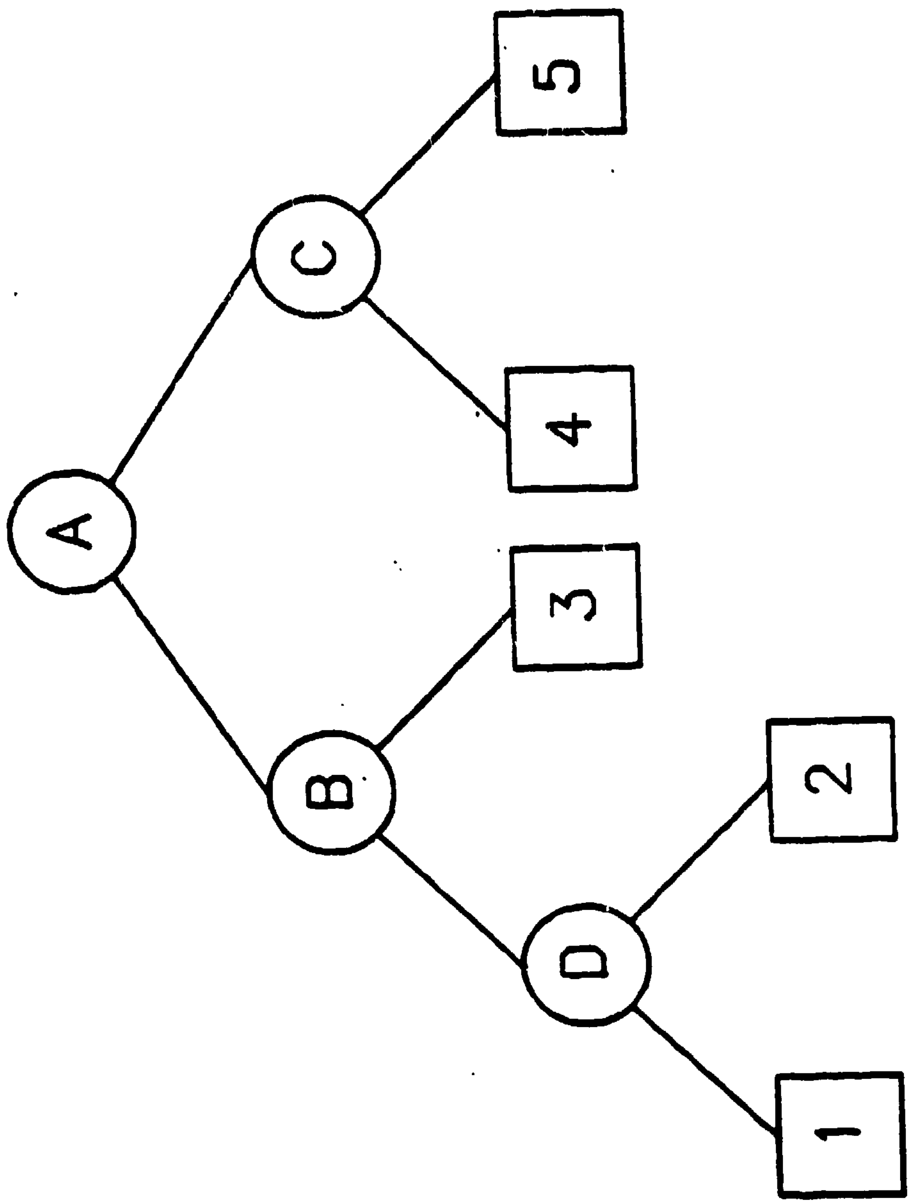
3.0 Formalize the strategy

- Identify objects of interest and their attributes
- Identify operations on the objects
- Establish interfaces among the objects
- Implement the objects and operations

1.0 Define the problem

1.1 State problem in a single sentence

Given a binary tree, count its leaves.



A
B
C
D
1
2
3
4
5

COUNTING LEAVES

- IF THE TREE IS A LEAF

  NUMBER_OF_LEAVES (TREE) = 1

- IF THE TREE CONSISTS OF
  TWO SUBTREES

  NUMBER_OF_LEAVES (TREE) =

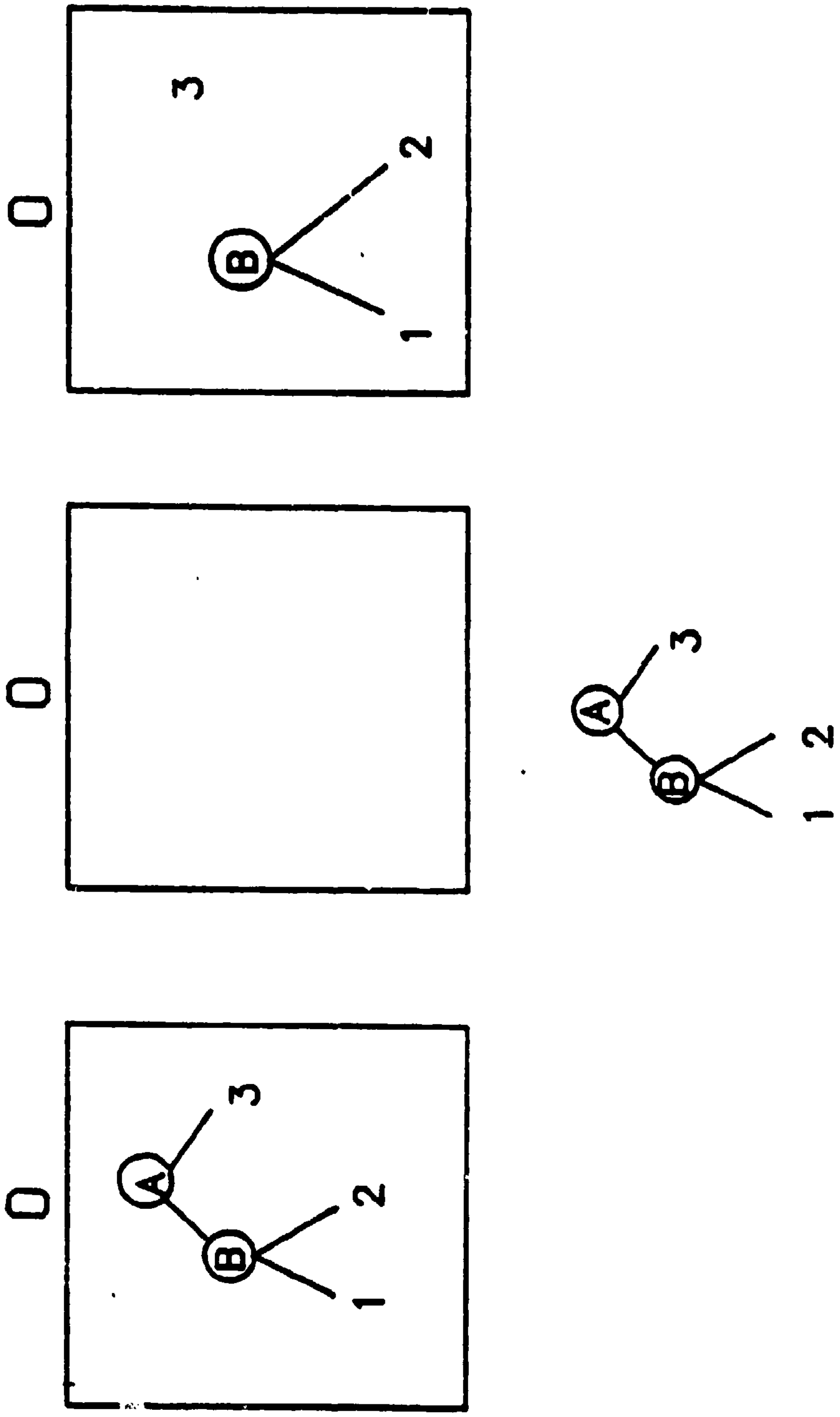
  NUMBER_OF_LEAVES (LEFT_SUBTREE) +
  NUMBER_OF_LEAVES (RIGHT_SUBTREE)

2.0 Develop an informal strategy

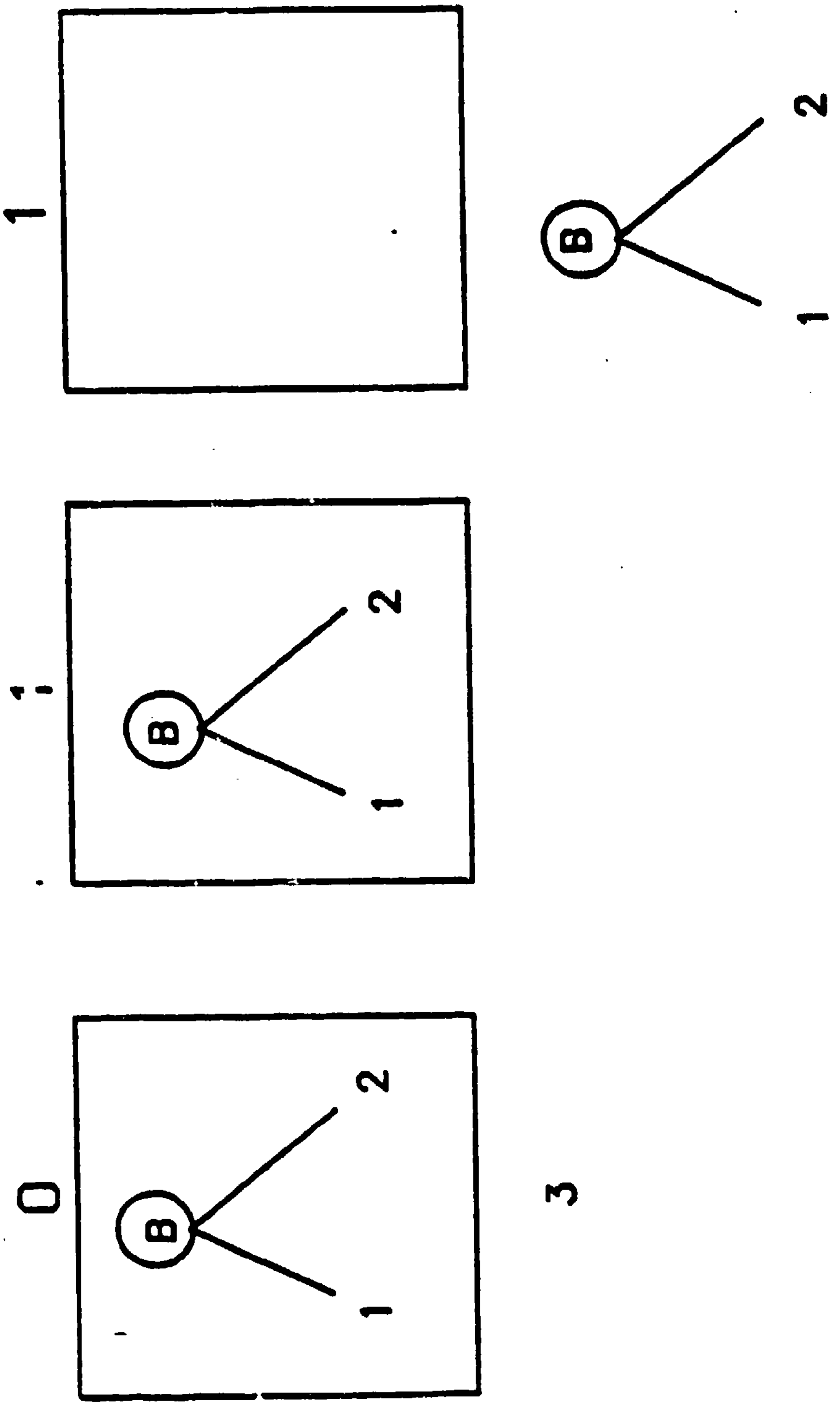
2.1 Establish strategy perspective

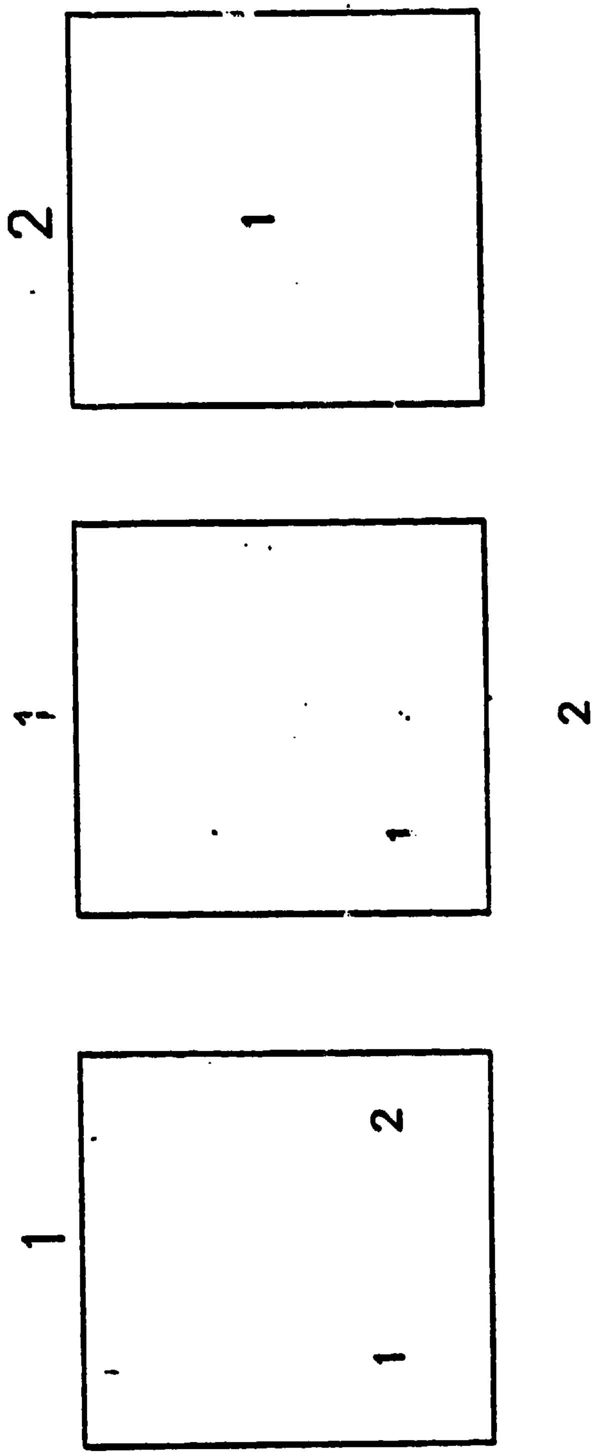
The perspective will be from the system viewpoint.

Keep a pile of the parts of the tree that have not yet been counted. Initially, get a tree and put it on the empty pile; the count of the leaves is initially set to zero. As long as the pile is not empty, repeatedly take a tree off the pile and examine it. If the tree consists of a single leaf, then increment the leaf counter and throw away that tree. If the tree is not a single leaf but instead consists of two subtrees, split the tree into its left and right subtrees and put them back on the pile. Once the pile is empty, display the count of the leaves.



0
B
1
2
3
0
A
B
1
2
3
0
A
B
1
2
3


0
1
1
3
B
1
2


2
1
1
2
1
1
2
1

Keep a <u>pile</u> of the <u>parts of the tree</u> that have not yet been counted. Initially, get a <u>tree</u> and put <u>it</u> on the <u>empty pile</u>; the <u>count of the leaves</u> is initially set to zero. As long as the <u>pile</u> is not empty, repeatedly take a <u>tree</u> off the <u>pile</u> and examine <u>it</u>. If the <u>tree</u> consists of a <u>single leaf</u>, then increment the <u>leaf counter</u> and throw away that <u>tree</u>. If the <u>tree</u> is not a <u>single leaf</u> but instead consists of <u>two subtrees</u>, split the <u>tree</u> into its <u>left</u> and <u>right subtrees</u> and put <u>them</u> back on the <u>pile</u>. Once the <u>pile</u> is empty, display the <u>count of the leaves</u>.

OBJECTS OF INTEREST

TREE, LEFT_SUBTREE, RIGHT_SUBTREE

PILE

LEAF_COUNT

Keep a pile of the parts of the tree that have not yet been counted. Initially, get a tree and put it on the empty pile; the count of the leaves is initially set to zero. As long as the pile is not empty, repeatedly take a tree off the pile and examine it. If the tree consists of a single leaf, then increment the leaf counter and throw away that tree. If the tree is not a single leaf but instead consists of two subtrees, split the tree into its left and right subtrees and put them back on the pile. Once the pile is empty, display the count of the leaves.

OBJECTS AND OPERATIONS

TREE, LEFT_SUBTREE, RIGHT_SUBTREE

GET_INITIAL
IS_SINGLE_LEAF
THROW_AWAY
SPLIT

PILE

IS_EMPTY
PUT_INITIAL
TAKE_OFF
PUT

LEAF_COUNT

SET_TO_ZERO
INCREMENT
DISPLAY

TREE_PACKAGE

TREE, LEFT_SUBTREE, RIGHT_SUBTREE

GET_INITIAL
IS_SINGLE_LEAF
THROW_AWAY
SPLIT

PILE_PACKAGE

PILE

IS_EMPTY
PUT_INITIAL
TAKE_OFF
PUT

COUNTER_PACKAGE

LEAF_COUNT

SET_TO_ZERO
INCREMENT
DISPLAY

TREE_PACKAGE

TREE_TYPE

GET_INITIAL
IS_SINGLE_LEAF
THROW_AWAY
SPLIT

PILE_PACKAGE

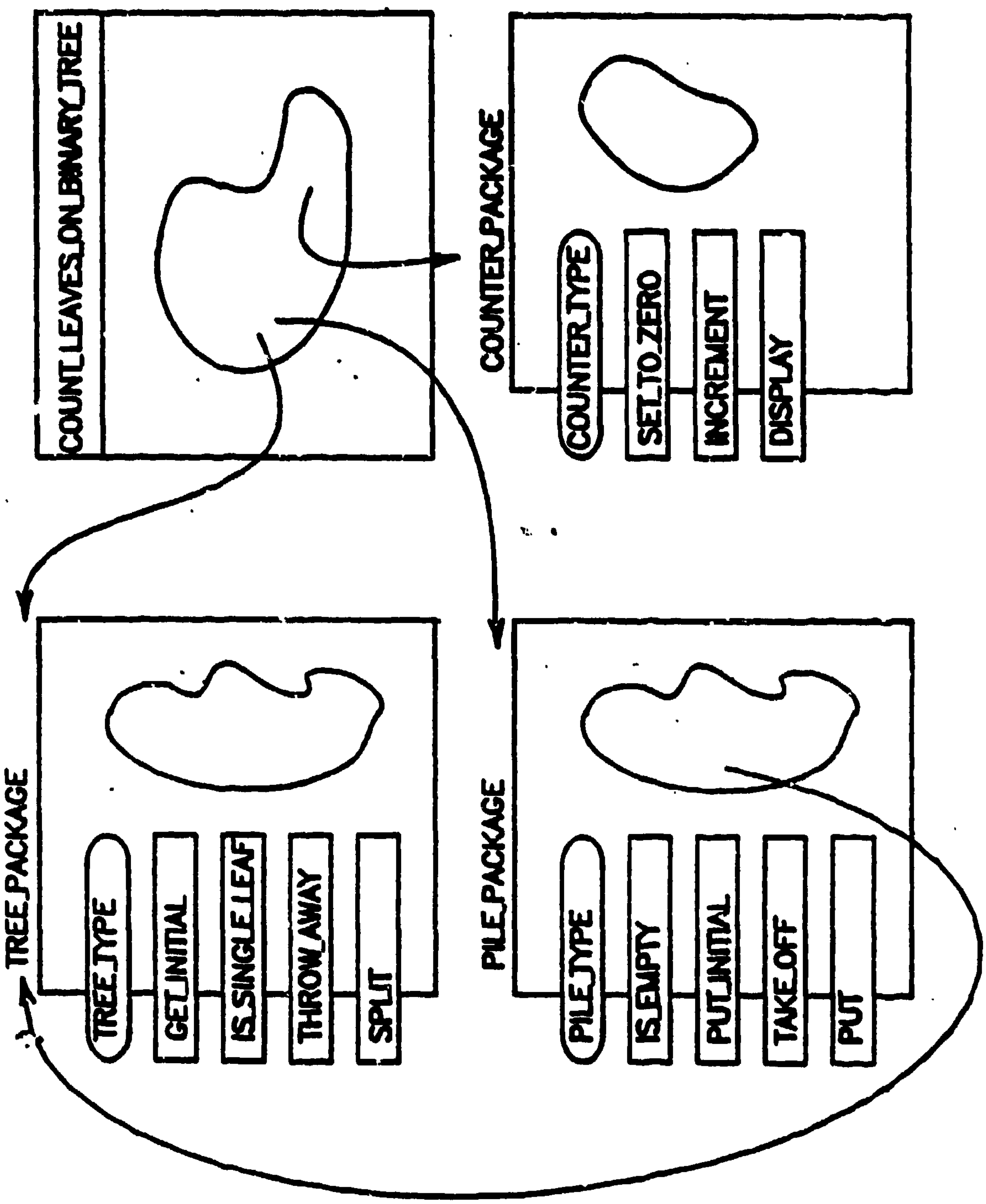
PILE_TYPE

IS_EMPTY
PUT_INITIAL
TAKE_OFF
PUT

COUNTER_PACKAGE

COUNTER_TYPE

SET_TO_ZERO
INCREMENT
DISPLAY


COUNT_LEAVES_ON_BINARY_TREE
COUNTER_PACKAGE
COUNTER_TYPE
SET_TO_ZERO
INCREMENT
DISPLAY
TREE_PACKAGE
TREE_TYPE
GET_INITIAL
IS_SINGLE_LEAF
THROW_AWAY
SPLIT
PILE_PACKAGE
PILE_TYPE
IS_EMPTY
PUT_INITIAL
TAKE_OFF
PUT

```
package COUNTER_PACKAGE is
  type COUNTER_TYPE is limited private;
  procedure DISPLAY (COUNTER : in COUNTER_TYPE);
  procedure INCREMENT (COUNTER : in out COUNTER_TYPE);
  procedure ZERO (COUNTER : out COUNTER_TYPE);
private
  . . .
end COUNTER_PACKAGE;
```

```
package TREE_PACKAGE is

   type TREE_TYPE is private;

   procedure GET_INITIAL (TREE : out TREE_TYPE);

   function IS_SINGLE_LEAF (TREE : in TREE_TYPE)
                                            return BOOLEAN;

   procedure SPLIT (TREE : in out TREE_TYPE;
                    LEFT_INTO : out TREE_TYPE;
                    RIGHT_INTO : out TREE_TYPE);

   procedure THROW_AWAY (TREE : in out TREE_TYPE);

private
   . . .
end TREE_PACKAGE;
```

```
with TREE_PACKAGE;
package PILE_PACKAGE is

  type PILE_TYPE is limited private;
  function IS_EMPTY (PILE : in PILE_TYPE) return BOOLEAN;
  procedure PUT (TREE : in out TREE_PACKAGE.TREE_TYPE;
                 ON : in out PILE_TYPE);
  procedure PUT_INITIAL (TREE : in out TREE_PACKAGE.TREE_TYPE;
                         ON : in out PILE_TYPE);
  procedure TAKE_OFF (TREE : out TREE_PACKAGE.TREE_TYPE;
                      OFF : in out PILE_TYPE);

private

   . . .

end PILE_PACKAGE;
```

```
with COUNTER_PACKAGE, PILE_PACKAGE, TREE_PACKAGE;
use  COUNTER_PACKAGE, PILE_PACKAGE, TREE_PACKAGE;
procedure COUNT_LEAVES_ON_BINARY_TREE is

   LEAF_COUNT : COUNTER_TYPE;
   LEFT_SUBTREE : TREE_TYPE;
   RIGHT_SUBTREE : TREE_TYPE;
   PILE : PILE_TYPE;
   TREE : TREE_TYPE;
```

```
begin
   GET_INITIAL (TREE);
   PUT_INITIAL (TREE, PILE);
   ZERO (LEAF_COUNT);
   while not IS_EMPTY (PILE) loop
     TAKE_OFF (TREE, PILE);
     if IS_SINGLE_LEAF (TREE) then
        INCREMENT (LEAF_COUNT);
        THROW_AWAY (TREE);
     else
        SPLIT (TREE, LEFT_SUBTREE, RIGHT_SUBTREE);
        PUT (LEFT_SUBTREE, PILE);
        PUT (RIGHT_SUBTREE, PILE);
     end if;
   end loop;
   DISPLAY (LEAF_COUNT);
 end COUNT_LEAVES_ON_BINARY_TREE;
```

LANGUAGE EXTENSION

- Assume an adequate set of control structures (loop, case, if–then–else, while, for, etc.)
- Identify objects and operations from the problem space
- Extend the PDL by adding the objects and operations
- Solve the problem using the extended language
- Give yourself the language
  * Refine the operations
  * Implement the objects
- Continue the process until all objects and operations have been realized in Ada

OBJECT ORIENTED DESIGN

1.0 Define the problem

1.1 State the problem to be solved in a single sentence

PURPOSE:

- To gain a clear, unified understanding of the problem by all interested parties
- Answers the question: " What are we trying to do? "

GUIDELINES:

- Write a single, clear and concise sentence
- Ensure it is grammatically correct
- All problems can be stated in a single sentence

1.2 Gather, organize, and analyze information about the problem

PURPOSE:

- To gather all information pertinent to understanding and solving the problem

GUIDELINES:

- Gather all pertinent information
- Can use formal analysis tools
- Include all levels of detail
- Organize information into logical groupings

2.0 Develop an Informal Strategy

2.1 Establish an appropriate perspective for the strategy

PURPOSE:

- Gives a starting point for the informal strategy

2.2 Write a solution to the problem in a single paragraph

PURPOSE:

- Establishes a plan of attack

- Brings out an appropriate level of abstraction for solution

- Unifies problem understanding

GUIDELINES:

- Use 7 plus or minus 2 sentences (Hrair limit)

- Write simple, clear and concise sentences

- Grammatically correct

- Place emphasis on writing a coherent paragraph, not just the objects and operations

- Use a uniform level of abstraction

- Use language appropriate for the level of abstraction and viewpoint

- The informal strategy should be a complete solution to the problem

- Should be a description of solution, not necessarily an algorithm

- Doesn't have to be a prize winning novel

3.0 **Formalize the strategy**

3.1 Identify Objects of Interest and their attributes

PURPOSE:

- To determine the abstract objects in the problem

- To determine the characteristics of the abstract objects

- To determine sets of values

3.1.1 **Underline all nouns, pronouns and noun clauses (with modifying adjectives) in the paragraph**

PURPOSE:

- To create a list of all potential objects

GUIDELINES:

- A noun clause is a clause that acts as a noun; i.e., count of the leaves

- Underline all nouns

<u>3.1.2</u> Place each unique noun, pronoun or noun clause in the column labeled OBJECT

PURPOSE:

- To separate potential abstract objects

<u>3.1.3</u> Identify all nouns referring to the same object

PURPOSE:

- To unclutter the name space

<u>3.1.4</u> Determine the space of each object and write it in a column labeled SPACE

PURPOSE:

- Determination of objects of interest

GUIDELINES:

- Solution space if needed to solve problem

- Problem space if needed to describe problem, but not to solve it

<u>3.1.5</u> List appropriate attributes of the objects

PURPOSE:

- Determine characteristics of abstract objects

GUIDELINES:

- From adjectives

- From gathered information

<u>3.1.6</u> Select an Ada identifier for each object in the solution space

<u>3.1.7</u> Group objects that are of the same type

PURPOSE:

- To visualize the structural equivalence of similar objects

- To facilitate the definition of types

- To track abstract objects later

3.2 Identify Operations on the objects

PURPOSE:

- To determine sets of operations

3.2.1 Underline all verbs, verb phrases and predicates in the informal strategy

PURPOSE:

- Determine potential abstract operations

GUIDELINES:

- Predicate indicates some sort of test followed by a change in control; usually a form of the verb "to be"

- Also underline adverbs

- Adverbs may be separated from verbs

3.2.2 Place each unique verb, verb phrase or predicate in a column labeled OPERATION

PURPOSE:

- Separate potential operations

3.2.3 Identify all verbs, verb phrases and predicates referring to the same operation

PURPOSE:

- To unclutter the operation-space

3.2.4 Determine the space of each operation and write it in a column labeled SPACE

PURPOSE:

- Identification of abstract operations

3.2.5 Determine the object operated on by each operation and write it in column labeled OBJECT

PURPOSE:

- Determine what object is being operated on for each operation

- To associate operations later with types

- To adhere to traditional design principles of coupling and cohesion

GUIDELINES:

- All operations operate on one object

- For an operation to operate on an object, the operation must be aware of the object's underlying representation

3.2.6 Identify other objects associated with the operation

PURPOSE:

- To use in defining parameters

- To use in defining interfaces

3.2.7 Select an Ada identifier for each operation and write it in a column labeled IDENTIFIER

PURPOSE:

- To formalize the abstract operations

3.3 Establish interfaces among the objects

PURPOSE:

- To determine abstract data types

- To establish software resources

- To determine compilation dependencies

- To precisely define interfaces between resources

3.3.1 Group objects with operations together in one place

PURPOSE:

- To visualize logical abstract data types

- To ease transition to program units

3.3.2 Associate a name with each grouping

PURPOSE:

- To formalize data types

- To ease transition to program units

3.3.3 Define types for each grouping

PURPOSE:

- Determination of abstract data types

3.3.4 Transform each grouping into its appropriate program unit symbol

PURPOSE:

- To visualize software resources

- To visualize the interfaces between software resources

3.3.5 Show access needs between program units

GUIDELINES:

- Be sure to use associated objects as keys

3.3.6 Develop Ada PDL for the Booch-o-grams

PURPOSE:

- Formalize the software system

- Make use of the Ada compiler as a tool

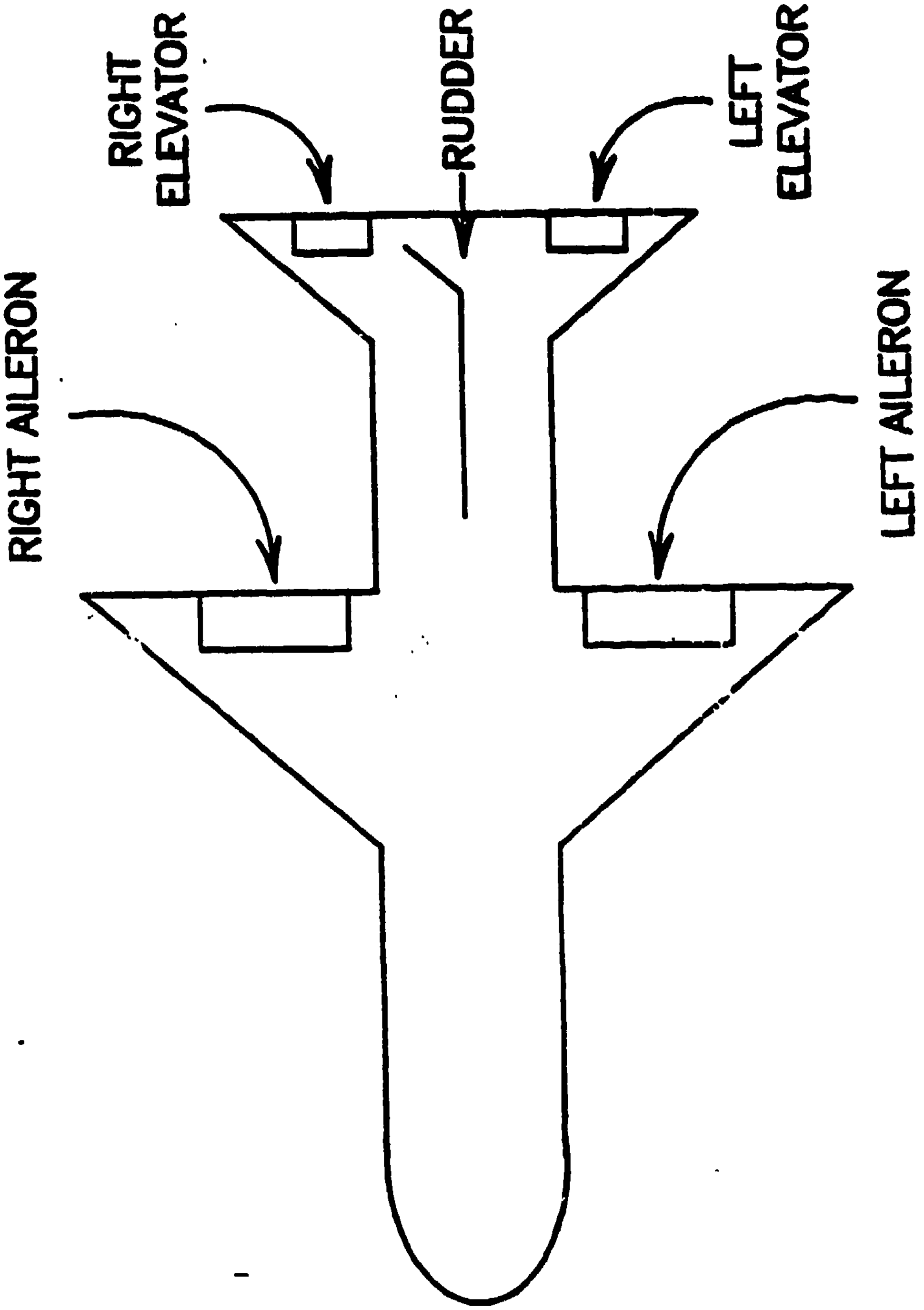
GUIDELINES:

- Be sure to use associated objects as keys for determining parameters

3.4 Implement the Objects and Operations

1.0 Define the problem

Control the flight surfaces
on an aircraft


RIGHT ELEVATOR
RUDDER
LEFT ELEVATOR
RIGHT AILERON
LEFT AILERON

The flight surface control system operates the left aileron, right aileron, elevator and rudder for an aircraft. The aileron controller gets roll amount and direction. If the roll direction is left, then it pivots the left aileron up and the right aileron down the roll amount; otherwise, it pivots the left aileron down and the right aileron up the roll amount. The elevator controller gets pitch amount and direction. If the pitch direction is up, it pivots the elevator up the pitch amount; otherwise, it pivots the elevator down the pitch amount. The rudder controller gets yaw amount and direction. If the yaw direction is left, it pivots the rudder left the yaw amount; otherwise, it pivots the rudder right the yaw amount.

The flight surface control system operates the left aileron, right aileron, elevator and rudder for an aircraft. The aileron controller gets roll amount and direction. If the roll direction is left, then it pivots the left aileron up and the right aileron down the roll amount; otherwise, it pivots the left aileron down and the right aileron up the roll amount. The elevator controller gets pitch amount and direction. If the pitch direction is up, it pivots the elevator up the pitch amount; otherwise, it pivots the elevator down the pitch amount. The rudder controller gets yaw amount and direction. If the yaw direction is left, it pivots the rudder left the yaw amount; otherwise, it pivots the rudder right the yaw amount.

OBJECT	SPACE	IDENTIFIER
flight surface control system	P	
left aileron	S	LEFT_AILERON
right aileron	S	RIGHT_AILERON
elevator	S	ELEVATOR
rudder	S	RUDDER
aircraft	P	
aileron controller	S	AILERON_CONTROLLER
roll amount	S	ROLL_AMOUNT
roll...direction	S	ROLL_DIRECTION
it	S	/
elevator controller	S	ELEVATOR_CONTROLLER
pitch amount	S	PITCH_AMOUNT
pitch...direction	S	PITCH_DIRECTION
it	S	/
rudder controller	S	RUDDER_CONTROLLER
yaw amount	S	YAW_AMOUNT
yaw...direction	S	YAW_DIRECTION
it	S	/

OBJECTS OF INTEREST

LEFT_AILERON, RIGHT_AILERON
AILERON_CONTROLLER
ROLL_AMOUNT
ROLL_DIRECTION
ELEVATOR
ELEVATOR_CONTROLLER
PITCH_AMOUNT
PITCH_DIRECTION
RUDDER
RUDDER_CONTROLLER
YAW_AMOUNT
YAW_DIRECTION

The flight surface control system <u>operates</u> the left aileron, right aileron, elevator and rudder for an aircraft. The aileron controller <u>gets</u> roll amount and direction. If the roll direction <u>is left</u>, then it <u>pivots</u> the left aileron <u>up</u> and the right aileron <u>down</u> the roll amount; otherwise, it <u>pivots</u> the left aileron <u>down</u> and the right aileron <u>up</u> the roll amount. The elevator controller <u>gets</u> pitch amount and direction. If the pitch direction <u>is up</u>, it <u>pivots</u> the elevator <u>up</u> the pitch amount; otherwise, it <u>pivots</u> the elevator <u>down</u> the pitch amount. The rudder controller <u>gets</u> yaw amount and direction. If the yaw direction <u>is left</u>, it <u>pivots</u> the rudder <u>left</u> the yaw amount; otherwise, it <u>pivots</u> the rudder <u>right</u> the yaw amount.

OPERATION	SPACE	OBJECT
operates	P	
gets	S	ROLL_AMOUNT
gets	S	ROLL_DIRECTION
is left	S	ROLL_DIRECTION
pivots...up	S	LEFT, RIGHT_AILERON
pivots...down	S	LEFT, RIGHT_AILERON
gets	S	PITCH_AMOUNT
gets	S	PITCH_DIRECTION
is up	S	PITCH_DIRECTION
pivots...up	S	ELEVATOR
pivots...down	S	ELEVATOR
gets	S	YAW_AMOUNT
gets	S	YAW_DIRECTION
is left	S	YAW_DIRECTION
pivots...left	S	RUDDER
pivots...right	S	RUDDER

OPERATION	OBJECT
pivots...up	left, right_aileron (roll_amount)
pivots...down	left, right_aileron (roll_amount)
pivots...up (down)	elevctor (pitch_amount)
pivots...left (right)	rudder (yaw_amount)

(ASSOCIATED OBJECTS)

OPERATION	SPACE	IDENTIFIER
operates	P	
gets	S	GET
gets	S	GET
is left	S	IS_LEFT
pivots...up	S	PIVOT_UP
pivots...down	S	PIVOT_DOWN
gets	S	GET
gets	S	GET
is up	S	IS_UP
pivots...up	S	PIVOT_UP
pivots...down	S	PIVOT_DOWN
gets	S	GET
gets	S	GET
is left	S	IS_LEFT
pivots...left	S	PIVOT_LEFT
pivots...right	S	PIVOT_RIGHT

ROLL_AMOUNT	**ROLL_DIRECTION**	**LEFT_AILERON, RIGHT_AILERON**
GET	GET IS_LEFT	PIVOT_UP PIVOT_DOWN
PITCH_AMOUNT	**PITCH_DIRECTION**	**ELEVATOR**
GET	GET IS_UP	PIVOT_UP PIVOT_DOWN
YAW_AMOUNT	**YAW_DIRECTION**	**RUDDER**
GET	GET IS_LEFT	PIVOT_LEFT PIVOT_RIGHT

ELEVATOR_CONTROLLER

AILERON_CONTROLLER

RUDDER_CONTROLLER

ROLL_AMOUNT	ROLL_DIRECTION	
GET	GET IS_LEFT	ROLL_PACKAGE
PITCH_AMOUNT	PITCH_DIRECTION	
GET	GET IS_UP	PITCH_PACKAGE
YAW_AMOUNT	YAW_DIRECTION	
GET	GET IS_LEFT	YAW_PACKAGE

LEFT_AILERON,
RIGHT_AILERON

PIVOT_UP
PIVOT_DOWN

AILERON_PACKAGE

ELEVATOR

PIVOT_UP
PIVOT_DOWN

ELEVATOR_PACKAGE

RUDDER

PIVOT_LEFT
PIVOT_RIGHT

RUDDER_PACKAGE

ELEVATOR_CONTROLLER

AILERON_CONTROLLER

RUDDER_CONTROLLER

AMOUNT_TYPE	DIRECTION_TYPE	
GET	GET IS_LEFT	ROLL_PACKAGE
AMOUNT_TYPE	**DIRECTION_TYPE**	
GET	GET IS_UP	PITCH_PACKAGE
AMOUNT_TYPE	**DIRECTION_TYPE**	
GET	GET IS_LEFT	YAW_PACKAGE

AILERON_TYPE

PIVOT_UP
PIVOT_DOWN

AILERON_PACKAGE

ELEVATOR_TYPE

PIVOT_UP
PIVOT_DOWN

ELEVATOR_PACKAGE

RUDDER_TYPE

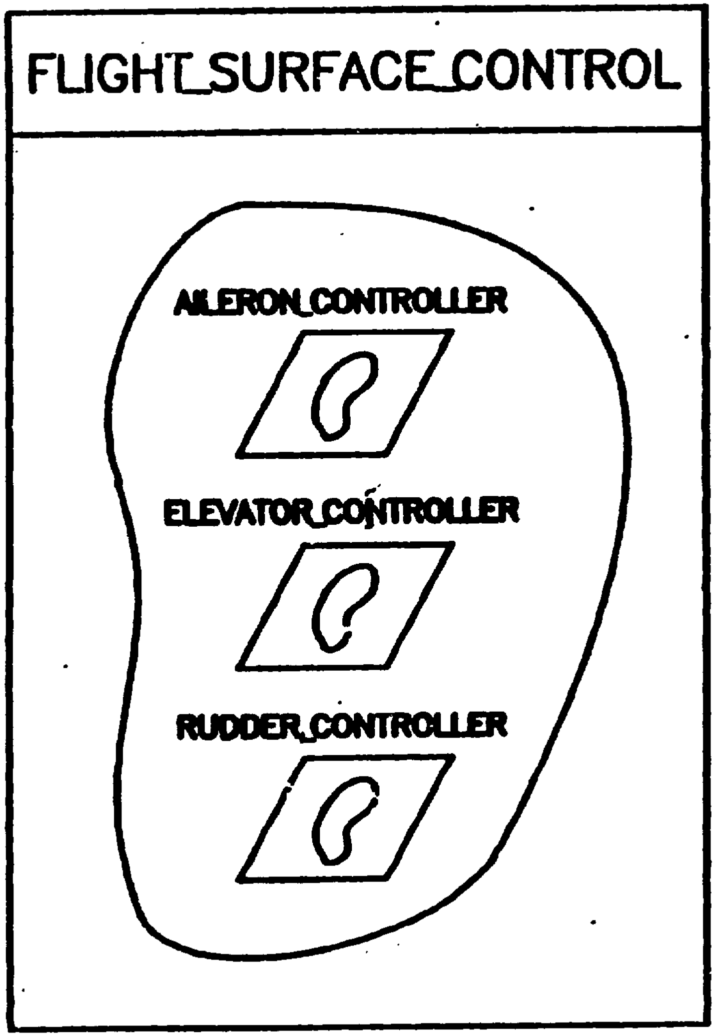
PIVOT_LEFT
PIVOT_RIGHT

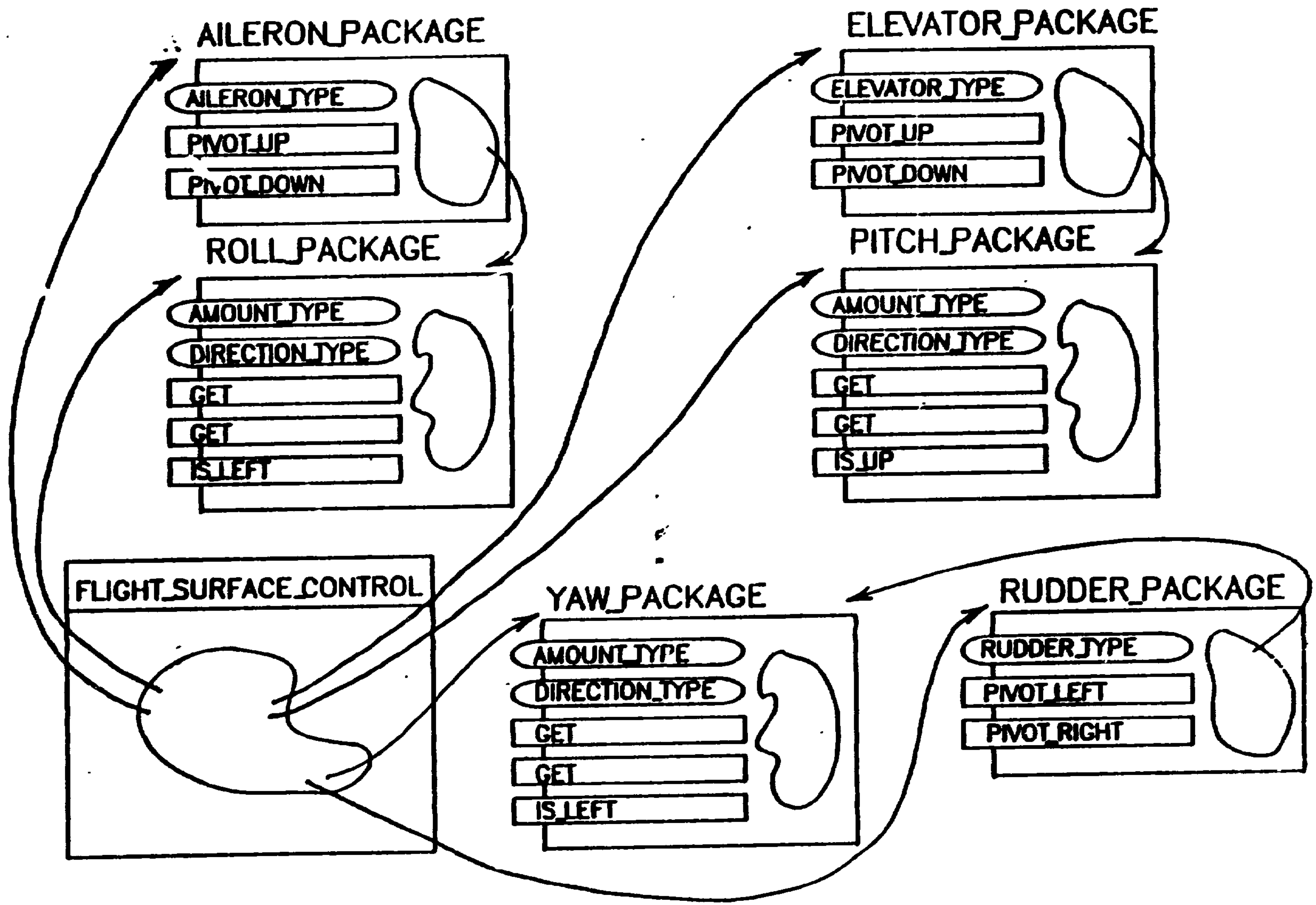
RUDDER_PACKAGE

ELEVATOR_CONTROLLER

AILERON_CONTROLLER

RUDDER_CONTROLLER


FLIGHT_SURFACE_CONTROL
AILERON_CONTROLLER
ELEVATOR_CONTROLLER
RUDDER_CONTROLLER


AILERON_PACKAGE
AILERON_TYPE
PIVOT_UP
PIVOT_DOWN
ROLL_PACKAGE
AMOUNT_TYPE
DIRECTION_TYPE
GET
GET
IS_LEFT
ELEVATOR_PACKAGE
ELEVATOR_TYPE
PIVOT_UP
PIVOT_DOWN
PITCH_PACKAGE
AMOUNT_TYPE
DIRECTION_TYPE
GET
GET
IS_UP
FLIGHT_SURFACE_CONTROL
YAW_PACKAGE
AMOUNT_TYPE
DIRECTION_TYPE
GET
GET
IS_LEFT
RUDDER_PACKAGE
RUDDER_TYPE
PIVOT_LEFT
PIVOT_RIGHT

```
package PITCH_PACKAGE is

   type AMOUNT_TYPE is range 0 .. 30; -- degrees
   type DIRECTION_TYPE is limited private;

   procedure GET (PITCH_AMOUNT : out AMOUNT_TYPE);
   procedure GET (PITCH_DIRECTION : out DIRECTION_TYPE);
   function IS_UP (PITCH_DIRECTION : in DIRECTION_TYPE)
                                         return BOOLEAN;

private

   ...

end PITCH_PACKAGE;
```

```
with PITCH_PACKAGE;
use  PITCH_PACKAGE;
package ELEVATOR_PACKAGE is

  type ELEVATOR_TYPE is limited private;

  procedure PIVOT_UP (ELEVATOR : in out ELEVATOR_TYPE;
                      AMOUNT : in AMOUNT_TYPE);
  procedure PIVOT_DOWN (ELEVATOR : in out ELEVATOR_TYPE;
                        AMOUNT : in AMOUNT_TYPE);

private

  ...

end ELEVATOR_PACKAGE;
```

```
package ROLL_PACKAGE is

   type AMOUNT_TYPE is range 0 .. 30; -- degrees
   type DIRECTION_TYPE is limited private;

   procedure GET (ROLL_AMOUNT : out AMOUNT_TYPE);
   procedure GET (ROLL_DIRECTION : out DIRECTION_TYPE);
   function IS_LEFT (ROLL_DIRECTION : in DIRECTION_TYPE)
                                     return BOOLEAN;

private

   ...

end ROLL_PACKAGE;
```

```
with ROLL_PACKAGE;
use  ROLL_PACKAGE;
package AILERON_PACKAGE is

   type AILERON_TYPE is limited private;

   procedure PIVOT_UP (AILERON : in out AILERON_TYPE;
                       AMOUNT : in AMOUNT_TYPE);
   procedure PIVOT_DOWN (AILERON : in out AILERON_TYPE;
                         AMOUNT : in AMOUNT_TYPE);

private

   ...

end AILERON_PACKAGE;
```

```
procedure FLIGHT_SURFACE_CONTROL is

   task AILERON_CONTROLLER;
   task ELEVATOR_CONTROLLER;
   task RUDDER_CONTROLLER;

   task body AILERON_CONTROLLER is separate;
   task body ELEVATOR_CONTROLLER is separate;
   task body RUDDER_CONTROLLER is separate;

begin

   null;

end FLIGHT_SURFACE_CONTROL;
```

```
with ELEVATOR_PACKAGE, PITCH_PACKAGE;
use  ELEVATOR_PACKAGE, PITCH_PACKAGE;

separate ( FLIGHT_SURFACE_CONTROL )

task body ELEVATOR_CONTROLLER is

  PITCH_AMOUNT      : AMOUNT_TYPE;
  PITCH_DIRECTION   : DIRECTION_TYPE;
  ELEVATOR          : ELEVATOR_TYPE;
begin
  loop
    GET (PITCH_AMOUNT);
    GET (PITCH_DIRECTION);
    if IS_UP (PITCH_DIRECTION) then
      PIVOT_UP (ELEVATOR, PITCH_AMOUNT);
    else
      PIVOT_DOWN (ELEVATOR, PITCH_AMOUNT);
    end if;
  end loop;
end ELEVATOR_CONTROLLER;
```

*U.S. GOVERNMENT PRINTING OFFICE:1987 -730-079/ 40055 REGION NO. 4