Ada® Training Curriculum

AD-A165 302
Software Engineering Methodologies
M201
Teacher's Guide
Volume III

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

Contract DAAB07-83-C-K506

Prepared By:
SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office
*Approved For Public Release/Distribution Unlimited
INSTRUCTOR NOTES

THEME: PDL's ARE RAPIDLY BECOMING THE REQUIRED (PREFERRED) METHODS FOR EXPRESSING A DETAILED SOFTWARE DESIGN.

PURPOSE: TO SHOW THE IMPORTANT FEATURES OF ADA PDL.

REFERENCE: VAN LEER, P. "TOP DOWN DEVELOPMENT USING A PROGRAM DESIGN LANGUAGE" IBM SYSTEMS JOURNAL VOL. 15, NO. 2; 1976

IEEE GUIDELINES FOR AN ADA PDL; 1985

Copyright by SofTech, Inc. 1984. This material may be reproduced by or for the U.S. Government pursuant to the copyright license under DAR clause 7-104.9(a) (May 81).
Section 21

PROGRAM DESIGN LANGUAGES
INSTRUCTOR NOTES

WE AREN'T GOING TO DEFINE A SINGLE PDL. WE ARE GOING TO GIVE YOU AN INTRODUCTION TO THE ISSUES INVOLVED WITH DEFINING AND USING PDL'S.

THE DIFFICULTY IN UPDATING FLOWCHARTS REALLY LED TO PDL. IN MAINTAINING A SYSTEM IT IS FOOLISH TO Rely ON FLOWCHARTS. SO A MECHANISM WAS NEEDED TO: 1) DESCRIBE THE HIGH LEVEL STRUCTURE AND 2) BE UPDATED EASILY.

STATE THAT MANY ELEMENTS OF DESIGN ARE GRAPHIC IN NATURE. TEXTUAL LANGUAGES, LIKE PDL, NATURALLY FALL SHORT OF MEETING ALL THE NEEDS OF A DESIGNER.
PROGRAM DESIGN LANGUAGE OVERVIEW

- IS A TEXTUAL LANGUAGE, PRECISE ENOUGH TO DESCRIBE SOFTWARE, YET EXPRESSIVE ENOUGH TO DESIGN WITH.

\[
\begin{align*}
\text{PRECISION} & \quad \text{EXPRESSION} \\
\text{ALWAYS COMPETE.} \\
\text{PROPER BALANCE IS UNKNOWN.}
\end{align*}
\]

- PDL'S GREW OUT OF THE PROBLEMS ON TRYING TO KEEP UP-TO-DATE FLOWCHARTS OF THE SYSTEM.

- PDL'S WILL BE REQUIRED FOR DOCUMENTING DETAIL DESIGNS.
INSTRUCTOR NOTES

A DESIGN LANGUAGE IS A TEXTUAL REPRESENTATION FOR THE PRECISE EXPRESSION OF PROGRAM OR SYSTEM DESIGNS.

IN USING THE IEEE PDL STANDARD, THERE IS A "RECOMMENDED PRACTICE" THAT GOES ALONG WITH IT. IT PROVIDES GENERAL GUIDANCE IN USING THE PDL, THE FEATURES OF AN ADA PDL, DESIGN LANGUAGE CHARACTERISTICS, MANAGEMENT ISSUES, ETC.
ADA AS A PROGRAM DESIGN LANGUAGE

- IEEE HAS PRODUCED A DRAFT SET OF GUIDELINES FOR USE OF ADA AS A DESIGN LANGUAGE.
  - ONLY "REAL" STANDARD FOR MODERN PDL.
  - USED AS A BASIS FOR THIS SECTION

- IEEE MOTIVATING GOALS FOR USE OF ADA AS A PDL ARE TO:
  - UTILIZE THE POWER OF THE ADA PROGRAMMING LANGUAGE IN THE DESIGN PROCESS.
  - ENHANCE COMMUNICATION BY USING THE SAME LANGUAGE NOTATION THROUGHOUT THE LIFE CYCLE.
  - SUPPORT QUALITY SOFTWARE DESIGN BY FOCUSING ON APPROPRIATE LEVELS OF DESIGN DETAIL.
  - CAPITALIZE ON THE EMERGING AVAILABILITY OF ADA TOOLS AND INDUSTRY SUPPORT FOR THE ADA LANGUAGE.
  - PROVIDE A MECHANISM THAT SUPPORTS THE TRANSITION TO ADA BASED SOFTWARE ENGINEERING PRACTICE.
  - PROVIDE A BASIS FOR STANDARDIZATION.
INSTRUCTOR NOTES

The use of a design language can increase productivity in a number of ways. First, as long as design documentation can be written in machine readable form, automated tools can be created to check for completeness and consistency, and to automate development activities. Second, such a language facilitates communication between various members of a project team. Finally, it is advantageous to have a single notation that can be used throughout all activities of the software life cycle.

Use of a design language can increase software quality by facilitating the early detection and correction of errors by helping team members to communicate through a common language, and by providing a mechanism for verification of design and the translation of the design notation. A design language provides a vehicle both for communications and for supporting various activities of the product life cycle, and should be human engineered, precise, analyzable, verifiable and supportive of various programming methodologies.

VG 778.1

21-31
RATIONALE FOR USING A PDL

- TO INCREASE PRODUCTIVITY
  - PDLs ARE MACHINE READABLE, TOOLS CAN CHECK FOR COMPLETENESS, CONSISTENCY AND CONFORMANCE TO STANDARDS.
  - PDLs DEFINE A COMMON LANGUAGE TO ENHANCE COMMUNICATIONS WITHIN A PROJECT.

- TO IMPROVE SOFTWARE QUALITY
  - PDLs FACILITATE EARLY DETECTION AND CORRECTION OF ERRORS BY AIDING IN COMMUNICATION AND VERIFICATION.

- TO MINIMIZE THE RISKS INVOLVED IN DEVELOPING AND MAINTAINING SOFTWARE
  - PDLs MAKE DESIGNS VISIBLE EARLY.
  - COMMON PDL AND IMPLEMENTATION LANGUAGE REDUCES TRAINING NEEDS.
INSTRUCTOR NOTES

A DESIGN METHODOLOGY IS A BODY OF PROCEDURES AND TECHNIQUES WHICH CAN BE USED TO CONSTRUCT A SYSTEM DESIGN. THE DESIGN ITSELF IS AN ABSTRACTION OF THE PROPOSED SYSTEM; WHEN THE SYSTEM IS IMPLEMENTED ACCORDING TO THE DESIGN, IT IS EXPECTED TO PERFORM ACCORDING TO THE SYSTEM SPECIFICATION AND TO SATISFY THE SYSTEM REQUIREMENTS. THE DESIGN LANGUAGE SHOULD SUPPORT THE DESCRIPTION OF A PRODUCT AT THE APPROPRIATE LEVELS OF DETAIL FOR EACH ACTIVITY OF A GIVEN METHODOLOGY.
PDL REQUIREMENTS

A PDL MUST SUPPORT ...

- **ABSTRACTION** - EMPHASIZING PARTICULAR CONCEPTS WHILE SUPPRESSING UNNECESSARY DETAIL.
- **DECOMPOSITION** - DIVIDING A LARGE SYSTEM INTO SMALLER, MORE MANAGEABLE PIECES WHILE MAINTAINING A FIXED LEVEL OF DETAIL.
- **INFORMATION HIDING** - ISOLATING DESIGNATED INFORMATION TO CONTROL THE DEPENDENCE ON A PARTICULAR IMPLEMENTATION.
- **STEPWISE REFINEMENT** - PROGRESSIVELY ADDING DETAIL TO A DESIGN DESCRIPTION.
- **MODULARIZATION** - ISOLATING PORTIONS OF A SYSTEM INTO LOGICALLY SEPARABLE PARTS SUCH THAT A CHANGE IN ONE PART HAS MINIMAL IMPACT ON OTHER PARTS.

NOTE: THESE ARE SOME OF THE DOD'S REQUIREMENTS FOR ADA
INSTRUCTOR NOTES

WHILE THESE QUALITIES ARE DIFFICULT TO DEFINE PRECISELY OR QUANTIFY, THEY ARE USEFUL IN COMPARING ONE DESIGN LANGUAGE TO ANOTHER. FOR EXAMPLE, A DESIGN LANGUAGE WHICH LEADS TO A READABLE DESIGN DESCRIPTION IS BETTER THAN ONE WHICH PRODUCES UNREADABLE DESIGNS. EACH ORGANIZATION SHOULD HAVE ITS OWN DEFINITIONS AND MEASUREMENT STANDARDS FOR THE ABOVE TERMS CONSISTENT WITH ITS PARTICULAR BUSINESS AND DEVELOPMENT ENVIRONMENT.
QUALITIES OF A DESIGN EXPRESSED IN A PDL

- READABLE
- UNDERSTANDABLE
- ANALYZABLE
- TRACEABLE
- VALIDATABLE
- RELIABLE
- MINIMUM COUPLING
- MAXIMUM COHESIVENESS
- COMPLETE
- PRECISE
- VERIFIABLE
- STRUCTURED
- CONSISTENT
PDL CHARACTERISTICS

A PROGRAM DESIGN LANGUAGE SHOULD ADDRESS ALL OF THE FOLLOWING:

- EXPRESSIVE POWER
  - ALGORITHM DESIGN
  - DATA STRUCTURES
  - SUPPLEMENTARY INFORMATION
  - CONNECTIVITY
  - MANAGEMENT OF COMPLEXITY
- HUMAN FACTORS
- ANALYZABILITY
- RELATIONSHIP BETWEEN THE PDL AND IMPLEMENTATION LANGUAGE
- TOOL IMPACT ON THE PROGRAM DESIGN LANGUAGE
- IMPLEMENTATION CONSIDERATIONS
PDL CHARACTERISTICS

- PDL MUST BE ABLE TO EXPRESS PRODUCT RELATED INFORMATION ...
  - CHARACTERISTICS - INCLUDES PERFORMANCE CONSTRAINTS, SIZE
    RESTRICTIONS, THE TARGET ENVIRONMENT, RELIABILITY, AND SECURITY
    REQUIREMENTS.
  - PERFORMANCE - INCLUDES CRITICAL TIMING, FREQUENCY, CAPACITY AND
    OTHER LIMITATIONS.
  - FAULT TOLERANCE - INCLUDES ERROR DETECTION, DIAGNOSIS, AND ERROR
    HANDLING; BACKUP AND RECOVERY; RELIABILITY, AND REDUNDANCY.
  - SECURITY - INCLUDES MULTILEVEL SECURITY CONSTRAINTS, SET/USE ACCESS,
    RESTRICTIONS, BREACH DETECTION AND HANDLING.
  - DISTRIBUTION - GEOGRAPHICAL DISTRIBUTION OF PROCESSING, DATA STORAGE
    AND ACCESS.
  - ADAPTATION - INCLUDES REQUIREMENTS PERTAINING TO ACCOMMODATION OF
    DIFFERING LEVELS OF USER EXPERTISE.
  - ASSERTIONS - DESCRIPTION OF A CONTEXT WITHIN WHICH AN ALGORITHM
    EXISTS. FORMAL ASSERTIONS PROVIDE A MECHANISM FOR DESCRIBING THIS
    CONTEXT FOR DESIGN EVALUATION PURPOSES.

- PDL MUST BE ABLE TO EXPRESS DEVELOPMENT PROCESS RELATED INFORMATION ...
  - TRACEABILITY - METHOD OF TRACING PRODUCT REQUIREMENTS TO ONE OR MORE
    DESIGN ELEMENTS. DURING THE DEVELOPMENT OF A SYSTEM, VARIOUS KINDS
    OF REQUIREMENTS MUST BE RECORDED. IN ADDITION TO FUNCTIONAL
    REQUIREMENTS, THERE ARE PERFORMANCE, SECURITY, LOGISTICS, AND OTHER
    REQUIREMENTS. FOLLOWING EACH DESIGN ACTIVITY, EACH REQUIREMENT IS
    "TRACED" TO ONE OR MORE DESIGN ELEMENTS.
  - ORGANIZATION - INCLUDES DIVISION OF WORK AND ALLOCATION TO TEAM
    MEMBERS; I.E. WORK BREAKDOWN STRUCTURING.
  - PLANNING - INCLUDES RESOURCE ESTIMATION AND SCHEDULING.
  - TRACKING - INCLUDES DESIGN UNIT COMPLETION, PROJECT PROGRESS, AND
    PRODUCT VERSION CONTROL.
INSTRUCTOR NOTES

THIS CHART ILLUSTRATES THAT SIGNIFICANT AMOUNTS OF DESIGN INFORMATION ARE EXPRESSIBLE USING ADA CONSTRUCTS. IF ANOTHER LANGUAGE WAS CHARTED, E.G. FORTRAN, THE CHART WOULD BE MUCH LESS FILLED IN.
### Mapping of Ada PDL Features to Design Characteristics

<table>
<thead>
<tr>
<th>DESIGN CHARACTERISTICS</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>O</th>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACTION</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA FLOW</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTROL FLOW</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA STRUCTURE DESIGN</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA REFERENCE</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERFACES</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPENDENCIES</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REUSABILITY</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFERRED DECISIONS</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Ada Language Features and Constructs

- **A**: PACKAGES
- **B**: SUBPROGRAMS
- **C**: TASKS
- **D**: SPEC/BODY
- **E**: PRAGMA
- **F**: TYPES
- **G**: OBJECTS
- **H**: STUBS/IS SEPARATE
- **I**: COMMENTS
- **J**: PRIVATE
- **K**: EXCEPTIONS
- **L**: REP SPECS
- **M**: ALLOCATIONS
- **N**: GENERICS
- **O**: PREDEFINED PACKAGES
- **P**: NAMING
- **Q**: STATEMENTS

**Source:** IEEE PDC GUIDELINES, 1985

**VG 778.1 21-8**
INSTRUCTOR NOTES

THE EXAMPLE, TAKEN FROM THE ADA LANGUAGE REFERENCE MANUAL [LRM], SHOWS HOW PACKAGES, SPECIFICATIONS, BODIES AND STUBS ALL WORK TOGETHER TO DEFER DECISIONS.

REFER BACK TO THE PREVIOUS CHART.
ADA PDL EXAMPLE
(DEFERRED IMPLEMENTATION DECISIONS)

procedure TOP is
  type REAL is digits 10;
  R, S: REAL := 1.0;
  package FACILITY is
    Pi: constant := 3.14159_26536;
    function F(X: REAL) return REAL;
    procedure G(Y, Z: REAL);
  end FACILITY;

  package body FACILITY is separate;
  -- postpone decision on implementation of F and G
  procedure TRANSFORM(U: in out REAL) is separate;
  -- postpone decision on implementation of TRANSFORM

  begin -- TOP
    TRANSFORM(R);
    ...
    FACILITY.G(R, S);
  end TOP;

INSTRUCTOR NOTES

IN AN ADA DESIGN LANGUAGE, COMMENTS PROVIDE AN INDISPENSABLE MECHANISM FOR ADDING NEW CONSTRUCTS TO THE DESIGN LANGUAGE TO ANNOTATE THE DESIGN.

UNSTRUCTURED COMMENTS SHOULD BE USED FOR NATURAL LANGUAGE EXPLANATION OF STATEMENTS MADE IN THE ADA LANGUAGE. THESE COMMENTS CAN BE USED TO INDICATE ANY INFORMATION REQUIRED IN THE DESIGN PROCESS WHEN FORMAL STRUCTURES ARE NOT REQUIRED.

A STRUCTURED COMMENT, OR ANNOTATION, PROVIDES EXPLANATORY DESIGN INFORMATION. IT IS IDENTIFIED BY A SENTINEL CHARACTER, WORD OR PHRASE IMMEDIATELY FOLLOWING THE DOUBLE DASH THAT INDICATES A COMMENT TO AN ADA COMPILER (E.G. "--!" OR "--*"). THE SENTINEL CHARACTER HAS TWO FUNCTIONS. THE FIRST IS TO HIGHLIGHT THE COMMENT AS BELONGING TO THE FORMAL STRUCTURE OF THE DESIGN LANGUAGE, ALERTING ANY REVIEWER OF THE DESIGN DOCUMENTATION THAT THE INFORMATION THAT FOLLOWS IS OF SPECIAL SIGNIFICANCE. THE SECOND FUNCTION IS TO INDICATE TO TOOLS WHICH PROCESS THE DESIGN THAT THE TOOL MAY BE REQUIRED TO ACT UPON THE INFORMATION GIVEN IN THE COMMENT. ASSOCIATED WITH THIS SYMBOL ARE SEMANTIC RULES INDICATING WHETHER THE COMMENT APPLIES TO THE PRECEDING OR TO THE FOLLOWING ADA CONSTRUCT AND WHAT SORT OF CONSTRUCTS ARE ALLOWED AFTER THE SYMBOL ACCORDING TO THE CONTEXT IN WHICH IT APPEARS. ALTHOUGH THE STRUCTURED COMMENT IS A RESTRICTION IN TERMS OF THE SYNTAX OF THE ADA LANGUAGE, IT PROVIDES A MECHANISM FOR EXTENDING THE DESIGN LANGUAGE SEMANTICS TO INCLUDE ADDITIONAL DESIGN ORIENTED INFORMATION.
ROLE OF COMMENTS IN ADA PDL

- Unstructured comments should be used for natural language explanations
  - for human to human communication
  - for rationale
  - use normal comment delimiters

- Structured comments (annotation) provide explanatory design information
  - extends ADA syntax to express design information not easily expressed in ADA

  // an English statement of conditions //</p>

*** HIGH LEVEL DESIGN
***
***
*** INCLUDES A STATEMENT OF HIGH LEVEL DESIGN
*** END HIGH LEVEL DESIGN
INSTRUCTOR NOTES

THIS SLIDE AND THE 6 THAT FOLLOW ALL SHOW SAMPLES OF AN ADA PDL WALKTHRU THEN WITH THE CLASS EMPHASIZING THE READABILITY OF THE DESIGN WHEN EXPRESSED IN ADA.

NOTATION

//~~~~~~~~~~~~~~~ // - AN ILLEGAL ADA CONSTRUCT TO EXPRESS DESIGN INFORMATION THAT WILL BE LATER EXPANDED OR THAT IS MORE INFORMATIVE EXPRESSED IN ENGLISH THAN IN ADA.
procedure SORT (TABLE : in out TABLE_TYPE) is
begin
  if -- TABLE has more than one entry then
    while -- TABLE is not sorted loop
      for -- each pair of entries in TABLE loop
        if -- 1st entry is greater than 2nd then
          // exchange the two entries//;
        end if;
      end loop;
    end loop;
  end if;
end SORT;

procedure SORT (TABLE : in out TABLE_TYPE) is
  SWAPPED_ITEMS : Boolean;
  TEMP : ITEM_TYPE;
begin
  if TABLE'LENGTH > 1 then
    SWAPPED_ITEMS := TRUE;
    -- loop while the table is not sorted
    while SWAPPED_ITEMS := FALSE;
      SWAPPED_ITEMS := FALSE;
      for I in TABLE'FIRST..INDEX_TYPE'Pred(TABLE'LAST) loop
        if TABLE (I) > TABLE(INDEX_TYPE'Succ(I))
          then
            SWAPPED_ITEMS := TRUE;
            TEMP := TABLE(I);
            TABLE(I) := TABLE(INDEX_TYPE'Succ(I))
            TABLE(INDEX_TYPE'Succ(I)) := TEMP;
          end if;
        end loop;
      end loop;
      -- for each pair of entries
      end loop;
    end if;
  end if;
end SORT;
INSTRUCTOR NOTES

POINT OUT HOW BETWEEN REFINEMENT MORE DETAILED AND SPECIFIC THEN THE DESIGN BECOMES.
AN ADA PDL
(STEP WISE REFINEMENT)

INITIAL DESIGN

procedure EMITTER_IDENTIFICATION is
begin
    /* this procedure searches the emitter identification table trying to
    match the characteristics of a newly identified emitter; */
end EMITTER_IDENTIFICATION

REFINEMENT 1

procedure EMITTER_IDENTIFICATION (NEW in : EMITTER; CLASS
out : EMITTER_CLASS) is
    -- this procedure searches the emitter identification table
    -- trying to match the characteristics of a newly identified
    -- emitter
    // search emitter table comparing on PULSE_WIDTH, PRI, JITTER,
    and FREQUENCY; //
    if // more than one emitter match occurred // then
        // perform pair wise ambiguity resolution; //
    end if

    // set outputs and return; //
end EMITTER_IDENTIFICATION;
AN ADA PDL
(STEP-WISE REFINEMENT)

REFINEMENT 2

procedure EMITTER_IDENTIFICATION (NEW : In EMITTER;
CLASS : out EMITTER CLASS;
AMBIGUOUS : out boolean)
is

-- this procedure searches the emitter identification table
-- trying to match the characteristics of a newly identified
-- emitter. If an ambiguous match is made, a best guess
-- emitter class is returned as well as an ambiguity
-- flag.

TYPE EMITTER_TABLE is // collection of EMITTER Objects //;
EMITTER_IDS : EMITTER_TABLE;
BEST_BET, CHALLENGER : EMITTER_CLASS;
EMITTER_MATCH : EMITTER_TABLE;

begin

-- assume an accurate match
AMBIGUOUS := TRUE;
-- search emitter table
for I in 1.. // # of emitter ID's // loop

if // PULSE_WIDTH, PRI, JITTER, and FREQUENCY are within
range of the Ith EMITTER_ID's parameters // then
// add Ith EMITTER_ID to the EMITTER_MATCH table //;
end if

end loop;
--
-- check for 0, 1, or multiple possible emitter matches
--
AN ADA PDL

(STEP-WISE REFINEMENT) (Continued)

if // # of EMITTER_MATCHES = 0 // then
    // call new emitter handler and issue error status //;
elsif // 1 EMITTER_MATCH // then
    // assign matched_class emitter to CLASS //;
else
    --
    -- perform pair wise ambiguity resolution
    --
    // assign 1st and 2nd EMITTER_MATCH to BEST_BET and
    CHALLENGER //;
    for I in 1..// number of possible EMITTER_MATCHES-1 //
        loop
            RESOLVE_AMBIGUITY (BEST_BET, CHALLENGER, NO_RESOLUTION);
            if NO_RESOLUTION then
                AMBIGUITY := TRUE;
            end if;
            // assign next EMITTER_MATCH to CHALLENGER //;
        end loop;
        // assign the reigning BEST_BET to CLASS as the identified
        emitter //;
    end if;
end EMITTER_IDENTIFICATION;
AN ADA PDL

(CCHARACTERIZING DATA)

NORMAL

DATA

TYPE TYPE:

ING

INTEGER //within table size////

ype FM_RADIO is new FREQUENCY //in FM range//;

type HAIR_COLOR is new COLOR;

package GLOBAL_DATA is

type FREQUENCY is //500 to 1500 kilocycle with 1 kilocycle

delta//;

ype MODE is (initialization, integration, acquisition,

SELF-TEST);

 MODE_Switch: MODE;

ERROR_FLAG: boolean;

end GLOBAL_DATA;

Package BANKING is

type KEY is limited private;

type DOLLARS is private

procedure EARN (PAY: in out DOLLARS);

procedure OPEN (K: out KEY);

procedure CLOSE (K: in out KEY);

procedure DEPOSIT (K: in KEY; MONEY: in out DOLLARS);

procedure WITHDRAW (K: in KEY; MONEY: in out DOLLARS);

private

type DOLLARS is NATURAL;

type KEY is access DOLLARS;

end BANKING;

VG 778.1

21-15
AN ADA PDL
(EXPRESSING FLOW OF CONTROL)

-- using AIRCRAFT_STATUS as an indicator
-- of an attack plane's mode of operation

case AIRCRAFT_STATUS is
  when //position is not known// =>
    //do nothing//;
  when //not busy// =>
    //select a potential target//;
    ACQUIRE_TARGET (//aircraft, target//);
  when //target is lost// =>
    //put target on search list//;
  when //target is located// =>
    STRIKE_TARGET;
  when others =>
    //indicate status error//;
    //send warning message to the operator//;
end case;

loop
  //compare entry with the incoming NAME//;
  exit when //match occurs//;
  //get the next entry from the NAME table//;
end loop;

for I in reverse 1.. //all unverified signals//
  loop
    //get a signal//;
    if //frequency is in friendly range// then
      IDENTIFY_FRIEND (//the current signal//);
    else
      IDENTIFY_FOE (//the current signal//);
    end if;
  end loop;

while //character is alpha numeric// loop
  //send a character to the printer//;
  //construct a character from input stream//;
end loop;
AN ADA PDL

(EXPRESSING ALGORITHMS)

CRITICAL_CALCULATION := (SIN(x) * COS(y) * DIRECTION +
begin DIRECTION := (SIN(x) * COS(y) * DIRECTION +
except when CONSTRAINT_ERROR => RECOMPUTE_DIRECTION;
end begin DIRECTION := (SIN(x) * COS(y) * DIRECTION +
procedure PUSH (E: in FLOAT;
S: in out STACK) is
begin // the STACK is full // then
// report overflow via an exception //;
else // increase the top of stack indicator //;
// place E on the stack //;
end PUSH;
end

function MAXIMUM (FIRST, SECOND: in FLOAT)
begin //FIRST is greater or even equal to SECOND // then
else return SECOND;
end if;
end if;
end

COMPUTATION WITH ERROR HANDLING

DATA STRUCTURE OPERATION

MATH FUNCTION

VG 778.1
AN ADA PDL

(EXPRESSING TIMING)

delay 60; -- suspends task for one minute
delay 20 x CYCLES; -- where CYCLE is a constant of
               type DURATION

loop
   EXECUTE_PASS 1;
   ATTACH_DEVICE;
   EXECUTE_PASS 2;
   delay .0039062-MY_TIME; -- where MY_TIME is
               execution time of
               -- this code loop. Thus, this
               -- loop executes 256 times per second
end loop;
INSTRUCTOR NOTES

IF NO ONE CHIMES UP FOR DISCUSSION, AS THE QUESTION "SHOULD ADA BE USED AS A PDL?"
INSTRUCTOR NOTES

THEME: OTHER METHODS ARE AVAILABLE AS ALTERNATIVES TO FLOW CHARTS TO EXPRESS DETAIL DESIGNS.

PURPOSE: A QUICK OVERVIEW OF SEVERAL METHODS.

REFERENCES: KATZAN, H. "SYSTEMS DESIGN AND DOCUMENTATION: AN INTRODUCTION TO THE HIPO METHOD" VAN NOSTRAND REINHOLD, NY; 1976

STAY, J. "HIPO AND INTEGRATED PROGRAM DESIGN" IBM SYSTEMS JOURNAL VOL. 15, NO. 2; 1976

NASSI, I., SHNEIDERMAN, B. "FLOWCHART TECHNIQUES FOR STRUCTURED PROGRAMMING" SIGPLAN VOL. 8, NO. 8; AUGUST 1973

MARCA, D. "A METHOD FOR SPECIFYING STRUCTURED PROGRAMS" SIGSOFT VOL. 4, NO. 3; JULY 1979
INSTRUCTOR NOTES

HIERARCHICAL INPUT PROCESSING OUTPUT (HIPO) WAS CREATED BY IBM TO DOCUMENT SOFTWARE MORE OR LESS AFTER THE FACT.

"PACKAGE" IS A HIPO KEYWORD. DON'T CONFUSE WITH Ada PACKAGE.

HIPO IS ORIENTED TOWARDS GIVING A TOP DOWN, STEP BY STEP DESCRIPTION.
HIPO

OVERVIEW

- HIPO SEE S SOFTWARE AS
  - A HIERARCHY OF FUNCTIONS, EACH HAVING AN INPUT-PROCESS-OUTPUT NATURE

- HIPO
  - IS A DOCUMENTATION TOOL DEVELOPED BY IBM,
    - DOCUMENTS PROCEDURAL DETAILS OF SOFTWARE,
    - COLLECTS INFORMATION INTO PACKAGES.
INSTRUCTOR NOTES

- NOTE TREE STRUCTURE OF VTOC

- NOTE SKETCHINESS OF OVERVIEW DIAGRAMS

- NOTE DETAIL OF DIAGRAMS

OVERVIEW DIAGRAMS SHOW CREATIONS AND CONSUMPTIONS OF FILES, DETAIL DIAGRAMS, SHOW RECORDS OR FIELDS.
A HIPO PACKAGE

1

A VISUAL TABLE OF CONTENTS
(HIERARCHY)

2

OVERVIEW DIAGRAMS
(FOR HIGH LEVELS IN THE VTOC)

3

DETAIL DIAGRAMS
(FOR LOW LEVELS IN THE VTOC)
INSTRUCTOR NOTES

- NOTE THAT THIS ONLY SHOWS SIMPLE HIERARCHY. CONTRAST THIS TO A CONSTANTINE STRUCTURE CHART.

- NOTE THAT THE VTOC IS THE BIG PICTURE.
INSTRUCTOR NOTES

- NOTE THE "CALLED FROM" AND "CALL TO" ARROWS OF THE PROCESS BOX.

- NOTE THE SKETCHINESS OF INPUTS AND OUTPUTS.
A DETAILLED DIAGRAM

1. IF M-ON-HAND MINUS Q-QUALITY-REQUESTED IS LESS THAN ZERO
   THEN a) QUANTITY AVAILABLE - M-ON-HAND
       b) PERFORM "DETERMINE-QUANTITY-BACK-ORDER (2 1)"
   ELSE c) QUANTITY AVAILABLE - Q-QUALITY-REQUESTED

2. PERFORM "REDUCE-INVENTORY-ON-HAND (2 2)"

3. PERFORM "UPDATE-TOTAL-SALES (2 3)"

4. PERFORM "REVISE-ACTIVITY-DATE (2 4)"

5. IF M-ON-HAND PLUS M-ON-ORDER IS LESS THAN M-REORDER-LEVEL
   THEN a) PERFORM "CALCULATE-REORDER-REQUIREMENTS (2 5)"

BOX NO. 20

CALL TO
DETERMINE-QUANTITY-BACK ORDER (2 1)
REDUCE-INVENTORY-ON-HAND (2 2)
UPDATE-TOTAL-SALES (2 3)
REVISE-ACTIVITY-DATE (2 4)
CALCULATE-REORDER-REQUIREMENTS (2 5)
INSTRUCTOR NOTES

NOTE THE DETAIL HERE, WHERE ESPECIALLY DATA IS BROKEN INTO ITS
CONSTITUENT AND PHYSICAL PARTS.

AT THIS LEVEL: FILES, FORMS AND RECORDS APPEAR.
AN OVERVIEW DIAGRAM

INPUT

- PROGRAM SOURCE CODE
- JOB CONTROL DATA
- OBJECT MODULES
- LOAD MODULES
- TEST DATA
- TEXTUAL DATA
- PROGRAM DESIGN STATEMENTS
- USER DATA

PROCESS

BASIC REQUIREMENTS

- PROVIDE DIRECT ACCESS STORAGE FOR
  - SOURCE CODE
  - PROGRAM MODULES
  - CONTROL DATA
  - TEST DATA
  - TEXTUAL DATA
  - PROGRAM DESIGN DATA
  - OTHER USER SPECIFIED DATA

FULL REQUIREMENTS

- COMPRESS DATA PRIOR TO STORING
- RESTORE COMPRESSED DATA TO ITS ORIGINAL FORM ON OUTPUT

OUTPUT

MIS DATA BASE

SOME PEOPLE USE A PDL HERE
INSTRUCTOR NOTES

GO OVER QUICKLY - POINT OUT HIGHLIGHTS.
<table>
<thead>
<tr>
<th>PROCESSING</th>
<th>INPUTS</th>
<th>OUTPUTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXECUTE INTERRUPT MANAGEMENT Monitor Clock Interrupt Handler</td>
<td>SAVE EXECUTION REGISTERS</td>
<td>EXAMPLIFY</td>
</tr>
<tr>
<td>TO HANDLE THE OCCURRENCE OF A MONITOR CLOCK INTERRUPT</td>
<td>SAVE STATUS REGISTERS AND PROGRAM COUNTER</td>
<td>EXAMINE RETURN LINKAGE</td>
</tr>
<tr>
<td></td>
<td>EXAMINE PROGRAM COUNTER</td>
<td>EXAMINE CLOCK</td>
</tr>
<tr>
<td></td>
<td>ENABLE INTERRUPTS</td>
<td>IF CURRENT MONITOR CLOCK TIME = NEXT MONITOR CLOCK TIME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>THEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>THEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ELSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDIF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDIF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDIF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
INSTRUCTOR NOTES

NSSF IS A WAY OF DESCRIBING THE ALGORITHMIC STRUCTURE OF A PROGRAM. IT WORKS WELL ONLY ON STRUCTURED PROGRAMMING.
NSSF OVERVIEW

- NASSI-SCHNEIDERMAN STRUCTURED FLOWCHARTS
  - DOCUMENT THE PROCEDURAL DETAILS OF SOFTWARE WITH
    IMPROVED FLOWCHART GRAPHICS

- BASED ON THE ASSUMPTION THAT FLOW CHARTING GRAPHICS CAN
  BE SIMPLIFIED TO REPRESENT STRUCTURED PROGRAMMING CONSTRUCTS

- NSSF GRAPHICS HELP DESIGNERS
  - REPRESENT PROGRAM STRUCTURES
  - IDENTIFY COMPLEXITY IN A MODULE
INSTRUCTOR NOTES

- COMPARE NSSF TO CONVENTIONAL FLOWCHART GRAPHICS.

- BE CAREFUL OF THE HIERARCHY EXAMPLE: THINK OF IT AS LOOKING "INSIDE" THE BIG BOX.
## NSSF Graphics
(REPRESENTING PROGRAM STRUCTURE)

<table>
<thead>
<tr>
<th>STRUCTURE</th>
<th>FLOWCHART GRAPHICS</th>
<th>NSSF GRAPHICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENCE</td>
<td><img src="image" alt="Flowchart for Sequence" /></td>
<td><img src="image" alt="NSSF for Sequence" /></td>
</tr>
<tr>
<td>SELECTION</td>
<td><img src="image" alt="Flowchart for Selection" /></td>
<td><img src="image" alt="NSSF for Selection" /></td>
</tr>
<tr>
<td>REPETITION</td>
<td><img src="image" alt="Flowchart for Repetition" /></td>
<td><img src="image" alt="NSSF for Repetition" /></td>
</tr>
<tr>
<td>HIERARCHY</td>
<td><img src="image" alt="Flowchart for Hierarchy" /></td>
<td><img src="image" alt="NSSF for Hierarchy" /></td>
</tr>
</tbody>
</table>
INSTRUCTOR NOTES

- TELL CLASS THAT NO ROOM TO WRITE MEANS DECOMPOSITION IS REQUIRED. THIS IS AN EXAMPLE OF A COMPLEXITY METRIC FEW TOOLS OR METHODS HAVE.
NSSF GRAPHICS
(IDENTIFYING COMPLEXITY)

CROWDED GRAPHICS INDICATES TOO MUCH COMPLEXITY -- TIME TO DECOMPOSE INTO A NEW MODULE.
INSTRUCTOR NOTES

NSSF IS VERY USEFUL IN TEACHING STRUCTURED PROGRAMMING AND FOR DOCUMENTATION. IT IS EXPENSIVE TO PRODUCE, THUS AN AUTOMATED TOOL IS NEEDED TO EXTRACT THE NSSF FROM THE CODE. WITHOUT SUCH A TOOL, IT WILL BE DIFFICULT TO KEEP THEM UP TO DATE.
NSSF SUMMARY

- Improves old flowchart graphics,
- Is good for module-sized software,
- Can identify complexity in a module.

Remember:

- There are several other structured alternatives to flowcharts.
INSTRUCTOR NOTES

THEME: TO PUT ALL OF THE DESIGN METHODS INTO PERSPECTIVE.

PURPOSE: TO PROVIDE A COMPARISON BETWEEN THE VARIOUS METHODS TO AID IN SELECTION OF A SET OF DESIGN METHODS.

REFERENCE: METHODMAN - SEE ANALYSIS WRAP-UP FOR REFERENCE.
INSTRUCTOR NOTES

MOST TECHNIQUES ARE USED OVER MORE THAN ONE PHASE, BUT ARE USUALLY AIMED AT ONE OR TWO.
DESIGN PHASE COVERAGE

DESIGN PHASE

- TRANSLATION OF REQUIREMENTS TO ARCHITECTURAL DESIGN
- ARCHITECTURAL DESIGN AND ASSESSMENT
- MODULAR STRUCTURE OF THE DESIGN
- DETAILED DESIGN
- INSIDES OF THE MODULES

EMPHASIS

SCRAP
OBJECT-ORIENTED DESIGN
STRUCTURED DESIGN
JACKSON DESIGN
WARNIER-ORR
HOS
HIPO
NSSF
PDL
METRICS

VG 778.1

23-1
INSTRUCTOR NOTES

THESE ARE THE DEFINITIONS OF DESIGN METHODS CHARACTERISTICS TO BE USED ON THE NEXT SLIDE. GO OVER THESE QUICKLY, BUT RELATE ONE OR TWO TO SPECIFIC TECHNIQUES BEFORE GOING TO THE CHART.
TECHNICAL CHARACTERISTICS
(DESIGN)

- FUNCTION HIERARCHY - A SITUATION CAN BE REPRESENTED IN WHICH A FUNCTION AT
ONE LEVEL IS ACTUALLY COMPOSED OF SEVERAL INTERCONNECTED FUNCTIONS THAT
EXIST AT A LEVEL OF GREATER DETAIL.

- DATA HIERARCHY - THE CONCEPT OF DATA CLASSES; A SITUATION CAN BE
REPRESENTED IN WHICH A SET OF DATA AT ONE LEVEL IS ACTUALLY COMPOSED OF
SEVERAL INTERRELATED PIECES OF DATA THAT EXIST AT A LEVEL OF GREATER DETAIL.

- INTERFACES - THE CONCEPT OF HAVING DISTINCT AND WELL-DEFINED BOUNDARIES
BETWEEN PROCESSES OR SETS OF DATA. SOFTWARE SYSTEMS AND THE LARGE
INFORMATION SYSTEMS OF WHICH THEY ARE A PART SHOULD CONTAIN MANY DISTINCT
PARTS. EACH PART SHOULD HAVE A CLEAR DEFINITION SO THAT IT CAN BE DEALT
WITH AS A SEPARABLE UNIT. IF THIS IS THE CASE, THEN WE HAVE INTERFACES, OR
CONNECTIONS, BETWEEN THE VARIOUS PARTS.

- CONTROL FLOW - REPRESENTATION OF THE SEQUENCE IN WHICH PROCESSES WILL TAKE
PLACE.

- DATA FLOW - REPRESENTATION OF THE FLOW OF INFORMATION TYPES BETWEEN VARIOUS
PROCESSING ELEMENTS AND/OR STORAGE ELEMENTS IN THE SYSTEM.

- DATA ABSTRACTION - THE CONCEPT OF HIDING INFORMATION ABOUT THE
IMPLEMENTATION OF A DATA TYPE AND PROVIDING A SET OF IMPLEMENTATION-
INDEPENDENT FUNCTIONS FOR USE OF THE DATA TYPE.

- PROCEDURAL ABSTRACTION - AN ALGORITHM FOR CARRYING OUT SOME OPERATION IS
ABSTRACTED TO A SINGLE NAME THAT CAN BE USED TO Invoke A PROCEDURE WITHOUT
KNOWING THE DETAILS OF ITS IMPLEMENTATION. (TRANSACTIONS ARE AN INSTANCE
OF THIS CASE.)

- PARALLELISM - A SITUATION IN WHICH TWO OR MORE COOPERATING SEQUENTIAL
PROCESSES ARE CONCURRENTLY IN EXECUTION.

- SAFETY - THE AVOIDANCE OF RUNTIME FAILURES WHICH COULD LEAD TO THE LOSS OF
LIFE OR THE OCCURRENCE OF OTHER CATASTROPHIC CONSEQUENCES.

- RELIABILITY - THE ABSENCE OF ERRORS THAT LEAD TO SYSTEM FAILURE.

- CORRECTNESS - FIDELITY TO HIGHER LEVEL SPECIFICATIONS.
INSTRUCTOR NOTES

PICK ONE OR TWO ROWS, THEN COLUMNS, AND INVESTIGATE IN DETAIL.
<table>
<thead>
<tr>
<th>TECHNICAL CHARACTERISTICS</th>
<th>_</th>
<th>_</th>
<th>_</th>
<th>_</th>
<th>_</th>
<th>_</th>
<th>_</th>
<th>_</th>
<th>_</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FUNCTION HIERARCHY</td>
<td>DATA HIERARCHY</td>
<td>INTERFACES</td>
<td>CONTROL FLOW</td>
<td>DATA FLOW</td>
<td>DATA ABSTRACTION</td>
<td>PROCEDURAL ABSTRACTION</td>
<td>PARALLELISM</td>
<td>CORRECTNESS</td>
</tr>
<tr>
<td>SCRIP</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>OBJECT-ORIENTED</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>M</td>
<td>L</td>
</tr>
<tr>
<td>STRUCTURED DESIGN</td>
<td>H</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>JACKSON DESIGN</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>L</td>
</tr>
<tr>
<td>WARNIER-ORR</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>HOS</td>
<td>H</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>H</td>
</tr>
<tr>
<td>HIPO</td>
<td>M</td>
<td>L</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>NSSF</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>PDL</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>M</td>
</tr>
<tr>
<td>METRICS</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>H</td>
</tr>
</tbody>
</table>

LEVEL OF SUPPORT
H - HIGH
M - MODERATE
L - LOW
BLANK - NO SUPPORT

VG 778.1 23-3
INSTRUCTOR NOTES

AGAIN, GO THROUGH THESE QUICKLY, AND PICK ONE OR TWO AND THE APPROPRIATE TECHNIQUES TO ILLUSTRATE AND MOTIVATE THE NEXT SLIDE.
USAGE CHARACTERISTICS
(DESIGN)

- UNDERSTANDABILITY - HOW EASY IT IS FOR SOMEONE WHO IS INTERESTED IN THE SYSTEM
  BEING DEVELOPED, BUT NOT ESPECIALLY KNOWLEDGEABLE IN THE TECHNIQUE, TO UNDERSTAND
  THE RESULTS OF THE DEVELOPMENT.

- TRANSFERABILITY - THE DEGREE TO WHICH THE METHOD OR TOOL CAN BE SUCCESSFULLY
  TAUGHT TO PERSONNEL NOT PREVIOUSLY FAMILIAR WITH IT. THIS INCLUDES NOT ONLY HOW
  EASY IT IS TO TEACH AND LEARN, BUT ALSO HOW WELL FORMULATED IT IS.

- REUSABILITY - THE EASE WITH WHICH PREVIOUSLY CREATED DESIGNS, CODE, OR OTHER WORK
  PRODUCTS CAN BE REUSED IN A NEW PROJECT.

- COMPUTER SUPPORT - EXISTENCE OF AUTOMATED TOOLS WHICH CAN BE EASILY OBTAINED AND
  WHICH AID IN THE USE OF THE METHODOLOGY OR SOME OF ITS STEPS.

- LIFE-CYCLE RANGE - THE SPAN OF PHASES IN THE DEVELOPMENT LIFE-CYCLE OVER WHICH THE
  TOOLS OR METHOD CAN BE USEFULLY APPLIED. THIS MUST BY NECESSITY BE AN APPROXIMATE
  MEASURE.

- TASK RANGE - THE SPAN OF TASKS TO WHICH THE ITEM MAY BE USEFULLY APPLIED.
  TRADITIONAL CLASSIFICATIONS OF SOFTWARE APPLICATIONS (BUSINESS, SCIENTIFIC,
  SYSTEMS, ETC.) ARE NOT VERY USEFUL FOR THIS EVALUATION. THERE ARE SCIENTIFIC AND
  BUSINESS PROBLEMS THAT SHARE THE SAME PROBLEMS WHILE THERE ARE OTHER TASKS THAT
  SAY, WITHIN THE BUSINESS CATEGORY, ARE QUITE DISSIMILAR.

- COHESIVENESS - THE EXTENT TO WHICH THE TECHNICAL METHODS, MANAGEMENT PROCEDURES,
  AND AUTOMATED TOOLS MAY BE COMBINED TO SUPPORT THE METHODOLOGY.

- EXTENT OF USAGE - A JUDGEMENT AS TO HOW WIDESPREAD THE CURRENT USAGE OF THE
  METHODOLOGY IS.
USAGE CHARACTERISTICS
(DESIGN) - continued

- **Ease of Phase Transition** - The extent to which information developed at one phase of development supports work to be done at a subsequent phase (e.g., from Analysis to Design).

- **Decision Highlighting** - The ability of a technique to make visible and highlight key technical or project decisions (e.g., choice of data structures, nearness to completion of a project); the ability of a technique to illuminate the consequences of a development decision.

- **Validation** - The extent to which the methodology assists in the determination of system correctness.

- **Repeatability** - The extent to which similar results are obtained when the methodology (or an included aspect) is applied more than once to the same problem.

- **Ease of Change to Work Products** - The amount of effort required to modify a work product when some aspect of requirements, specification, or design is changed.
INSTRUCTOR NOTES

PICK ONE OR TWO ROWS, THEN COLUMNS, AND INVESTIGATE IN DETAIL.

NOTE, THIS CHART EMPHASIZES SUITABILITY.
## USAGE CHARACTERISTICS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SCRIP</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>I</td>
<td>S</td>
<td>W</td>
</tr>
<tr>
<td>OBJECT-ORIENTED</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>I</td>
<td>S</td>
<td>W</td>
<td>S</td>
<td>I</td>
</tr>
<tr>
<td>STRUCTURED DESIGN</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>S</td>
</tr>
<tr>
<td>JACKSON DESIGN</td>
<td>W</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>WARNIER-ORR</td>
<td>W</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>I</td>
</tr>
<tr>
<td>HOS</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>S</td>
<td>W</td>
<td>I</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>HIPO</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>NSSF</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>S</td>
<td>I</td>
</tr>
<tr>
<td>PDL</td>
<td>W</td>
<td>W</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>METRICS</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>W</td>
<td>S</td>
</tr>
</tbody>
</table>

**Legend:**

- **W** - Well Suited
- **S** - Satisfactory
- **I** - Inappropriate
### Usage Characteristics

<table>
<thead>
<tr>
<th>Method</th>
<th>Understandability</th>
<th>Transferability</th>
<th>Reusability</th>
<th>Computer Support</th>
<th>Extent of Usage</th>
<th>Ease of Phase Transition</th>
<th>Decision Highlighting</th>
<th>Validation</th>
<th>Repeatability</th>
<th>Ease of Change of Work Products</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCRIP</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>M</td>
</tr>
<tr>
<td>OBJECT-ORIENTED</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>M</td>
</tr>
<tr>
<td>STRUCTURED DESIGN</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>L</td>
</tr>
<tr>
<td>JACKSON DESIGN</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>L</td>
</tr>
<tr>
<td>WARNIER-ORR</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>L</td>
</tr>
<tr>
<td>HOS</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L, L</td>
</tr>
<tr>
<td>HIPO</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>NSSF</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>PDL</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>METRICS</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Level of Support**
- **H** - High
- **M** - Moderate
- **L** - Low
- **Blank** - No Support

VG 778.1 23-7
MANAGEMENT CHARACTERISTICS
(DESIGN)

• MANAGEABILITY - THE DEGREE TO WHICH THE METHOD OR TOOL PERMITS STANDARD MANAGEMENT
  TECHNIQUES OF ESTIMATION, IN-PROCESS STATUS CHECKING AND CONTROL TO BE APPLIED.
  THE EVALUATIONS ARE BASED ON THE EXISTENCE OF WELL-DEFINED STEPS AND INTERMEDIATE
  PRODUCTS WHICH MAKE MANAGEMENT POSSIBLE AND THE EXISTENCE OF MANAGEMENT TECHNIQUES
  APPLYING SPECIFICALLY TO THE METHOD AT HAND.

• TEAMWORK - THE EXTENT TO WHICH THE METHODOLOGY AND THE DEVELOPMENT ENVIRONMENT
  AID, RATHER THAN HINDER, TEAMWORK.

• PHASE DEFINITIONS - THE IDENTIFICATION WITHIN THE METHODOLOGY OF DEVELOPMENT
  PHASES THAT REPRESENT INTERMEDIATE STAGES OF THE DEVELOPMENT PROCESS.

• WORK PRODUCTS - THE DOCUMENTS AND SYSTEMS THAT RESULT FROM APPLICATION OF THE
  METHODOLOGY.

• CONFIGURATION MANAGEMENT - THE WAY IN WHICH THE METHODOLOGY AND/OR ITS TOOLS
  PROVIDES FOR ORGANIZATION, TRACKING, AND MAINTENANCE OF THE EMERGING WORK
  PRODUCTS, INCLUDING CONTROL OF RELEASES AND MULTIPLE VERSIONS.

• EXIT CRITERIA - THE WAY IN WHICH PHASES OF DEVELOPMENT AND WORK PRODUCTS ARE
  DEFINED TO PROVIDE EXPLICIT STOPPING OR EXIT CRITERIA FOR EACH STAGE OF
  DEVELOPMENT.

• SCHEDULING - THE METHODOLOGICAL SUPPORT FOR PROJECT SCHEDULING.

• COST ESTIMATION - THE METHODOLOGICAL SUPPORT FOR COST ESTIMATION.
INSTRUCTOR NOTES

TRY TO GET THE STUDENTS TO ANSWER "HOW DO THESE TECHNIQUES HELP MANAGEMENT OF A PROJECT?"
### MANAGEMENT CHARACTERISTICS

<table>
<thead>
<tr>
<th></th>
<th>Manageability</th>
<th>Teamwork</th>
<th>Phase Definition</th>
<th>Configuration Management</th>
<th>Exit Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCRP</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>H</td>
</tr>
<tr>
<td>OBJECT-ORIENTED</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>STRUCTURED DESIGN</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>JACKSON DESIGN</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>WARNIER-ORR</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>HOS</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>H</td>
</tr>
<tr>
<td>HIPO</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>NSSF</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>PDL</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>H</td>
</tr>
<tr>
<td>METRICS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LEVEL OF SUPPORT**

- **H** - HIGH
- **M** - MODERATE
- **L** - LOW
- **BLANK** - NO SUPPORT

VG 778.1 23-9
### SUPPORT OF Ada UNDERLYING CONCEPTS AND FEATURES

<table>
<thead>
<tr>
<th></th>
<th>USE OF STRUCTURED CONSTRUCTS</th>
<th>STRONG-TYPING DATA-TYPING</th>
<th>INFORMATION Hiding</th>
<th>DATA ABSTRACTION</th>
<th>CONCURRENT PROGRAMMING</th>
<th>EXCEPTION HANDLING</th>
<th>GENERIC FACILITIES</th>
<th>MACHINE DEPENDENCY FACILITIES</th>
<th>PACKAGES</th>
<th>ABSOLUTE AND RELATIVE PRECISION</th>
<th>ORIENTED TOWARD EMBEDDED SYSTEMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCRJ</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>OBJECT-ORIENTED</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>H</td>
</tr>
<tr>
<td>STRUCTURED DESIGN</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>JACKSON DESIGN</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>WARNER-ORR</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>HOS</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>HIP0</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>NSSF</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>PDL</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>

**LEVEL OF SUPPORT**
- **H** - HIGH
- **M** - MODERATE
- **L** - LOW
- **BLANK** - NO SUPPORT
INSTRUCTOR NOTES

ENTERTAIN QUESTIONS, BUT DON'T GET STUCK ON DETAILS. KEEP TALK ON A MORE GENERAL LEVEL RATHER THAN SPECIFIC.
REMINDEERS

- ITERATE - WORK IN THE GOOD IDEAS, WORK OUT THE BAD

- ABSTRACT - SEPARATE SIMILARITIES AND DIFFERENCES

- MECHANIZE - BUILD WORKHORSES

- DECOMPOSE - BREAK INTO MODULES CONSISTENTLY

- SYNTHESIZE - BRING TOGETHER SIMILAR FUNCTIONS AND STRUCTURES

- USE METRICS - TO EVALUATE THE QUALITY OF YOUR DESIGN

- DOCUMENT - EVERYTHING!
THEME: IMPLEMENTATION IS MORE THAN JUST CODING!

PURPOSE: TO IDENTIFY THE KEY ISSUES ONE MUST CONSIDER DURING THE IMPLEMENTATION PHASE OF THE LIFE CYCLE.

REFERENCE: DOD-STD-SDS
INSTRUCTOR NOTES

The implementation phase should create a system which implements the design correctly and meets the resource, accuracy, and performance constraints in the specification. Implementation translates design into the language of the target computer, the database, and other products needed to create the operational system.
IMPLEMENTATION PHASE

- THE IMPLEMENTATION PHASE CONSISTS OF
  - CODING (I.E. TRADITIONAL PROGRAMMING)
  - UNIT TESTING
  - SOFTWARE INTEGRATION AND TEST
  - SYSTEM TESTING

- WHEN OTHERS TALK OF SOFTWARE DEVELOPMENT THEY NORMALLY MEAN IMPLEMENTATION
  - CURRENTLY WE SPEND A LOT OF TIME TRASHING IN THIS PHASE

- IF ANALYSIS AND DESIGN ARE DONE PROPERLY, IMPLEMENTATION IS A WELL DEFINED EASY PROCESS
INSTRUCTOR NOTES

EACH OF THESE TECHNIQUES CAN HELP CREATE CLEAR, UNDERSTANDABLE PROGRAMS.
IMPLEMENTATION SCOPE

• THE IMPLEMENTATION PHASE CONCERNS ITSELF WITH
  ISSUES LIKE ...

  - STEP-WISE PROGRAM COMPOSITION
  - PROGRAM FAMILIES
  - DATA ABSTRACTIONS AND TYPES
  - DATA STRUCTURES
  - FUNDAMENTAL ALGORITHMS
  - TIME/SPACE TRADEOFFS
  - ACCEPTABILITY VS. CORRECTNESS
INSTRUCTOR NOTES

HERE, THIS VISUAL TABLE OF CONTENTS DEPICTS THE SUBPROGRAMS CREATED DUE TO STEP-WISE REFINEMENT. WE'VE ALREADY SEEN THIS CONCEPT IN THE DESIGN PHASE.
IMPLEMENTATION ISSUES
(STEP-WISE REFINEMENT)

- **STEP-WISE PROGRAM COMPOSITION (REFINEMENT)** STATED THAT PROGRAMS SHOULD BE WRITTEN IN LEVELS; THE HIGHEST LEVEL CALLING ONE OR MORE SUBPROGRAMS AT THE NEXT LOWEST LEVEL:

  ![Diagram of level structure]

- EARLY PROGRAMMING WAS CONTENT TO JUST WRITE A CALL STATEMENT: THAT ALONE WAS CONSIDERED THINKING IN STEP-WISE REFINEMENT TERMS.

- TODAY, STEP-WISE REFINEMENT IS USED PRIMARILY IN THE DESIGN PHASE, WHERE THE TECHNIQUE SHAPES THE SYSTEM'S ARCHITECTURE.
INSTRUCTOR NOTES

HERE, THE NOTATION 1.1 MEANS VERSION 1 PROGRAM 1. THE NOTATION "2" MEANS THIS IS A NEW PROGRAM, SPAWNED FROM ITS ANCESTOR (PROGRAM 1) BY THE CHANGING OF A DESIGN DECISION.

- PROGRAM FAMILIES WORK AT THE MODULE LEVEL. IT DESCRIBES AMOUNT AND TYPES OF IMPACTS WHICH ARE ON THE STRUCTURE OF THE PROGRAM.

- STRUCTURED PROGRAMMING AND PROGRAM FAMILIES ARE NEITHER EQUIVALENT OR CONTRADICTORY. BOTH USE STEP-WISE REFINEMENT TO ENCOURAGE ONE TO MAKE DESIGN DECISIONS EARLY ON.
IMPLEMENTATION ISSUES
(PROGRAM FAMILIES)

- PROGRAM FAMILIES WAS A CONCEPT THAT STATED THAT A PROGRAM EVOLVES INTO A SET OF RELATED PROGRAMS:

```
  1.0  
   /  
  1.1 1.2
   /   
 2.0  2.1  2.2
     /   /   
2.3  2.4  2.5
```

O = PROGRAM VERSION
X = DESIGN DECISION

- VERY FEW HAVE WORRIED ABOUT PROGRAM FAMILIES. BUT TODAY'S CONCERNS ABOUT THE EVOLUTION OF INDIVIDUAL MODULES AS THEY SIMULTANEOUSLY PARTICIPATE IN SEVERAL SOFTWARE PRODUCTS BRINGS THIS ISSUE AGAIN INTO CENTER STAGE.
INSTRUCTOR NOTES

A PICTURE OF THESE TYPES COULD BE:

- Card suits: ♠️, ♦️, ♥️, ♣️
- Days of the week:
  - MON, TUES, THURS, FRI
  - WED, SAT, SUN

Boolean: 0, 1
IMPLEMENTATION ISSUES
(DATA ABSTRACTIONS AND TYPES)

DATA ABSTRACTIONS AND TYPES CATEGORIZED PROGRAM VARIABLES AND THEIR VALUES INTO COLLECTIONS HAVING STRONGLY-RELATED CHARACTERISTICS:

TYPE SUIT = (CLUB, DIAMOND, HEART, SPADE);

TYPE DAY = (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY);

TYPE BOOLEAN = (TRUE, FALSE);
INSTRUCTOR NOTES

DATA STRUCTURES IMPLY RULES ON HOW TO ACCESS THE DATA THEY CONTAIN.
IMPLEMENTATION ISSUES
(DATA STRUCTURES)

DATA STRUCTURES CONCERNS ITSELF WITH ORGANIZING DATA IN WAYS THAT COMPUTERS CAN PROCESS. FOR EXAMPLE ...

- SCALARS - SINGLE ITEMS (ATOMS) HAVING ONLY A SINGLE VALUE AT ANY GIVEN TIME

- RECORDS - A LINEAR COLLECTION OF SCALARS OF DIFFERENT TYPE

- ARRAYS - A LINEAR COLLECTION OF SCALARS OF THE SAME TYPE

- STACK - A DYNAMIC ARRAY WHERE ELEMENTS ARE TAKEN OFF OF, AND PUT ON AT, THE SAME END

- QUEUE - A DYNAMIC ARRAY WHERE ELEMENTS ARE PUT ON AT ONE END AND TAKEN OFF THE OTHER END
IMPLEMENTATION ISSUES
(FUNDAMENTAL ALGORITHMS)

- FUNDAMENTAL ALGORITHMS CONCERNED THEMSELVES WITH STANDARD WAYS OF CALCULATING NUMBERS AND MANIPULATING DATA STRUCTURES. SOME OF THEM WERE ...

  - FIBONACCI SEQUENCE GENERATION
  - FACTORIAL COMPUTATION
  - MATRIX MULTIPLICATION
  - LINKED-LIST ALLOCATION
  - TREE MANIPULATION
  - MONITORS
  - INSERTION SORT
  - QUICK SORT
IMPLEMENTATION ISSUES
(TIME/SPACE TRADEOFFS)

THE TIME/SPACE TRADEOFF MADE CLEAR THAT MORE MEMORY USAGE MEANT
FASTER PROGRAM EXECUTION, AND LESS MEMORY USAGE MEANT PROGRAMS
WORKED MORE SLOWLY. THIS HAS ALWAYS REMAINED TRUE.

AVAILABLE MEMORY

EXECUTION TIME
INSTRUCTOR NOTES

SPEND SOME TIME ON THIS SLIDE. IT IS IMPORTANT.

TALK ABOUT THE GUIDANCE AVAILABLE TODAY (E.G. ON SOFTWARE COST MODELS) TO MAKE THESE TRADE-OFFS.
IMPLEMENTATION ISSUES

(EFFICIENCY)

BUT MANY HAVE TAKEN THE CONCEPT AND MADE EFFICIENCY OF PARAMOUNT IMPORTANCE. THIS IS FALLACIOUS, FOR EFFICIENCY IS JUST ANOTHER ENGINEER GOAL. IT MUST BE EVALUATED WITH, AND TRADEOFF AMONG, THE OTHER SOFTWARE ENGINEERING GOALS:
INSTRUCTOR NOTES

IT COSTS A GREAT DEAL TO GET MORE THAN A MINIMAL LEVEL OF ACCEPTABILITY. ATTAINING CORRECTNESS IS IMPOSSIBLE THESE DAYS.
ACCEPTABILITY VS. CORRECTNESS

- THE ISSUE OF ACCEPTABILITY STATED THAT, SINCE CORRECTNESS COULD NOT BE ATTAINED OF SOFTWARE, TESTING UP TO A LEVEL OF ACCEPTABILITY WAS DEEMED AN ETHICAL PROGRAMMING PRACTICE. THE SAME IS TRUE TODAY.

- THIS IS WHY TESTING, V&V AND QUALITY ASSURANCE HAVE EVOLVED INTO DISCIPLINES OF THEIR OWN -- TO MEET THE NEED OF VALIDATING PROGRAMS TO THE HIGHEST DEGREE OF ACCEPTABILITY POSSIBLE.
INSTRUCTOR NOTES

THIS IS SDS'S REQUIREMENTS FOR IMPLEMENTATION. NOTE HOW THEY RELATE TO THE PREVIOUS TOPICS.
DOD-STD-SDS VIEW OF IMPLEMENTATION

SDS IMPLEMENTATION ACTIVITIES

- Units are coded in a systematic, top-down hierarchical sequence
- Must code units in accordance coding standards
- Maintain unit development folders on all units
- Test each unit
- Develop integration test procedures
- Develop procedures for formal (acceptance) testing
- Integrate units in a sequence by which top-level software are integrated before lower-level
- Test integrate item in accordance with software test plan
DOOD-STD-SDS VIEW OF IMPLEMENTATION

- SDS IMPLEMENTATION PRODUCTS
  - SOURCE AND OBJECT FOR EACH UNIT
  - SOFTWARE TEST PROCEDURES (FORMAL TESTS)
  - UNIT DEVELOPMENT FOLDERS
  - TEST REPORTS
  - SOFTWARE PRODUCT SPECIFICATION

- SDS IMPLEMENTATION REVIEWS
  - REVIEW OF SOURCE CODE, TEST PROCEDURES AND TEST RESULTS
  - FUNCTIONAL CONFIGURATION AUDIT
    - DEMONSTRATES THAT SOFTWARE WAS TESTED AND MEETS ALL REQUIREMENTS IN SOFTWARE SPECIFICATIONS
  - PHYSICAL CONFIGURATION AUDIT
    - DEMONSTRATES THAT SOFTWARE PRODUCT SPECIFICATION IS COMPLETE
INSTRUCTOR NOTES

ASK THE CLASS HOW THEY FEEL ABOUT THE CODING STANDARDS. NOTE, STRUCTURED PROGRAMMING IS COMING IN THE NEXT SECTION.
DOD-STD-SDS MINIMUM LEVEL CODING STANDARDS

- **ALL CODE SHALL BE WRITTEN IN HIGHER LEVEL LANGUAGE**
  - WAIVER REQUIRED FOR USE OF ASSEMBLY LANGUAGE
  - EVEN IF YOU GET A WAIVER THE OTHER STANDARDS MUST BE MET

- **ONLY FIVE CONTROL CONSTRUCTS SHALL BE USED**
  - SEQUENCE
  - IF-THEN-ELSE
  - DO-WHILE
  - DO-UNTIL
  - CASE

- **SOURCE CODE FOR EACH UNIT SHALL NOT EXCEED ...**
  - ON THE AVERAGE, 100 STATEMENTS
  - AT MOST, 200 STATEMENTS
INSTRUCTOR NOTES

ASK THE CLASS "DO YOU THINK THESE STANDARDS ARE NECESSARY OR USABLE?"
CODING STANDARDS
(CONTINUED)

- UNITS SHALL EXHIBIT THE FOLLOWING CHARACTERISTICS:
  - UNITS SHALL NOT SHARE TEMPORARY STORAGE LOCATIONS
  - A UNIT SHALL PERFORM A SINGLE FUNCTION
  - CODE SHALL NOT BE MODIFIED DURING EXECUTION
  - ALL UNIT SOURCE CODE SHALL INCLUDE A PROLOGUE, DECLARATIVE
    STATEMENTS THEN EXECUTABLE STATEMENTS OR COMMENTS
  - EXCEPT FOR ERROR EXITS, UNITS SHALL HAVE A SINGLE ENTRY AND A SINGLE
    EXIT

- USE MEANINGFUL NAMES AND MAKE ALL PARAMETERS SYMBOLIC

- AVOID MIXED MODE OPERATIONS

- USE PARAGRAPHING, BLOCKING, AND INDENTING TO ENHANCE READABILITY
UNIT DEVELOPMENT FOLDERS

- PROVIDES A UNIFORM AND VISIBLE COLLECTION POINT FOR ALL UNIT INFORMATION

SOURCE: BEOHM, SOFTWARE ENGINEERING ECONOMICS, 1983

VG 778.1   24-15
IMPLEMENTATION PERSPECTIVES AND FORMATS

- PRIMARY PERSPECTIVES CONSIDERED DURING IMPLEMENTATION
  - CODING STRATEGIES
  - COMPLEXITY MANAGEMENT
  - QUALITY ASSURANCE
  - TESTING STRATEGIES

- PRIMARY FORMATS
  - GUIDELINES
  - MEETING-ORIENTED
  - PROCEDURAL
IMPLEMENTATION METHODOLOGIES TO BE PRESENTED

<table>
<thead>
<tr>
<th>METHODOLOGY</th>
<th>PERSPECTIVE</th>
<th>FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRUCTURED PROGRAMMING</td>
<td>CODING STRATEGIES</td>
<td>GUIDELINES</td>
</tr>
<tr>
<td>PROGRAM COMPLEXITY</td>
<td>COMPLEXITY MANAGEMENT</td>
<td>GUIDELINES</td>
</tr>
<tr>
<td>PROGRAM CORRECTNESS</td>
<td>QUALITY ASSURANCE</td>
<td>PROCEDURAL</td>
</tr>
<tr>
<td>TESTING</td>
<td>TESTING STRATEGIES</td>
<td>GUIDELINES</td>
</tr>
<tr>
<td>REVIEW</td>
<td>QUALITY ASSURANCE</td>
<td>MEETING ORIENTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GUIDELINES</td>
</tr>
</tbody>
</table>

VG 778.1 24-17
INSTRUCTOR NOTES

THEME: TO RELATE THE EARLY 70'S CONCEPTS OF STRUCTURED PROGRAMMING TO MODERN REQUIREMENTS AND Ada.

PURPOSE: TO PROVIDE A LIMITED VIEW INTO STRUCTURED PROGRAMMING.

REFERENCE: WIRTH, N. "PROGRAM DEVELOPMENT BY STEPWISE REFINEMENT" CACM VOL. 14, NO. 4; DECEMBER 1971

DAHL, O., DIJKSTRA, E., HOARE, C. "STRUCTURED PROGRAMMING" ACADEMIC PRESS, LONDON; 1972
INSTRUCTOR NOTES

- GIVE SOME MOTIVATION BY DESCRIBING IN GENERAL WHAT STRUCTURED PROGRAMMING IS ABOUT AND WHY IT'S USEFUL. A FEW DIFFERENT DEFINITIONS MAY BE HELPFUL. FOR EXAMPLE,

- EMPHASIZE THAT STRUCTURED PROGRAMMING IS A METHODOLOGY. USING UNIQUE CONTROL STRUCTURES (AND NO GO TO'S) AND NESTING CODE DO NOT BY THEMSELVES CONSTITUTE STRUCTURED PROGRAMMING. ONE ALSO NEEDS AN ORGANIZATIONAL METHOD TO COLLECTIVELY USE THESE TECHNIQUES BY EVERYONE ON A PROJECT, SUCH AS IBM'S PROGRAMMING TEAMS.

- IT IS LANGUAGE INDEPENDENT, BUT CERTAIN LANGUAGES HELP MORE THAN OTHERS, E.G. Ada. STEP-WISE REFINEMENT IS THE KEY.

- IT IS NOT A RELIGION (OF NOT USING GO TO'S). IT IS MEANT TO ENHANCE READABILITY, RELIABILITY, PROGRAMMER EFFICIENCY, ETC. THE ABSENCE OR RARE OCCURRENCE OF GO TO'S IS NO MORE THAN A SYMPTOM OF STRUCTURED PROGRAMMING.
STRUCTURED PROGRAMMING

MOTIVATION

- PROGRAMMING IS A MODELING ACTIVITY AND MODELS MUST BE REVIEWED BY PEOPLE ...
- TO VERIFY CORRECTNESS OF THE APPROACH
- TO FIND ERRORS
- TO SHARE TECHNIQUES

- PROGRAMS MUST BE DESIGNED AND IMPLEMENTED TO BE READ AND UNDERSTOOD BY PEOPLE, NOT JUST COMPUTERS.

- STRUCTURED PROGRAMMING IS A COLLECTION OF TECHNIQUES WHICH EVOLVED TO ANSWER CONCERNS ABOUT THE PROGRAMS WHICH PEOPLE WRITE.
SOFTWARE IS ALWAYS READ MORE OFTEN THAN IT IS WRITTEN.

STRUCTURED PROGRAMMING KEEPS THINGS SIMPLE, THUS ENHANCING THE LIKELIHOOD THAT THINGS WILL BE CORRECT.

THIS DIAGRAM REQUIRES A GREAT AMOUNT OF WORK TO DETERMINE WHAT CONDITION HOLDS AT A SPECIFIC POINT. MOREOVER, IT'S NOT CLEAR WHERE THAT POINT EXISTS.
MOTIVATION

- Early programs were just written with no thought of human concerns. A typical flow diagram of such a program looked like spaghetti...

- Structured programming provides rules/guidelines for "structuring" programs.
INSTRUCTOR NOTES

ALL THE PREVIOUS METHODOLOGIES HAVE AS THEIR ROOTS STRUCTURED PROGRAMMING. THE FOUNDER IS EDSPER DIJKSTRA. ALTHOUGH IT STARTED OUT AS A REACTION AGAINST "GO TO" TYPE PROGRAMMING, IT NOW HAS NO STANDARD DEFINITION.
STRUCTURED PROGRAMMING

A DEFINITION ...

"STRUCTURED PROGRAMMING IS THE ART OF ORGANIZING COMPLEXITY, MASTERING MULTITUDES, AND ITS RESULTING BASTARD CHAOS AS EFFECTIVELY AS POSSIBLE."

DIJKSTRA
ALL THE FACTORS OF RELIABILITY, COST, AND COMPLEXITY ARE SKYROCKETING; EVEN TODAY.
STRUCTURED PROGRAMMING

SCOPE

- STRUCTURED PROGRAMMING DEALS WITH MORE THAN STRUCTURED CONTROL FLOWS...

- STRUCTURED PROGRAMMING EVOLVED FROM THE NEED TO MAKE RELIABLE AND COST-EFFECTIVE SOFTWARE.

- Ada'S SCOPE PARTLY EVOLVED FROM STRUCTURED PROGRAMMING CONCEPTS
  - Ada EXTENDED THE CONCEPTS

VG 778.1
ANY PROGRAM CAN BE WRITTEN USING 3 BASIC CONTROL STRUCTURES.

CONTROL STRUCTURING
RULES/GUIDELINES

- BOHM AND JACCOPINNI PROVED THAT ANY LOGIC FLOW COULD BE REDUCED TO THREE SIMPLE RULES ...

- THE THREE BASIC STRUCTURING RULES THEY STATED WERE:

  [Diagram of Sequence, Selection, Repetition]

- AND ANY OF THESE RULES CAN BE NESTED HIERARCHICALLY ...
INSTRUCTOR NOTES

REMEMBER, SDS ONLY ALLOWS 5 CONTROL CONSTRUCTS.
DOD-STD-SDS AND STRUCTURE PROGRAMMING

(SEQUENCE)

ENTER → A → B → EXIT

SEQUENCE CONSTRUCT

SOURCE: DOD-STD-SDS, 1984
INSTRUCTOR NOTES

NOTICE ESPECIALLY THE CASE STATEMENT. ALL CONDITIONS MUST BE SPECIFIED. THERE IS NO DEFAULT CONDITIONS. CERTAIN IMPLEMENTATIONS OF PASCAL COULD NOT MEET THE SDS STANDARD.
INSTRUCTOR NOTES

POINT OUT THE DIFFERENCES TO THE CLASS, AND HOW THEY ARE BOTH DIFFERENT TO A WHILE-DO (TO BE SEEN LATER).
INSTRUCTOR NOTES

THIS IS A RATHER OBVIOUS BUT IMPORTANT EXAMPLE OF A CASE CONSTRUCT. SPEND A LITTLE TIME HERE AND SEE IF ALL CONDITIONS ARE SPECIFIED.
Ada AND STRUCTURED PROGRAMMING
(SEQUENCE)

GET(USER_COMMAND);
NEW_LINE;
PUT_LINE("Command accepted");
case USER_COMMAND is
  when DISABLE =>
    PUT("Enter sensor name:");
    GET(NAME);
    NEW_LINE;
    COLLECTION_OF_SENSORS.DISABLE(SENSOR => NAME);
    PUT_LINE("Sensor disabled");
  when ENABLE =>
    PUT("Enter sensor name:");
    GET(NAME);
    NEW_LINE;
    COLLECTION_OF_SENSORS.ENABLE(SENSOR => NAME);
    PUT_LINE("Sensor enabled");
  when RECORD STATUS =>
    PUT("Enter sensor name:");
    GET(NAME);
    NEW_LINE;
    COLLECTION_OF_SENSORS.FORCE_RECORD (OF_SENSOR => NAME);
    PUT_LINE("Sensor status set");
  when SET_LIMITS =>
    PUT("Enter sensor name:");
    GET(NAME);
    NEW_LINE;
    PUT("Enter lower limit:");
    GET (LOW_BOUND);
    NEW_LINE;
    PUT_LINE("Lower limit accepted");
    PUT("Enter upper limit:");
    GET(HIGH_BOUND);
    NEW_LINE;
    PUT_LINE("Upper limit accepted");
    COLLECTION_OF_SENSORS.SET_THE_LIMITS
      (FOR_SENSOR => NAME,
       LOW_LIMIT => LOW_BOUND,
       HIGH_LIMIT => HIGH_BOUND);
    PUT_LINE("Limits set");
end case;
INSTRUCTOR NOTES

ASK CLASS,

"WHEN WOULD YOU USE ONE OVER THE OTHER?"
Ada AND STRUCTURED PROGRAMMING
(SELECTION)

IF STATEMENT

if VALVE_RECORD(1).OPEN then
   VALVE_RECORD(2).OPEN := TRUE;
   VALVE_RECORD(3).OPEN := FALSE;
else
   VALVE_RECORD(2).OPEN := FALSE
   VALVE_RECORD(3).OPEN := TRUE;
end if;

CASE STATEMENT

case PROCESS_STATE is
   when RUNNING => SCHEDULER_TABLE(RUNNING) := 1
                  IS_ACTIVE := TRUE;
   when READY   => SCHEDULER_TABLE(READY) :=
                  SCHEDULER_TABLE(READY) + 1;
                  IS_ACTIVE := FALSE;
   when BLOCKED => SCHEDULER_TABLE(BLOCKED) :=
                  SCHEDULER_TABLE(BLOCKED) + 1;
                  IS_ACTIVE := FALSE;
   when DEAD    => SCHEDULER_TABLE(DEAD) :=
                  SCHEDULER_TABLE(DEAD) + 1;
                  IS_ACTIVE := FALSE;
end case;

VG 778.1

25-10
INSTRUCTOR NOTES

EXPLAIN THE DIFFERENCES CAREFULLY.
Ada AND STRUCTURED PROGRAMMING
(REPETOITION)

FOR LOOP

for INDEX in RUNNING .. DEAD
loop
  SCHEDULER_TABLE(INDEX) := 0
end loop;

for INDEX in reverse TOTAL_VALVES
loop
  VALVE_RECORD(INDEX).OPEN := FALSE;
end loop;

WHILE_LOOP

while not END_OF_FILE(DATA_FILE)
loop
  ACTIVE_RECORDS := ACTIVE_RECORDS + 1;
  READ(DATA_FILE,ITEM => DATA(INDEX));
end loop;

• FOR LOOP IS THE MORAL EQUIVALENT
  OF A DO LOOP
INSTRUCTOR NOTES

DON'T GET BOGGED DOWN HERE. JUST ILLUSTRATE ONE OR TWO FUNCTION CALLS.
Ada AND STRUCTURED PROGRAMMING

- Ada GOES BEYOND JUST STRUCTURED CONTROL; IT ALSO PROVIDES STRUCTURED DATA

- ABSTRACT DATA TYPE EXAMPLE OF STRUCTURED DATA

```ada
package body COMPLEX is
  function "+" (A,B : in NUMBER) return NUMBER is
    RESULT : NUMBER
  begin
    RESULT.REAL_PART := A.REAL_PART + B.REAL_PART;
    RESULT.IMAGINARY_PART := A.IMAGINARY_PART + B.IMAGINARY_PART;
    return RESULT;
  end "+";

  function "-" (A, B : in NUMBER) return NUMBER is
    begin
      return NUMBER'(REAL_PART => A.REAL_PART - B.REAL_PART,
                      IMAGINARY_PART => A.IMAGINARY_PART - B.IMAGINARY_PART);
    end "-";

  function "*" (A, B : in NUMBER) return NUMBER is
    begin
      RESULT.REAL_PART := (A.REAL_PART * B.REAL_PART) -
                           (A.IMAGINARY_PART * B.IMAGINARY_PART);
      RESULT.IMAGINARY_PART := (A.REAL_PART * B.IMAGINARY_PART) +
                              (A.IMAGINARY_PART * B.REAL_PART);
      return RESULT;
    end "*";
end COMPLEX;
```

VG 778.1 25-12
INSTRUCTOR NOTES

Most people today agree that structured programming is a set of tools and techniques to simplify programs and therefore increase reliability.
SUMMARY

- THE GOAL OF STRUCTURED PROGRAMMING HAS ALWAYS BEEN TO DEVELOP A STANDARD SET OF RULES FOR STRUCTURING DATA AND ALGORITHMS INTO PROGRAMS.
INSTRUCTOR NOTES

THEME: MANAGING THE COMPLEXITIES ASSOCIATED WITH THE DEVELOPMENT OF SOFTWARE ARE KEY TO ALL PHASES OF THE LIFE CYCLE.

PURPOSE: TO PROVIDE A MIX OF TECHNIQUES FOR MANAGING COMPLEXITY IN PROGRAMMING.

REFERENCES: KNUTH, D. "FUNDAMENTAL ALGORITHMS" ADDISON WESLEY, MA; 1973

MCGOWAN, C., KELLY, J. "TOP-DOWN STRUCTURED PROGRAMMING TECHNIQUES" VAN NOSTRAND REINHOLD, NY; 1975
INSTRUCTOR NOTES

THERE IS NO CORRELATION BETWEEN THE LIMIT OF PERCEPTION CATEGORIES AND THE LIMIT OF DESCRIPTORS.
WHY CONSIDER PROGRAM COMPLEXITY MANAGEMENT?

- BECAUSE PEOPLE ARE LIMITED IN THE WAY THEY THINK, THEY MODEL ...  
  - DATA BY DESCRIPTORS AND THE RULES THAT ACT UPON IT  
  - ACTIVITIES BY CATEGORIZING DESCRIPTORS AND INTEGRATING THEM INTO RULES

- THE NUMBER OF PERCEPTION CATEGORIES HAS A LIMIT FROM AROUND 5 TO AROUND 9.

- THE NUMBER OF DESCRIPTORS THAT CAN BE PUT INTO A SINGLE CATEGORY ALSO HAS A LIMIT FROM AROUND 5 TO AROUND 9.

- THESE LIMITS GIVE HUMANS ONLY A SMALL TOOL KIT TO HANDLE COMPLEX PROBLEMS.
INSTRUCTOR NOTES

THESE TECHNIQUES ARE WAYS OF HELPING ORGANIZE COMPLEXITY.
PROGRAM COMPLEXITY MANAGEMENT TECHNIQUES

- TECHNIQUES HAVE BEEN DEVELOPED TO MODEL PORTIONS OF A SYSTEM IN WAYS THAT DO NOT TAX THE LIMITS OF HUMAN THINKING.

- SOME EARLY PROGRAMMING TECHNIQUES WERE ...
  - STEP-WISE REFINEMENT
  - DECISION TABLES
  - DECISION TREES
  - FINITE-STATE MAPS

- IN GENERAL THESE TECHNIQUES MAKE THE COMPLEXITY ASSOCIATED WITH A PORTION OF A PROGRAM MORE APPROACHABLE.
INSTRUCTOR NOTES

THIS HAS BEEN DEALT WITH BEFORE. BRIEFLY GO THROUGH THIS AREA.
STEP-WISE REFINEMENT

- **STEP-WISE REFINEMENT** IS THE BREAKING OF A PROBLEM INTO SEVERAL SMALLER PIECES:

![Diagram of step-wise refinement]

- THEN EACH PIECE CAN BE INVESTIGATED SEPARATELY, AND POTENTIALLY BROKEN INTO STILL SMALLER PIECES:

![Diagram of further refinement]

- THE NUMBER OF PIECES IN EACH BREAK UP HAS A LIMIT WITHIN THE BOUNDARIES OF HUMAN COMPLEXITY LIMITS.

- MOST OF THE ANALYSIS AND DESIGN METHODS UTILIZE STEP-WISE REFINEMENT
INSTRUCTOR NOTES

POINT OUT THAT THE "TOP" IS NOT NECESSARILY ALWAYS THE BEGINNING.

ALSO RECALL THE TECHNIQUE OF ITERATION.
STEP-WISE REFINEMENT

- PROGRAMMERS START AT THE "TOP" OF THE SYSTEM AND BREAK IT APART SLOWLY AS THEY FOLLOW THE STEP-WISE REFINEMENT PARADIGM.

- OFTEN, THE HARDEST TASK IN STEP-WISE REFINEMENT IS FINDING THE RIGHT "TOP."

- NO METHODOLOGY TELLS YOU WHAT THE "TOP" OF THE SYSTEM IS.

- SOMETIMES, THE ONLY WAY TO MAKE A "TOP" IS TO CREATE A SCENARIO (BLOW-BY-BLOW ACCOUNT) AND GENERALIZED TO A HIGHER LEVEL PATTERN.
INSTRUCTOR NOTES

GO OVER QUICKLY.
## Step-Wise Refinement Example

### Here's a Scenario for Adding Two Numbers:

<table>
<thead>
<tr>
<th>592</th>
<th>+236</th>
</tr>
</thead>
<tbody>
<tr>
<td>884</td>
<td></td>
</tr>
</tbody>
</table>

1. Look at rightmost column
2. Add 2 to 8 giving 10
3. Enter 0 in rightmost column of answer
4. Move to second column from right
5. Add 9 to 3 giving 12
6. Separate 12 into 1 and a carry of 1
7. Enter 1 in second column from right to answer
8. Move to third column from right
9. Add 5 to 2 and the carry of 1, giving 8
10. Enter 8 in third column from right of answer

### And Here's a More Generalized Procedure:

- Clear carry
- Set current-column to right-column
- Repeat until no digits in current-column
- Add carry and digits in column
- Split result into answer-digit and carry
- Enter answer-digit in current-column of answer
- Move current-column left by 1

**End Repeat**

**But watch out!! A single scenario may not give enough clues!**

This could have been expressed in a PDL.
INSTRUCTOR NOTES

AFTER READING THIS, ASK SOMEONE TO REPEAT IT WITHOUT LOOKING AT IT.
DECISION TABLES

- IN PROGRAMMING, DETAILS LEAVE US REELING IN CONFUSION, FOR EXAMPLE ...

WE LIMIT THE AMOUNT AN EMPLOYEE CAN SAVE, DEPENDING ON HIS SALARY AND HIS LENGTH OF SERVICE WITH US. AN EMPLOYEE CAN PUT AWAY UP TO 5%, 6% OR 7% OF THE FIRST $15,000 OF HIS SALARY IF HE'S BEEN WITH US 1 YEAR, 2 YEARS OR MORE, RESPECTIVELY. IF HE'S BEEN WITH US 1 YEAR, HE CAN PUT AWAY UP TO 4% OF THE NEXT $10,000. TWO-YEAR WORKERS GET 5% OF ANY AMOUNT FROM $15,000 TO $30,000 AND 4% AFTER THAT. LONG-SERVICE PEOPLE GET TO SAVE UP TO 7% OF THEIR FIRST $15,000 AND THEN 6% AND 5% AFTER THAT.

- HOW WELL DOES THIS TEXTUAL MODEL DESCRIBE THE RULES?
INSTRUCTOR NOTES

POINT OUT THE UNSPECIFIED PERCENTAGE ALLOWED.
DECISION TABLES

TRANSLATING THE TEXTUALLY EXPRESSED RULES TO TABULAR FORMAT PUTS RULES IN PERSPECTIVE

<table>
<thead>
<tr>
<th>LENGTH OF SERVICE (YEARS)</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1-2</th>
<th>1-2</th>
<th>1-2</th>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALARY (THOUSANDS)</td>
<td>&lt;15</td>
<td>15-25</td>
<td>&gt;25</td>
<td>&lt;15</td>
<td>15-30</td>
<td>&gt;30</td>
<td>&lt;15</td>
<td>15-30</td>
<td>&gt;30</td>
</tr>
<tr>
<td>PERCENTAGE ALLOWED %</td>
<td>5</td>
<td>4</td>
<td>?</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
INSTRUCTOR NOTES

DECISION TABLES ARE USEFUL FOR FINDING MISSING, CONFUSING, OR REDUNDANT ACTIONS.
DECISION TABLES

DECISION TABLES ARE USED TO IDENTIFY ACTIONS (RATHER THAN SIMPLY NUMBERS)
WHICH RESULT FROM A CERTAIN COMBINATION OF CONDITIONS.

IN GENERAL, DECISION TABLES ARE OF THE FORM:

| CONDITION 1 | T | T | T | T | F | F | F | F |
| CONDITION 2 | T | T | F | F | T | T | F | F |
| CONDITION 3 | T | F | T | F | T | F | T | F |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ACTION 1 | | | | | | | | |
| ACTION 2 | | | | | | | | |
| ACTION 3 | | | | | | | | |
| ACTION 4 | | | | | | | | |
| ACTION 5 | | | | | | | | |
| ACTION 6 | | | | | | | | |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

EACH POSSIBLE COMBINATION OF CONDITIONS GETS ITS OWN COLUMN IN THE TABLE.
The actions to be taken are marked in the columns.
DEcision tables example

A decision table can represent the rules for filling out the 1979 federal income tax guide prior to writing a program to automate the calculations:

- Textually statement - You must use schedule B if you had more than $400 in interest or dividends or you had a foreign account. If you had more than $400 in interest, fill out part I of schedule B. If you had more than $400 in dividends, fill out part II. If you filled out part I or part II or you had a foreign account, fill out part III.
INSTRUCTOR NOTES

SKIM THROUGH THE STEPS FOR THIS EXAMPLE. NO NEED TO ELABORATE MORE THAN WHAT THE SLIDE SAYS.
DECISION TABLES
EXAMPLE

STEP 1:
IDENTIFY ALL THE CONDITIONS IN THE PROBLEM AND ALL THE POSSIBLE VALUES OF EACH CONDITION. (DON'T WORRY ABOUT THE ACTIONS RIGHT NOW.)

<table>
<thead>
<tr>
<th>CONDITIONS</th>
<th>VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEREST &gt; $400</td>
<td>Y OR N</td>
</tr>
<tr>
<td>DIVIDENDS &gt; $400</td>
<td>Y OR N</td>
</tr>
<tr>
<td>FOREIGN ACCOUNT</td>
<td>Y OR N</td>
</tr>
</tbody>
</table>

STEP 2:
WORK OUT HOW MANY COMBINATIONS OF CONDITIONS THERE ARE IN TOTAL BY MULTIPLYING THE NUMBER OF VALUES FOR EACH CONDITION.

IN THIS CASE, THERE ARE A TOTAL OF 8 COMBINATIONS OF CONDITIONS ($8 = 2 \times 2 \times 2$).
INSTRUCTOR NOTES

THESE STEPS ARE, IN REALITY, DONE VERY QUICKLY. HOWEVER, IF THERE ARE MANY DECISIONS, THIS TECHNIQUE CAN BE LABORIOUS.
**DECISION TABLES**

**EXAMPLE**

STEP 3:

**WRITE DOWN ALL OF THE INDEPENDENT ACTIONS.**

**ACTIONS**

DO NOTHING  
FILL OUT PART I  
FILL OUT PART II  
FILL OUT PART III

STEP 4:

**DRAW A SKELETON DECISION TABLE WITH THE CONDITIONS AT THE TOP AND THE ACTIONS AT THE BOTTOM. DRAW AS MANY COLUMNS AS THERE ARE COMBINATIONS OF CONDITIONS. NUMBER THE COLUMNS.**

<table>
<thead>
<tr>
<th>CONDITIONS</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEREST &gt; $400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVIDENDS &gt; $400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOREIGN ACCOUNT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACTIONS</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DO NOTHING</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILL OUT PART I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILL OUT PART II</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILL OUT PART III</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DECISION TABLES

STEP 5:

FILL IN THE CONDITION VALUES:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEREST &gt; $400</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>DIVIDENDS &gt; $400</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>FOREIGN ACCOUNT</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

DO NOTHING
FILL OUT PART I
FILL OUT PART II
FILL OUT PART III

STEP 6:

MARK APPROPRIATE ACTION(S) FOR EACH COMBINATION OF CONDITIONS.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEREST &gt; $400</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>DIVIDENDS &gt; $400</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>FOREIGN ACCOUNT</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

DO NOTHING
FILL OUT PART I
FILL OUT PART II
FILL OUT PART III

STEP 7: IMPLEMENT IN Ada

VG 778.1

26-12
INSTRUCTOR NOTES

ASK CLASS, "WHERE ARE TABLES NOT USEFUL?"
DECISION TABLES

- TO SUM UP, A DECISION TABLE
  - IS A USEFUL TOOL WHERE COMPLICATED CONDITIONS APPLY OR WHERE COMPLEX DECISIONS ARE TO BE MADE
  - IS USED TO FIND INCONSISTENCIES OR MISSING ACTIONS
  - IS USED TO ORGANIZE ONE'S THOUGHTS PRIOR TO WRITING CODE
  - CAN BE USED AS DOCUMENTATION WITHIN THE CODE

- DECISION TABLES ARE TYPICALLY IMPLEMENTED WITH SOME COMBINATION OF THE STRUCTURED CONTROL CONSTRUCTS IN PREVIOUS SECTION.
DECISION TREES

- A DECISION TREE IS A MORE GRAPHIC FORM OF A DECISION TABLE

- FOR EXAMPLE, THE TABLE OF EMPLOYEE SAVINGS RATES ...

<table>
<thead>
<tr>
<th>LENGTH OF SERVICE (YEARS)</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1-2</th>
<th>1-2</th>
<th>1-2</th>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALARY (THOUSANDS) &lt;15</td>
<td>15-25</td>
<td>&gt;25</td>
<td>&lt;15</td>
<td>15-30</td>
<td>&gt;30</td>
<td>&lt;15</td>
<td>15-30</td>
<td>&gt;30</td>
<td></td>
</tr>
<tr>
<td>PERCENTAGE ALLOWED % 5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

- CAN BE REPRESENTED IN TREE FORM BY ...

```
SAVINGS PLAN
    ├──<1 YEAR
    │   ├──<15,000
    │   │   └──5%
    │   │   └──4%
    │   └──15,000
    │       └──?
    │   └──25,000
    │       └──5%
    │   └──30,000
    │       └──5%
    │   └──>$30,000
    │       └──?
    └──1 YEAR - 2 YEARS
        └──<15,000
            └──7%
        └──15,000
            └──6%
        └──30,000
            └──6%
        └──>$30,000
            └──5%
    └──>2 YEARS
```

VG 778.1
DECISION TREES

- MANY PEOPLE FIND DECISION TREES TO BE MORE IMMEDIATE THAN DECISION TABLES
  - EXPLICITLY SHOW RELATIONSHIP BETWEEN CAUSE AND EFFECT (CONDITION AND ACTION)

- SOMETIMES, TREES ARE EASIER TO IMPLEMENT THAN TABLES.
  SO PICK A REPRESENTATION (TREE OR TABLE) MOST SUITABLE FOR PROGRAMMING. (YOU CAN ALWAYS DOCUMENT IT IN THE OTHER FORM IF YOU WISH.)
INSTRUCTOR NOTES

A BLACK BOX IS A MACHINE WHOSE ACTIONS ARE HIDDEN FROM THE OBSERVER. ONLY BY OBSERVING THE BOX'S RESPONSE TO INPUTS CAN ONE MODEL ITS BEHAVIOR. A SCENARIO IS THE OBSERVING OF OUTPUTS BASED ON GIVEN INPUTS.

STATE THAT FOR THIS EXAMPLE WE AREN'T USING A TRUE BLACK BOX. WE NEED TO HAVE SOME INFORMATION ON HOW OUR TELEPHONE SYSTEM WORKS FOR THIS EXAMPLE TO END UP BEING RIGHT. FURTHER, THESE "INPUTS" ARE REALLY ACTIONS, NOT CONTROL VARIABLES. THE "WAIT" INPUT IS REALLY A COMMUNICATIONS SWITCHING WAIT. PICKING UP THE RECEIVER GIVES AN INSTANT DIAL TONE WITH NO WAIT.
FINITE STATE MAPS

FINITE STATE MAPS MODEL THE DYNAMIC BEHAVIOR OF A PORTION OF A SYSTEM, BY SHOWING ITS STATES AND TRANSFORMATIONS.

MANY REAL TIME OR EMBEDDED APPLICATIONS HAVE A PORTION THAT CAN EASILY BE REPRESENTED AS A FSM (I.E. COMMUNICATIONS PROTOCOLS).

FINITE STATE MAPS DESCRIBE "BLACK BOXES." FOR EXAMPLE, HERE'S A BLACK BOX SCENARIO OF HOW TO USE THE TELEPHONE TO TALK WITH THE OPERATOR ...

POSSIBLE CONDITIONS
- RECEIVER OFF/ON
  CRADLE
- DIALING

USER STIMULUS
- PICK UP RECEIVER
- DIAL "0"
- WAIT
- HANG UP RECEIVER

TELEPHONE STATE
- HUNG UP
- DIAL TONE
- CS WAIT
- TALK

TELEPHONE RESPONSE
- DIAL TONE
- CLICK
- OPERATOR'S VOICE
INSTRUCTOR NOTES

THERE ARE THREE INPUTS, EACH WITH A BINARY STATE, OR \(2^3\) POSSIBLE COMBINATIONS. A STATE EXISTS IF SOMETHING HAPPENS AT AN OUTPUT GIVEN AN INPUT. 0 MEANS THAT CONDITION IS NOT TRUE.

<table>
<thead>
<tr>
<th>INPUT</th>
<th>pick up</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;0&quot;</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CS wait</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>possible state</th>
<th>*</th>
<th>*</th>
<th>*</th>
<th>**</th>
<th>*</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>OUTPUT</th>
<th>dial tone</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>click</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>op voice</td>
<td>0*</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

* This is a state because by definition we need a "zeroth" state.

**In real world, but not this problem, this would be another state (i.e., a wait state).
FINITE STATE MAPS
EXAMPLE

A TABLE IS DRAWN, DEFINING EACH STATE OF THE PHONE USING A SPECIFIC COMBINATION OF INPUTS AND OUTPUTS...

<table>
<thead>
<tr>
<th>PROPERTY</th>
<th>CONDITIONS</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RECEIVER OFF/ON CRADLE</td>
<td>DIALING</td>
</tr>
<tr>
<td>0. HUNG UP</td>
<td>ON</td>
<td>NO</td>
</tr>
<tr>
<td>1. DIAL TONE</td>
<td>OFF</td>
<td>NO</td>
</tr>
<tr>
<td>2. CS WAIT</td>
<td>OFF</td>
<td>DIAL &quot;0&quot;</td>
</tr>
<tr>
<td>3. TALK</td>
<td>OFF</td>
<td>NO</td>
</tr>
</tbody>
</table>

LASTLY, A MAP IS DRAWN, USING THE SYMBOLS ...

STATE

TRANSITION (AN ACTION THAT GETS YOU OUT OF ONE STATE AND INTO ANOTHER)
INSTRUCTOR NOTES

THUS WE HAVE FOUR STATES. WE NEED TO DRAW THE TRANSITIONS BETWEEN THEM. THERE ARE 12 POSSIBLE TRANSITIONS (6 FORWARD, 6 BACKWARDS, i.e.

```
0 --- 1 --- 2 --- 3 and vice versa)
```

NOW DRAW THE FS MAP AS FOLLOWS, DESCRIBING WHY ONLY CERTAIN TRANSITIONS ARE ALLOWED.
INSTRUCTOR NOTES

FSM's represent the dynamic nature of a system, while decision trees and tables are more static in nature.
FINITE STATE MAPS

- FINITE STATE MAPS ARE EASY FOR PEOPLE TO READ AND UNDERSTAND

- DECISION TABLES AND FINITE STATE MAPS SHARE SOME COMMON ATTRIBUTES
  - BOTH EXPRESS CONDITIONS VS. ACTIONS
  - A TABULAR REPRESENTATION OF A FSM CAN BE TRANSLATED TO A
    DECISION TABLE TO AID IN PROGRAMMING
SUMMARY

- PROGRAMS ARE COMPLEX.

- HUMANS HAVE LIMITS TO THEIR THINKING.

- TECHNIQUES WERE DEVELOPED TO COPE WITH COMPLEXITY BY LIVING WITHIN HUMAN LIMITS:
  - STEP-WISE REFINEMENT
  - DECISION TABLES
  - DECISION TREES
  - FINITE STATE MAPS
INSTRUCTOR NOTES

THEME: INTEREST IN "PROVING" PROGRAMS ARE CORRECT IS GROWING, THE TECHNOLOGY ISN'T HERE YET, BUT IT COULD BE HERE SOON.

PURPOSE: TO PROVIDE A PRACTICAL APPROACH TO APPLYING PROGRAM CORRECTNESS TECHNIQUES TO Ada PROGRAMS.

REFERENCES: DJIKSTRA, E. "GUARDED COMMANDS, NONDETERMINACY AND FORMAL DERIVATION OF PROGRAMS" CACM VOL. 18, NO. 5; AUGUST 1975

MILLS, H. "HOW TO WRITE CORRECT PROGRAMS AND KNOW IT" CONFERENCE ON RELIABLE SOFTWARE PROCEEDINGS ACM SIGPLAN; 1975

WIRTH, N. "ON THE COMPOSITION OF WELL-STRUCTURED PROGRAMS" ACM COMPUTING SURVEYS VOL. 6, NO. 4; DECEMBER 1974
PROGRAM CORRECTNESS

- **CORRECTNESS** - MEANS THE PROGRAM WILL WORK FOR ANY COMBINATION OF INPUTS

  - ANY MEANS ALL. FOR EXAMPLE, TO VERIFY THE ADDITION OF TWO INTEGER NUMBERS FROM 1 TO 1,000,000 WOULD TAKE 1,000,000,000,000 TESTS!

- TESTING FOR CORRECTNESS IS NOT FEASIBLE

  - COMBINATIONS OF TESTS EXPAND EXPONENTIALLY
  
  - TESTING ONLY SHOWS THE **PRESENCE** OF ERRORS, NOT THEIR **ABSENCE**

- CORRECTNESS IS A MEASURE OF THE **ABSENCE** OF ERRORS

- A WEALTH OF THEORY EXISTS THAT IS JUST NOW SHOWING LIMITED PRACTICAL USE
INSTRUCTOR NOTES

DON'T CONFUSE CORRECTNESS WITH OPERABILITY. A PROGRAM MAY BE CORRECT, BUT STILL NOT USABLE BY THE INTENDED USER.
PROGRAM CORRECTNESS MOTIVATION

- IT IS EASIER TO DEBUG WHEN THERE ARE FEWER ERRORS ...

- RELIABILITY REQUIREMENTS ARE GETTING STRICTER ...

VG 778.1

27-2
INSTRUCTOR NOTES

A MAJOR REASON FOR USING CORRECTNESS ANALYSIS IS THAT IT IS GETTING CHEAPER TO BUILD "CORRECT" PROGRAMS THAN TO FIX INCORRECT ONES.

HEURISTIC = RULE OF THUMB.
BUILDING IN CORRECTNESS

- IT IS EASIER TO BUILD CORRECTNESS IN THAN TO ...
  - TEST IT IN
  - GET THE DEFECT OUT

- THERE ARE SOME SIMPLE HEURISTICS ...
  - ASSERTIONS -- ONE OF THE FIRST NOTED BENEFITS OF COMMENTING CODE WAS OPPORTUNITY TO DOCUMENT ASSERTIONS ABOUT PROGRAMS
  - INVARIANT -- IS THE SITUATION THAT IS ALWAYS TRUE ABOUT THE LOOP; BEFORE AND AFTER
  - GUARDS -- A GUARD ENSURES THE SAFETY OF CARRYING OUT EACH STEP OF A LOOP
INSTRUCTOR NOTES

AN ASSERTION IS SOME RELATIONSHIP THAT THE PROGRAMMER KNOWS TO BE TRUE.

POINT OUT THAT THE EXAMPLE IS TRIVIAL, BUT DOES SHOW WHAT IS MEANT BY AN ASSERTION.
ASSERTIONS

• ASSERTION - IS A STATEMENT (COMMENT) ABOUT WHAT HAS HAPPENED SO FAR.
  (AS OPPOSED TO WHAT IS ABOUT TO HAPPEN.)

  B = 5;
  A = B + 7;

  -- {A IS NOW 12 AND B IS NOW 5.}

• ASSERTIONS CAN BE ENGLISH STATEMENTS ABOUT THE RESULT OF AN EVALUATION OF A CONDITIONAL.
ASSERTIONS

• ASSERTIONS ARE THE "STORY SO FAR."

• THE ONE-IN, ONE-OUT CONTROL STRUCTURES OF STRUCTURED PROGRAMMING ENABLES US TO MAKE GUARANTEED ASSERTIONS

• ASSERTIONS HELP CLARIFY THE ELSE PARTS OF CONDITIONALS ESPECIALLY NESTED ONES.

• ASSERTIONS WHEN USED WITH VERIFIER TOOLS PROVIDE SOME OPPORTUNITIES TO PROVE PROGRAM CORRECTNESS
INSTRUCTOR NOTES

THE LOOP INVARIANT IS A TECHNIQUE THAT WILL HELP US CODE LOOPS CORRECTLY.
INERSIONS

- Perhaps the toughest control structure to get right is the loop. Problems include ...
  - Code looping indefinitely
  - Code does one too many or too few iterations

- All these problems are the result of improperly understanding the loop invariant

- From an invariants standpoint a loop is a step-by-step procedure to go from a beginning to a conclusion
INSTRUCTOR NOTES

THE INVARIANT NEVER CHANGES. "SUM" IS ALWAYS EQUAL TO THE SUMMATION OF THE Jth ELEMENT IN THE ARRAY "A."

- SUM IS SUMMATION OF 1st j ELEMENTS OF THE LOOP.

- AT THE BOTTOM OF THE LOOP J > 10, IN FACT J IS 11.

- WHAT'S TRUE AT THE BOTTOM OF THE LOOP IS THE NEGATION OF THE CONDITION AND INVARIANT.
INVARINTS

FOR EXAMPLE, THE LOOP:

\begin{verbatim}
J:= 0;
SUM:= 0;
WHILE J < 10 LOOP
  J:= J + 1;
  SUM = SUM + A(J);
END LOOP;
\end{verbatim}

HAS THE INVARIANT:

\begin{align*}
SJM &= \sum_{i=1}^{J} A(i) \\
\text{INITIALIZATION SETS UP THE INVARIANT:} \\
\text{WHEN J = 0, SUM = 0}
\end{align*}
INSTRUCTOR NOTES

EVERY POINT MUST BE ABLE TO BE VERIFIED.
I', ARIANTS

START
CONDITION FOR STOPPING
STEP
GOAL

IS THE RESULT OF INITIALIZING THE LOOP INVARIANT AND STEPPING ENOUGH TIMES TO SATISFY THE STOPPING CONDITION

ASSERTIONS ABOUT

VGI 778.1
A GUARD "ALLOWS" US TO CONTINUE.
GUARDS

- EACH STEP OF A LOOP HAS TWO PARTS ...
  - ONE THAT MAKES PROGRESS TOWARDS THE GOAL
  - ONE THAT PRESERVES THE LOOP INARIANT

- A GUARD IS THE OPPOSITE OF THE CONDITION FOR STOPPING
INSTRUCTOR NOTES

DON'T GO THROUGH THIS IN MUCH DEPTH. IT IS STRAIGHTFORWARD AND IT WILL BE INVESTIGATED IN DETAIL IN THE FOLLOWING SLIDE.
PROGRAM CORRECTNESS EXAMPLE

- CODE FRAGMENT MANIPULATING A STACK ...

Make_Empty (Stack);
Count := 0;
Push (Tree, Onto => Stack);
while not Is_Empty (Stack) loop

Pop (T, Off_Of => Stack);
if Is_Leaf (T) then
  Count := Count + 1;
else

  Push (Left_Subtree (T), Onto => Stack);
  Push (Right_Subtree (T), Onto => Stack);
end if;

end loop;
INSTRUCTOR NOTES

THE ASSERTIONS ARE IN THE BOXES. EACH ASSERTION TELLS WHAT HAS HAPPENED (OR SHOULD HAVE HAPPENED) SO FAR.
PROGRAM CORRECTNESS - AN OPTIMIST'S VIEW

• IN THE FUTURE, SPECIFICATIONS WILL BE COMPLETELY FORMAL AND RIGOROUS

[REQUIREMENTS DEFINITION] WILL CULminate IN A COMPLETE, UNABMIGUOUS, AND PROVABLY CONSISTENT SPECIFICATION FOR THE ENTIRE END PRODUCT ...

INSTEAD OF PRETTY PICTURES AND DRAWINGS, [THE SPECIFICATION WILL CONSIST OF] A COLLECTION OF DEFINITIONS, MATHEMATICAL FORMULAS, AND LOGICAL PROOFS ...

C.A.R. HOARE
"PROGRAMMING: SORCERY OR SCIENCE?"
IEEE SOFTWARE, APRIL 1984

• PROGRAMMERS WELL-VERSED IN LOGIC WILL DEVELOP A PROGRAM AND A PROOF OF CORRECTNESS HAND-IN-HAND

• PROVEN PROGRAMS ARE GUARANTEED TO MEET THEIR SPECIFICATIONS, SO IT WILL BE UNNECESSARY TO PROVIDE FOR SOFTWARE ERROR RECOVERY
INSTRUCTOR NOTES

ON THE OTHER HAND, OTHERS BELIEVE CORRECTNESS ISN'T WORTH THE TIME, TROUBLE, OR EXPENSE, AND ISN'T COMPLETE ANYWAY.
PROGRAM CORRECTNESS - A PESSIMIST'S VIEW

- VERIFIABILITY IS NOT A PRIMARY GOAL OF MOST PROGRAMMING LANGUAGES, INCLUDING Ada
  - HARD TO WRITE PROOF RULES FOR PARAMETER PASSING, SHARED VARIABLES, ETC.
  - DEFINITION OF Ada IS LARGE AND COMPLEX

- MACHINE ARITHMETIC DOESN'T OBEY ARITHMETIC AXIOMS:
  EXAMPLES:
  \[ a*(b+c) = a*b + a*c \]
  \[ (a/b)*b = a \]

- EMBEDDED SYSTEMS ARE TOO LARGE AND COMPLEX TO PROVE CORRECT

- EVEN FOR NONEMBEDDED SYSTEMS, INPUT-OUTPUT ASSERTIONS ARE EXTREMELY COMPLEX AND DIFFICULT EVEN FOR A TRAINED LOGICIAN TO GET RIGHT
  - COMPLETENESS
  - NOTATION
INSTRUCTOR NOTES

MOREOVER, THE COMPUTER FIELD ISN'T ONLY POPULATED WITH C.A.R. HOARE'S.
PROGRAM CORRECTNESS - A PESSIONIST'S VIEW (CONTINUED)

- THERE EXIST CORRECT PROGRAMS THAT CAN'T BE PROVEN CORRECT

- VERIFIED SOFTWARE CAN FAIL
  - COMPILER, OPERATING SYSTEM, HARDWARE ERRORS
  - INPUT DATA VIOLATING INPUT ASSUMPTIONS
  - NONFUNCTIONAL REQUIREMENTS (PERFORMANCE)

- FEW PEOPLE WITHOUT GRADUATE DEGREES IN COMPUTER SCIENCE WILL BE CAPABLE OF USING
  FORMAL VERIFICATION
  - FACILITY IN READING, WRITING, AND MANIPULATING LOGICAL FORMULAS
  - DIFFICULTY OF FORMULATING INVARIANTS

- DEVELOPING A PROOF WHILE WRITING A PROGRAM ENTAILS VOLUMINOUS, ERROR-PRONE FORMULA
  MANIPULATION
INSTRUCTOR NOTES

THESE CONCLUSIONS ARE ABOUT THE "BEST" COMPROMISES BETWEEN THE OPTIMIST'S AND PESSIMIST'S VIEWS.
PROGRAM CORRECTNESS - A PRAGMATIST'S VIEW

- IDENTIFY A VERIFIABLE SUBSET OF Ada.
  - AVOID DIFFICULT LANGUAGE FEATURES, ENCOURAGE GOOD STYLE

- ACCEPT INCOMPLETENESS
  - MOST "ACCIDENTALLY CORRECT" PROGRAMS CAN'T BE PROVEN CORRECT
  - MOST POORLY WRITTEN PROGRAMS CAN'T BE PROVEN CORRECT
  - PROGRAMS USING CERTAIN LANGUAGE FEATURES (E.G. SHARED VARIABLES) CAN'T BE PROVEN CORRECT

- PROVE SPECIFIC, LIMITED PROPERTIES OF PROGRAMS
  - NO NEED TO FORMULATE COMPLEX INPUT/OUTPUT ASSERTIONS FOR A FULL PROGRAM
  - SOME PRECONDITIONS/POSTCONDITIONS CAN BE FORMULATED AUTOMATICALLY
  - VERIFY PROGRAM COMPONENTS INDEPENDENTLY AND COMPOSE PROOFS TO VERIFY LARGE SYSTEMS
Ada AND PROGRAM CORRECTNESS

- PRINCIPAL GOALS OF Ada INCLUDE:
  - CONSTRUCTION OF RELIABLE PROGRAMS
  - EASE OF MAINTENANCE

- THESE GOALS CAN BE ADVANCED BY FORMALLY PROVING THAT AN Ada PROGRAM WILL BEHAVE IN A CERTAIN WAY
  - CORRECT IMPLEMENTATION
  - MODIFICATIONS THAT DON'T VIOLATE INITIAL ASSUMPTIONS
INSTRUCTOR NOTES

THESE ARE THE THINGS WE CAN PROVE ABOUT Ada PROGRAMS. EACH WILL BE REVIEWED IN DETAIL.
SPECIFIC PROVABLE PROPERTIES OF Ada PROGRAMS

- ABSENCE OF CERTAIN FORMS OF ERRONEOUSNESS
- ABSENCE OF UNANTICIPATED EXCEPTIONS
- CORRECTNESS OF NARROWLY-DEFINED PROGRAM COMPONENTS
- CORRECTNESS OF DATA ABSTRACTIONS PROVIDED BY PACKAGES
- NUMERIC PROPERTIES
FOR AN Ada ORIENTED CLASS, SPEND SOME TIME AND GIVE AN EXAMPLE. OTHERWISE, SKIM THROUGH THESE POINTS.
PROVING THE ABSENCE OF CERTAIN FORMS OF ERRONEOUSNESS

- STRAIGHTFORWARD FOR PROGRAMS THAT FOLLOW CERTAIN REASONABLE PROGRAMMING PRACTICES:
  - NO PATHS ALONG WHICH A SCALAR VARIABLE IS EXAMINED BEFORE IT IS GIVEN A VALUE
  - RESTRICTED USE OF GLOBAL VARIABLES
  - RESTRICTED USE OF EXPRESSIONS WITH SIDE-EFFECTS
  - NO USE OF THE SAME RECORD, OR SLICES OF THE SAME ARRAY, AT MORE THAN ONE ACTUAL PARAMETER POSITION IN THE SAME SUBPROGRAM CALL
  - NO DIRECT USE OF THE SUPPRESS PRAGMA, UNCHECKED CONVERSION, UNCHECKED DEALLOCATION, OR SHARED VARIABLES

- COMPONENTS THAT DEPART FROM THESE PRACTICES CAN STILL BE PROVABLY NONERRONEOUS

- BENEFITS OF PROVING THE ABSENCE OF ERRONEOUSNESS:
  - FILLS IN THE GAPS IN COMPILe-TIME CHECKING
  - MAKE PROGRAMS SAFER TO RUN AND SAFER TO PORT
  - PRECONDITIONS AND POSTCONDITIONS CAN BE GENERATED AUTOMATICALLY
INSTRUCTOR NOTES

IF ONE CAN DO ALL ERRONEOUS PROOFS, APPLY SOME TO THESE AREAS.
MOST PROMISING AREAS FOR NON-ERRORONEOUSNESS PROOFS

- EVALUATION OF A SCALAR VARIABLE WITH AN UNDEFINED VALUE
  - VERIFICATION IS MORE POWERFUL THAN STRAIGHTFORWARD DATA FLOW (LIVE/DEAD, DEF/REF) ANALYSIS, BECAUSE IT CONSIDERS THE CONDITIONS UNDER WHICH A POINT IN A PROGRAM IS REACHED

- OCCURRENCE OF AN ERROR SITUATION FOR WHICH RUN-TIME CHECKS HAVE BEEN SUPPRESSED
  - SUPPRESSION OF CHECKS CAN BE ONE OF THE MOST EFFECTIVE WAYS TO SPEED UP A CRITICAL PART OF A PROGRAM, BUT IT IS ALSO ONE OF THE MOST DANGEROUS
  - IF AN EFFECTIVE VERIFIER IS AVAILABLE, MANAGERS CAN FORBID THE USE OF THE SUPPRESS PRAGMA WITHOUT PROOF OF ITS SAFETY

- USE OF AN ALLOCATED VARIABLE AFTER IT HAS BEEN DEALLOCATED
  - VERIFICATION VALUABLE FOR VALIDATING LEGITIMATE USES OF UNCHECKED DEALLOCATION
FORMS OF ERRONEOUSNESS THAT PRESENT VERIFICATION PROBLEMS

- AN UNCHECKED CONVERSION THAT VIOLATES THE PROPERTIES THAT ARE GUARANTEED BY THE LANGUAGE FOR OBJECTS OF THE RESULT TYPE
  - ILL-DEFINED

- UPDATE OF A SHARED VARIABLE BETWEEN SYNCHRONIZATION POINTS IF ANOTHER TASK UPDATES OR EXAMINES THE VARIABLE BETWEEN THE SAME SYNCHRONIZATION POINTS
  - IT MAY BE DIFFICULT TO RELATE DYNAMIC SYNCHRONIZATION POINTS BETWEEN TWO TASKS TO POINTS IN THE PROGRAM TEXT. (COMPLICATIONS INCLUDE ENTRY FAMILIES; SELECT STATEMENTS; AND ENTRY CALLS AND ACCEPT STATEMENTS WITHIN CONDITIONAL STATEMENTS.)
  - IMPOSSIBILITY OF ERRONEOUSNESS MAY BE DUE TO IMPLEMENTATION-DEPENDENT SCHEDULING CHARACTERISTICS
  - VERIFICATION OF CONCURRENT PROGRAMS USING SHARED VARIABLES IS NOT WELL UNDERSTOOD

- ADDRESS CLAUSES THAT CAUSE OBJECTS OR PROGRAM UNITS TO BE OVERLAID, OR THAT CAUSE THE SAME INTERRUPT TO BE LINKED TO MORE THAN ONE ENTRY
  - IMPLEMENTATION-DEPENDENT
OTHER FORMS OF ERRONEOUSNESS WHOSE ABSENCE CAN BE PROVEN

- Invocation of a subprogram in a situation where its effect depends on whether composite parameters are passed by copying or by reference.

- Assignment to a component in a record variant, if execution of the assignment statement has the side-effect of changing the record's discriminant.

- Passing a component in a record variant as a composite or in out or out parameter to a subprogram, if execution of the subprogram changes the record's discrimination.
VERIFYING THE ABSENCE OF UNANTICIPATED EXCEPTIONS

- PROGRAMMER SPECIFIES THE EXCEPTIONS THAT HE ANTICIPATES EACH SUBPROGRAM MAY RAISE

- VERIFIER PROVES THAT NO OTHER EXCEPTIONS CAN BE RAISED

- BENEFITS:
  - PRECONDITIONS AND POSTCONDITIONS CAN BE GENERATED AUTOMATICALLY
  - PROGRAMMER ALERTED TO BOUNDARY CONDITIONS AND OTHER POSSIBLE EXCEPTIONS HE HASN'T ACCOUNTED FOR

- DIFFICULTIES:
  - IMPLEMENTATION-DEPENDENT EXCEPTIONS (IO_Exceptions.Use_Error, Program_Error)
  - "UNPREDICTABLE" EXCEPTIONS (Storage_Error, I/O exceptions dependent on existence and content of external files, IO_Exceptions.Device_Error)
VERIFYING DATA ABSTRACTIONS ARE NOT A DIFFICULT THING TO DO, AND IS HIGHLY BENEFICIAL.
VERIFYING DATA ABSTRACTIONS

- IN Ada, a DATA ABSTRACTION USUALLY CORRESPONDS TO A PACKAGE PROVIDING A PRIVATE TYPE AND OPERATIONS ON THAT TYPE

- BEHAVIOR OF THE TYPE CAN BE DESCRIBED BY AXIOMS, AND THE AXIOMS CAN BE USED TO VERIFY PARTS OF THE PROGRAM THAT USE THE DATA ABSTRACTION

- VERIFYING THE DATA ABSTRACTION CONSISTS OF PROVING THAT THE PACKAGE'S SUBPROGRAMS OBEY THE AXIOMS

- BENEFITS:
  - WELL UNDERSTOOD
  - SUPPORTED BY Ada PACKAGES AND VISIBILITY RULES
  - ALLOWS COMPOSITION OF PROOFS
  - FACILITATES A LIBRARY OF TESTED, FORMALLY SPECIFIED DATA ABSTRACTIONS
INSTRUCTOR NOTES

THIS IS PROBABLY NOT WORTH DOING IN MOST CASES.
VERIFYING NUMERIC PROPERTIES

- BASE AXIOMS FOR REAL ARITHMETIC ON MODEL INTERVALS

- PROVE ASSERTIONS ABOUT ACCURACY OF RESULTS

- BENEFITS:
  - FORMALIZE PROOF OF NUMERIC PROPERTIES
  - PROVE PORTABILITY
  - PROVE ABSENCE OF Numeric_Error

- PROBLEMS:
  - MODEL ARITHMETIC TENDS TO BE FAR MORE PESSIMISTIC THAN REALITY, BUT
  Ada SEMANTICS DO NOT JUSTIFY ANY STRONGER METHOD
  - MAY BE IMPOSSIBLE TO PROVE CORRECTNESS OF CORRECT ALGORITHMS
SUMMARY

- PROOF OF TOTAL CORRECTNESS INFEASIBLE, ESPECIALLY FOR
  - A LANGUAGE WITH ALL OF Ada'S FEATURES
  - PROBLEMS AS COMPLEX AS EMBEDDED SYSTEMS SOFTWARE
  - A WORKFORCE WITH LIMITED EXPERTISE IN FORMAL LOGIC

- SIGNIFICANT BENEFITS CAN BE DERIVED FROM APPLYING VERIFICATION TECHNIQUES
  TO MORE LIMITED PROBLEMS. BENEFITS INCLUDE:
  - PROGRAMMING DISCIPLINE
  - FORMAL SEMANTIC INTERFACES
  - RELIABLE PROGRAMS
  - RELIABLE MAINTENANCE
  - RELIABLE SOFTWARE REUSE

- THESE MORE LIMITED PROBLEMS SEEM FEASIBLE
  - IMPACT OF Ada'S COMPLEXITY IS CONTROLLED
  - DEMANDS ON USER EXPERTISE ARE REDUCED

VG 778.1   27-25
INSTRUCTOR NOTES

THEME: SOFTWARE TESTING IS MORE THAN DEBUGGING CODE.

PURPOSE: TO PROVIDE AN OVERVIEW OF THE MAJOR STRATEGIES INVOLVED IN SOFTWARE TESTING.

REFERENCE: MILLER, E., HOWDEN, W. "TUTORIAL: SOFTWARE TESTING AND VALIDATION TECHNIQUES" IEEE, EHO 138-8; 1978
INSTRUCTOR NOTES

- TESTING IS IN THE PROCESS OF EXECUTING A PROGRAM WITH THE INTENT OF FINDING ERRORS.

- DEBUGGING IS WHAT YOU DO AFTER YOU HAVE DISCOVERED AN ERROR -- IDENTIFYING THE EXACT CAUSE OF IT, WITH THE OPTION TO CORRECT IT.

- THE GOALS OF TESTING ARE TO FIND ERRORS - NOT TO NOT FIND ERRORS, AND TO INCREASE OUR CONFIDENCE LEVEL THAT OUR PROGRAM WILL PERFORM CORRECTLY.
TESTING

- Often testing and debugging are confused
  - Testing -- showing that no errors are present.
  - Debugging -- is identifying and correcting an error.

- Testing uncovers many kinds of errors ...
  - Logic
  - Overload
  - Timing
  - Throughput
  - Capacity
  - Recovery
  - System software
      - Mistakes when coding from the design
      - When data structures are filled to capacity
      - Coordination among parallel processes
      - Processing speed
      - Memory limits
      - What happens when the system fails
      - Assumptions about the software
INSTRUCTOR NOTES

NOTE HOW WE MOVE FROM FORMAL (CORRECTNESS) TO INFORMAL (REVIEWS) TECHNIQUES.
RELATIONSHIP OF TESTING AND OTHER ERROR REMOVAL TECHNIQUES

<table>
<thead>
<tr>
<th>TECHNIQUE</th>
<th>ANALYSIS</th>
<th>DESIGN</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM CORRECTNESS</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>TESTING</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- STRUCTURAL</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>- FUNCTIONAL</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>REVIEWS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- REQUIREMENTS</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- DESIGN</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>- CODE</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

X - INDICATED TECHNIQUE IS EFFECTIVE IN ERROR REMOVAL
**RELATIONSHIP OF TESTING AND OTHER ERROR REMOVAL TECHNIQUES**

**EFFECTIVENESS OF TECHNIQUES**

<table>
<thead>
<tr>
<th>IMPLEMENTATION ERROR TYPES</th>
<th>CODE REVIEWS</th>
<th>PROGRAM CORRECTNESS</th>
<th>TESTING STRUCTURAL</th>
<th>TESTING FUNCTIONAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPUTATIONAL</td>
<td>MEDIUM</td>
<td>HIGH</td>
<td>HIGH</td>
<td>MEDIUM</td>
</tr>
<tr>
<td>LOGIC</td>
<td>MEDIUM</td>
<td>HIGH</td>
<td>HIGH</td>
<td>MEDIUM</td>
</tr>
<tr>
<td>INPUT AND OUTPUT</td>
<td>HIGH</td>
<td>LOW</td>
<td>MEDIUM</td>
<td>HIGH</td>
</tr>
<tr>
<td>DATA HANDLING</td>
<td>HIGH</td>
<td>MEDIUM</td>
<td>LOW</td>
<td>HIGH</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>HIGH</td>
<td>LOW</td>
<td>HIGH</td>
<td>MEDIUM</td>
</tr>
<tr>
<td>DATA DEFINITION</td>
<td>MEDIUM</td>
<td>MEDIUM</td>
<td>LOW</td>
<td>MEDIUM</td>
</tr>
<tr>
<td>DATABASE</td>
<td>HIGH</td>
<td>LOW</td>
<td>MEDIUM</td>
<td>MEDIUM</td>
</tr>
</tbody>
</table>
INSTRUCTOR NOTES

IF THE TEST RESULT IS CONSISTENT WITH THE EXPECTED RESULT, THE COMPONENT IS DEEMED CORRECT IN THE LIMITED CONTEXT OF THE TEST.

FOR COMPLEX PROGRAMS, THE LIFE CYCLE MAINTENANCE ASPECTS ALSO BECOME IMPORTANT IN CHOOSING THE PROPER TESTING STRATEGY. AREAS THAT CHANGE OFTEN MAY USE A DIFFERENT TYPE OF TESTING THAN AN AREA WHICH DOESN'T.
TESTING

- A TESTED PROGRAM IS ONE WHICH YOU HAVE NOT YET FOUND THE CONDITIONS THAT MAKE IT FAIL.

- TESTING METHODS MUST BE ESTABLISHED ACCORDING TO EACH PROJECT.
  - TYPE OF SYSTEM DIFFERS
  - SIZE OF PROGRAMS VARY

- PRIMARY CATEGORIES OF TESTING
  - UNIT LEVEL
  - INTEGRATION
INSTRUCTOR NOTES

TESTING COSTS GO UP ALMOST EXPONENTIALLY AS PROGRAMS BECOME LARGER. THUS TESTING IS OFTEN NOT DONE TO THE LEVEL REQUIRED. MOREOVER, IN LARGE SYSTEMS INTEGRATION TESTING AS WELL AS UNIT TESTING IS REQUIRED.
### TESTING

<table>
<thead>
<tr>
<th>PROGRAM TYPE</th>
<th>PROGRAM CHARACTERISTICS</th>
<th>TESTING CHARACTERISTICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMPLE PROGRAMS</td>
<td>- LESS THAN 1000 STATEMENTS</td>
<td>- ANY METHOD WILL DO</td>
</tr>
<tr>
<td></td>
<td>- ONE PROGRAMMER FOR ABOUT 1 YEAR</td>
<td>- EASY TO FIND AND FIX BUGS</td>
</tr>
<tr>
<td></td>
<td>- NO INTERACTIONS WITH OTHER PROGRAMS</td>
<td></td>
</tr>
<tr>
<td>MEDIUM PROGRAMS</td>
<td>- LESS THEN 10,000 STATEMENTS</td>
<td>- EFFORT OF 6-12 MONTHS</td>
</tr>
<tr>
<td></td>
<td>- 1-5 PROGRAMMERS FOR LESS THEN 2 YEARS</td>
<td>- BRUTE FORCE STILL WORKS, BUT COSTLY</td>
</tr>
<tr>
<td></td>
<td>- FEW INTERACTIONS WITH OTHER PROGRAMS</td>
<td>- ORGANIZED TESTING CAN BE EFFECTIVE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- STILL EASY TO LOCATE BUGS</td>
</tr>
<tr>
<td>COMPLEX PROGRAMS</td>
<td>- LESS THAN 100,000 STATEMENTS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- 5-20 PROGRAMMERS FOR ABOUT 2-3 YEARS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- OFTEN INTERACTS WITH OTHER PROGRAMS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- 100-1,000 MODULES</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- SEVERAL SUBSYSTEMS</td>
<td></td>
</tr>
<tr>
<td>NEARLY IMPOSSIBLE PROGRAMS</td>
<td>- OVER 100,000 BUT OFTEN OVER 1,000,000</td>
<td>- COORDINATION AMONG SEVERAL TEAMS NEEDED</td>
</tr>
<tr>
<td></td>
<td>- MORE THAN 100 PROGRAMMERS, BUT SOMETIMES 1,000</td>
<td>- TURNOVER OF PERSONNEL</td>
</tr>
<tr>
<td></td>
<td>- BETWEEN 5-10 YEAR DEVELOPMENT TIME</td>
<td>- REQUIRES GOOD RECORDS</td>
</tr>
<tr>
<td></td>
<td>- 1,000 - 10,000 MODULES</td>
<td>- RETESTING REQUIRED TO VERIFY REQUIREMENT CHANGES</td>
</tr>
<tr>
<td></td>
<td>- COMPLEX INTERACTIONS WITH OTHER PROGRAMS</td>
<td>- HARD TO LOCATE ERRORS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ORGANIZED TESTING REQUIRED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VG 778.1  28-5
UNIT TESTING

- UNIT TESTING IS THE COMPLETE VERIFICATION OF AN INDIVIDUAL MODULE

- UNIT TESTING REQUIRES US TO CONSIDER
  - TESTING APPROACHES
  - TESTING PRINCIPLES
  - TEST CASE DESIGN
  - TESTING STRATEGY
INSTRUCTOR NOTES

BLACK-BOX TESTING USES THE SPECIFICATION TO DEVELOP TEST CASES AND IS MOST APPROPRIATE FOR SYSTEM TESTING.

WHITE-BOX TESTING USES DESIGN INFORMATION TO DEVELOP TEST CASES AND IS MOST APPROPRIATE FOR COMPONENT TESTING.
TESTING APPROACHES

• BLACK-BOX TESTING
  - DOES NOT USE DESIGN KNOWLEDGE
  - REQUIRES **EXTERNAL SPECIFICATION**
  - SPECIFY INPUTS/OBSERVE OUTPUTS

• WHITE BOX TESTING
  - BASED ON INTERNAL DESIGN LOGIC
  - REQUIRES UNDERSTANDING (DOCUMENTATION) OF DESIGN
  - EMPHASIZES PATH/BRANCH COVERAGE
INSTRUCTOR NOTES

THESE ARE THE MOST COMMON TEST CASES. EACH WILL BE EXPLAINED IN SUBSEQUENT SLIDES.
TEST-CASE DESIGN

- BLACK-BOX
  - EQUIVALENCE PARTITIONING
  - BOUNDARY-VALUE ANALYSIS
  - CAUSE - EFFECT GRAPHING
  - ERROR GUESSING

- WHITE-BOX
  - LOGIC - COVERAGE TESTING
EQUIVALENCE PARTITIONING

- REQUIRES EXTERNAL, FUNCTIONAL SPECIFICATION
- PARTITION EXTERNAL CONDITIONS INTO VALID AND INVALID EQUIVALENCE CLASSES
  - "ITEM COUNT CAN BE FROM 1 TO 999"
  - 1 VALID CLASS 1 < ITEM COUNT < 999
  - 2 INVALID CLASSES: ITEM COUNT < 1, ITEM COUNT > 999
  - "COLOR CAN BE RED OR GREEN"
  - 2 VALID CLASSES, 1 INVALID CLASS
- ASSIGN UNIQUE MEMBERS TO EACH CLASS
- WRITE NEW TEST CASES COVERING AS MANY VALID CLASSES AS POSSIBLE
- WRITE SEPARATE TEST CASE FOR EACH INVALID TEST CASE
BOUNDARY VALUE ANALYSIS

- Usually used to augment equivalence partitioning

- Add tests to check "ends" of valid range: item count = 1, item count = 999

- Add tests to check limits in number of parameters

- Focus on output values as well

- Search for other boundaries
CAUSE EFFECT GRAPHING

- BREAKS SPECIFICATION INTO SUBSETS
- WITHIN SUBSET IDENTIFY "CAUSES" AND "EFFECTS"
- DRAW BOOLEAN GRAPH LINKING CAUSES AND EFFECTS
- ANNOTATE WITH CONSTRAINTS TO IDENTIFY IMPOSSIBLE COMBINATIONS
- CONVERT TO DECISION TABLE
- EACH COLUMN BECOMES A TEST CASE
ERROR GUESSING

- INTUITIVE AND AD-HOC

- IDENTIFY ERROR-PRONE SITUATIONS

- EXAMPLES - FOR A SORT ROUTINE

  - INPUT LIST IS EMPTY
  - INPUT LIST CONTAINS ONE ITEM
  - ALL ENTRIES HAVE SOME VALUE
  - INPUT LIST ALREADY SORTED
WHILE VERY COMPLETE, A DETAILED KNOWLEDGE OF THE SYSTEM IS REQUIRED. OFTEN THE
DOCUMENTATION IS NOT COMPLETE ENOUGH TO DO WHITE-BOX TESTING.
LOGIC - COVERAGE TESTING (WHITE-BOX)

- **STATEMENT COVERAGE**
  - EXECUTE EACH STATEMENT AT LEAST ONCE

- **DECISION COVERAGE**
  - EACH BRANCH DECISION TAKES ON ALL POSSIBLE VALUE

- **CONDITION COVERAGE**
  - EACH CONDITION (IN DECISION) TAKES ON ALL POSSIBLE VALUES

- **DECISION/CONDITION COVERAGE**
  - ALL COMBINATIONS OF ABOVE

- **MULTIPLE CONDITION COVERAGE**
INTEGRATION TESTING

- INTEGRATION TESTING IS THE "COMPLETE" VERIFICATION
  OF THE SET OF MODULES THAT MAKE UP THE SYSTEM

- BUILDS ON PREVIOUSLY UNIT TESTED MODULES

- INTEGRATION TESTING REQUIRES US TO CONSIDER

- TESTING APPROACHES

- INTEGRATION STRATEGIES
INSTRUCTOR NOTES

PROBABLY THE MOST COMMON APPROACH USED. ALSO THE ONE THAT HAS THE LEAST CHANCE OF SUCCESS AS THE SYSTEM GROWS LARGER IN SIZE.
INTEGRATION TESTING APPROACHES  
(BIG BANG)  

- COMBINE ALL MODULES INTO A SYSTEM AND TEST THE CONGLOMERATE

"THE DEADLINE IS NOT MET, AND THE WORKERS CONTINUE TO STUMBLE THROUGH THE TESTING PHASE UNTIL EITHER THEY WEAR DOWN THE PROBLEMS OR THE PROBLEMS WEAR THEM DOWN."

- THIS IS THE OLDEST APPROACH WHICH EVOLVED FROM UNSTRUCTURED TESTING (DEBUGGING) OF SMALL PROGRAMS
INTEGRATION TESTING APPROACHES
(BIG BANG)

PROBLEMS
- NO WAY TO PREDICT WHAT KIND (EASY OR HARD)
  OF ERROR WILL OCCUR NEXT
- ERRORS ARE DIFFICULT TO IDENTIFY
- FIXING ONE ERROR SPAWNS NEW ERRORS
- NEVER SURE WHEN TESTING WILL FINISH

CONSEQUENCES
- MANAGEMENT OF PEOPLE AND COMPUTER RESOURCES
  IS MADE DIFFICULT
- NO FLEXIBILITY IN THE APPROACH
- BIG SYSTEMS NEVER QUITE WORK
INSTRUCTOR NOTES

THIS METHOD IS POSSIBLY THE ONLY WAY TO HANDLE LARGE SYSTEMS. BUT IT DOES TAKE TIME AND CAREFUL THOUGHT.
INTEGRATION TESTING APPROACHES
(INCREMENTAL)

- CAME ABOUT WHEN LARGE PROGRAMS STARTED BEING BUILT
- USEFUL FOR MEDIUM TO COMPLEX PROGRAMS
- OFFERS SOME CONTROL OF THE TESTING PROCESS
- BASIC APPROACH

1. WRITE AND TEST EACH MODULE SEPARATELY
2. COMBINE TWO MODULES TOGETHER INTO A COMBINATION
3. TEST THE COMBINATION
4. COMBINE ANOTHER MODULE WITH THE COMBINATION
5. GO TO STEP 3
INSTRUCTOR NOTES

THERE ARE VERY FEW DISADVANTAGES TO THIS METHOD. ANOTHER ADVANTAGE IS THAT THE SYSTEM DEVELOPMENT CAN BE STAGGERED TO PROVIDE THE MODULES FOR TESTING IN PROPER ORDER, THUS MAKING THE DEVELOPMENT MORE EFFICIENT.
INTEGRATION TESTING APPROACHES
(INCREMENTAL TESTING)

- ADVANTAGES INCLUDE

  INTERFACE DISCREPANCIES CAN BE EASILY IDENTIFIED AND CORRECTED

  YOU CAN PICK WHICH MODULE TO TEST NEXT

  ERRORS ARE LOCALIZED TO THE NEW MODULE AND ITS INTERCONNECTIONS

  FIXING AN ERROR DOESN'T SPAWN NEW ERRORS

  TESTING IS DONE WHEN THE LAST MODULE IS TESTED
PROGRAM CORRECTNESS EXAMPLE
(ANNOTATED WITH ASSERTIONS)

(True)

[0 + leaves in tree (Tree) = leaves in tree (Tree)]
Make Empty (Stack);
[leaves in stack (Stack) + leaves in tree (Tree) = leaves in tree (Tree)]
Count := 0;

(Count +
leaves in stack (Stack) +
leaves in tree (Tree) = leaves in tree (Tree))
Push (Tree, Onto => Stack);

(Count + leaves in stack (Stack) = leaves in tree (Tree))
while not Is_Empty (Stack) loop

(Count +
leaves in stack ([stack with top item removed]) +
leaves in tree ([top item on stack]) = leaves in tree (Tree))
Pop (T, Off Of => Stack);

(Count +
leaves in stack (Stack) +
leaves in tree (T) = leaves in tree (Tree))
if Is Leaf (T) then

(Count + 1 + leaves in stack (Stack) = leaves in tree (Tree))
Count := Count + 1;
else

(Count +
leaves in stack (Stack) +
leaves in tree (Left Subtree (T)) +
leaves in tree (Right Subtree (T)) = leaves in tree (Tree))
Push (Left Subtree (T), Onto => Stack);

(Count +
leaves in stack (Stack) +
leaves in tree (Right Subtree (T)) = leaves in tree (Tree))
Push (Right Subtree (T), Onto => Stack);
end if;

(Count + leaves in stack (Stack) = leaves in tree (Tree))
end loop;

(Count = leaves in tree (Tree))
INSTRUCTOR NOTES

SOME COMPUTER SCIENTISTS, ESPECIALLY THOSE OF A MATHEMATICAL BENT, SEE CORRECTNESS AS THE (ONLY?) WAVE OF THE FUTURE.
INTEGRATION TESTING APPROACHES
(INCREMENTAL TESTING)

- CONSEQUENCES ARE

- STAFFING AND COMPUTER RESOURCE NEEDS CAN BE BETTER PREDICTED

- TROUBLE IN A SUBSYSTEM CAN BE IDENTIFIED EARLIER

- APPROACH IS FLEXIBLE (NEW AREA CAN BEGIN BEING TESTED WHILE SUSPENDING AN OLD AREA)

- GREATER CONFIDENCE THAT THE SYSTEM WORKS
INSTRUCTOR NOTES

AT THIS POINT, THE IMPORTANCE OF BOTH UNIT TESTING AND THE INTEGRATION STRATEGY COMES INTO PLAY.
INTEGRATION STRATEGIES

- THE WAY MODULES ARE COMBINED DURING INCREMENTAL TESTING IS CALLED INTEGRATION.

- INTEGRATION TAKES ONE OF TWO FORMS ...
  - TOP DOWN
  - BOTTOM UP
THIS STRATEGY IS VERY NATURAL TO A SYSTEMS DEVELOPMENT EFFORT.
TOP-DOWN STRATEGY

- IMPLEMENT THE TOP MODULE OF THE DESIGN FIRST

- SIMULATE THE MODULE'S SUBORDINATES BY STUB MODULES

- GRADUALLY WORK DOWN THE DESIGN, REPLACING STUBS BY REAL MODULES AND INTRODUCING NEW STUBS WHERE NECESSARY
TOP-DOWN INTEGRATION

- STUBS (ALIAS DUMMY MODULES) ARE STAND-INS FOR NOT-YET-WRITTEN SUBORDINATE MODULES.

- A STUB MAY:
  - DO NOTHING - NOT VERY USEFUL, BUT EASY TO IMPLEMENT.
  - DISPLAY A TRACE MESSAGE - TO ENABLE A TESTER TO SEE WHAT'S HAPPENING TO THE PROGRAM.
  - CONVERSE WITH AN INTERACTIVE TERMINAL - TO ALLOW, PERHAPS, A TESTER TO ENTER TEST DATA.
  - RETURN A CONSTANT NUMBER, RANDOM NUMBER OR A NUMBER FROM A TABLE.
  - BE A SIMPLER VERSION OF THE REAL MODULE - THIS IS HALF-WAY BETWEEN A STUB AND THE REAL THING.
  - TEST OR DISPLAY ITS DATA - TO ALLOW THE TESTER A FURTHER CHANCE TO MONITOR THE PROGRAM.
TOP-DOWN INTEGRATION

EXAMPLE

• HOW MIGHT THIS DESIGN BE TESTED TOP-DOWN?
  • THREE WAYS
TOP-DOWN INTEGRATION

EXAMPLE

1. DEPTH FIRST (TEST ALL INPUT) ...

2. BREADTH FIRST (TEST MAJOR FUNCTIONS FIRST) ...

3. BACKWARDS DEPTH FIRST (TEST OUTPUT) ...

\[\square = \text{STUB}\]

VG 778.1 28-26
TOP-DOWN INTEGRATION ADVANTAGES

- IMPORTANT FEEDBACK IS PROVIDED TO THE USER IN THE BEST POSSIBLE WAY.

- THE USER MAY TAKE DELIVERY OF SEVERAL SKELETON VERSIONS OF THE SYSTEM, WHICH WILL SMOOTH HIS TRANSITION FROM OLD SYSTEM TO NEW.

- THE PROJECT SHOULD BE IN BETTER POLITICAL SHAPE IF IT FALLS BEHIND SCHEDULE.

- MAJOR INTERFACES ARE TESTED EARLY AND OFTEN.

- IMPLEMENTORS GET A BOOST FROM SEEING SOMETHING WORKING.

- CODING AND TESTING CAN BEGIN BEFORE DESIGN IS FINISHED.
TOP-DOWN INTEGRATION ADVANTAGES

- Management and monitoring is more straightforward. For example:

  - Managers can estimate resources required to implement a whole design by summing the estimates for each part of the design.

  - Feedback is by solid deliverables rather than by non-operational code of unknown quality.

  - Progress is measured in more useful quantity: inch pebbles, not milestones.
DRIVERS ARE USUALLY HARDER TO WRITE THAN STUBS.
BOTTOM-UP INTEGRATION STRATEGY

- BOTTOM-UP INTEGRATION IS, IN GENERAL, THE INVERSE OF TOP-DOWN INTEGRATION ...

- IMPLEMENT A MODULE AT THE BOTTOM OF THE DESIGN.

- SIMULATE THE MODULE'S SUPERORDINATES BY A DRIVER.

- GRADUALLY WORK UP THE DESIGN REPLACING DRIVERS BY REAL MODULES.

- DRIVERS (ALIAS TEST HARNESSES) ARE STAND-INS FOR NOT-YET-WRITTEN SUPERORDINATE MODULES.
BOTTOM-UP INTEGRATION

HOW MIGHT THIS DESIGN BE TESTED BOTTOM UP?

- TWO WAYS
INSTRUCTOR NOTES

PICK THE MOST SUITED APPROACH.
BOTTOM-UP INTEGRATION

1. TEST DETAILS OF INPUT LOGIC ...

   A.  
   B.  
   C.  
   D.  ...

2. TEST DETAILS OF OUTPUT LOGIC ...

   A.  
   B.  
   ...  

\[ \text{= DRIVER} \]

VG 778.1  28-31
INSTRUCTOR NOTES

DRIVERS CAN FORCE A MODULE TO OPERATE UNDER UNUSUAL OR ILLEGAL COMBINATIONS OF INPUTS. A MODULE ONLY OPERATES ON "EXPECTED" DATA. YOU NEED TO KNOW MORE ABOUT THE BEHAVIOR OF A MODULE WHEN IT'S BEEN TESTED BOTTOM UP.
BOTTOM-UP INTEGRATION
PLUSES AND MINUSES

+ STARTS OFF WITH GOOD PARALLEL DEVELOPMENT.

- PARALLEL DEVELOPMENT GETS HARDER TO MANAGE AS TEAMS REACH THE TOP OF THE DESIGN.

+ TESTS PHYSICAL I/O INTERFACES EARLY.

- TEST MAJOR INTERNAL INTERFACES LATE.

+ USEFUL FOR SOME CRITICAL MODULES.

- CODING AND TESTING CANNOT BEGIN BEFORE DESIGN FINISHES.

- SKELETON SYSTEMS DIFFICULT TO PRODUCE.
SUMMARY
TOP-DOWN VS. BOTTOM-UP

<table>
<thead>
<tr>
<th>TOP-DOWN</th>
<th>BOTTOM-UP</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEST FOR TESTING MAJOR Interfaces Early</td>
<td>BEST FOR TESTING PHYSICAL I/O EARLY</td>
</tr>
<tr>
<td>ASSUMES LOW LEVELS IN DESIGN WILL WORK AS PLANNED</td>
<td>ASSUMES LOW LEVELS WILL INTEGRATE AS PLANNED</td>
</tr>
<tr>
<td>CONTAINS NO SURPRISES</td>
<td>VERIFIES LOW-LEVEL ASSUMPTIONS</td>
</tr>
</tbody>
</table>

- IN THE REAL WORLD, A SINGLE STRATEGY WILL NOT WORK. A PROJECT NEEDS THE RIGHT MIXTURE OF BOTH.
INSTRUCTOR NOTES

THEME: THE QUALITY OF VARIOUS PRODUCTS DEVELOPED DURING THE SOFTWARE LIFE IS DETERMINED BY THE DEGREE A REVIEW THEY RECEIVE DURING THEIR DEVELOPMENT.

PURPOSE: TO PROVIDE A SET OF GUIDELINES FOR EFFECTIVE REVIEWS.

REFERENCES: FREEDMAN, D., WEINBERG, G. "WALKTHROUGHS, INSPECTIONS, AND TECHNICAL REVIEWS" LITTLE, BROWN, MA; 1977
YOURDON, E. "STRUCTURED WALKTHROUGHS" YOURDON, NYC; 1977
INSTRUCTOR NOTES

THIS AREA COVERS THE FULL RANGE OF REVIEWS DURING THE LIFE CYCLE IN A FAIRLY GENERAL MANNER.
REVIEWs

- QUESTIONS OFTEN ASKED ABOUT REVIEWS
  - WHY REVIEW?
  - WHO PARTICIPATES?
  - HOW ARE REVIEWS ORGANIZED?
  - HOW ARE REVIEWS RUN?
  - SHOULD REVIEWS REPEAT?
  - WHAT KIND OF REVIEWS ARE THERE?
  - WHAT SHOULD WE LOOK FOR?

- REMEMBER REVIEWS ARE REQUIRED IN ALL PHASES OF A
  SOFTWARE DEVELOPMENT
WHY REVIEW?

- BECAUSE TO ERR IS HUMAN.
  - REVIEWS ARE ESTABLISHED TECHNIQUES FOR REMOVING ERRORS

- SOFTWARE ENGINEERING IS A MODELING PROCESS. MODELS
  CONTAIN ERRORS. REVIEW IDENTIFIES AND CORRECTS ERRORS.

- THREE CATEGORIES OF ERRORS CAN EXIST IN MODELS.
  - INACCURACIES - MAKE THE MODEL LESS USEFUL
  - OMISSIONS - MAKE THE MODEL INEFFECTIVE OR DANGEROUS
    TO USE
  - EXTRAS - GET IN THE WAY, MAKING THE MODEL HARDER
    TO USE
INSTRUCTOR NOTES

INACCURACIES - WRONG STATEMENTS ARE DANGEROUS (E.G. MISPLACED DECIMAL POINT BLOWS UP A ROCKET)

OMISSIONS - WHAT YOU FORGOT IS ALWAYS MORE DANGEROUS THAN WHAT YOU WROTE DOWN (E.G. WHAT HAPPENS IF LIGHTENING HITS THE AIRPLANE AND THE SOFTWARE FAILS?)

EXTRAS - EXTRA JUNK JUST MAKES THE MODEL HARDER TO READ, OR CREATES EXTRA WORK.

ALL TYPES OF ERRORS ARE EQUALLY BAD FROM THE REVIEW PROCESS VIEWPOINT.
WHY REVIEW?

- INACCURACIES
  - Are simply things stated wrong in the model.
  - Example: System response time must be within 20 seconds.

- OMISSIONS
  - Are things that were forgotten.
  - Example: Your tax is 40% of your gross income.

- EXTRAS
  - Are things that don't belong in the model.
  - Example: Go down one block and turn left. Then make the next three right turns. Then take a left.

- ALL CATEGORIES OF ERRORS MUST BE REMOVED
INSTRUCTOR NOTES

IN MOST REVIEWS THE PARTICIPANTS ARE ASSIGNED THESE ROLES TO HELP FOCUS THE REVIEW ACTIVITY.

DON'T SET UP ORGANIZATIONALLY A "WATCH-DOG" TEAM FOR REVIEW. THEY WILL START TO WORRY ABOUT THEIR ROLE INSTEAD OF THE REVIEW.
WHO PARTICIPATES?

- In a typical review the following roles must be filled by selected participants.
  - The leader
  - The recorder
  - Someone old
  - Someone new
  - Someone borrowed
  - Someone blue

- In our discussion we assume the review is held as a meeting not as an informal discussion or individual review.
GOAL - THE LEADER OBTAINS A GOOD REVIEW OR REPORTS WHY GOOD REVIEW WASN'T ATTAINED IF THERE ARE PROBLEMS.

MANAGERS AND TEAM LEADERS CARRY CONFLICTS OF INTEREST INTO THE REVIEW.

REQUIRES GOOD MEETING SKILLS.

OFFENSIVE PEOPLE SHOULD NOT LEAD THE REVIEW. BUT AN OCCASIONAL AND WELL-TIMED REMARK OR GESTURE THAT GETS RESULTS IS VERY IMPORTANT.
WHO PARTICIPATES?

The Leader

- Focuses on obtaining a good review
- Is never a manager or team leader
- Prepares the review kits
- Conducts an orderly review
- Keeps the meeting on track
- Makes sure everyone participates
- Makes sure all important points are covered
INSTRUCTOR NOTES

THE RECORDER DOESN'T PROVIDE A "BLOW-BY-BLOW" ACCOUNT. HE.Focuses ON THE REAL ISSUES.
WHO PARTICIPATES?

- **THE RECORDER**

  - produces an accurate report of the review session
  - may take minutes (private)
  - may record on flip charts (public)
  - may be a technician or a librarian
  - is never the same person at every review
INSTRUCTOR NOTES

SHOULD BE AN ACTIVE, RESPECTED PERSON. NOT SOMEONE WHO TELLS "WAR-STORIES", OR SAYS CONSTANTLY "THIS WAS HOW WE USED TO DO IT, TEN YEARS AGO."
WHO PARTICIPATES?

- SOMEONE OLD
  - IS A SENIOR TECHNICAL PERSON IN THE PROJECT
  - IS A ROLE MODEL FOR JUNIOR STAFF
  - PROVIDES TECHNICAL DEPTH
  - ACTS IN A TEACHING CAPACITY
INSTRUCTOR NOTES

SHOULD NOT BE A SHY PERSON, AND SHOULD BE MADE TO FEEL HE CAN CONTRIBUTE.
WHO PARTICIPATES?

- SOMEONE NEW
  - IS A JUNIOR TECHNICAL PERSON IN THE PROJECT
  - LEARNS FROM SENIOR STAFF
  - LEADS A FRESH VIEW TO THE MATERIAL
  - SOMETIMES CONTRIBUTES NOVEL IDEAS
WHO PARTICIPATES?

- SOMEONE BORROWED

- COMES FROM OUTSIDE THE PROJECT

- LENDS A FRESH VIEW TO THE MATERIAL

- SOMETIMES CONTRIBUTES NOVEL IDEAS

- LEARNS NEW TECHNIQUES FROM OTHERS
WHO PARTICIPATES?

- SOMEONE BLUE

- IS THE PERSON WHO'S WORK IS BEING REVIEWED

- IS THE AUTHORITY ON THE WORK

- ANSWERS QUESTIONS ABOUT THE MATERIAL

- LEARNS ABOUT THE WORK'S QUALITY
INSTRUCTOR NOTES

TEAM LEADERS MUST NEVER CONDUCT A REVIEW OF THEIR OWN TEAM'S WORK.
HOW ARE REVIEWS ORGANIZED?

- TWO VERY DIFFERENT WAYS ...
  - AS A TEAM - RESPONSIBLE FOR OBTAINING A CONSENSUS ON THE QUALITY OF WORK BEING REVIEWED.
  - AS A GROUP INDIVIDUALS - RESPONSIBLE FOR GIVING THEIR TECHNICAL OPINIONS AND ADVICE TO THE AUTHOR.

- TEAM CHARACTERISTICS
  - MEET AND REACH CONSENSUS
  - COMMUNICATE VERBALLY
  - PUT TOGETHER SPECIFICALLY TO REACH A PROJECT GOAL

- INDIVIDUAL CHARACTERISTICS
  - GIVE WRITTEN FEEDBACK TO AUTHOR(S)
  - REVIEW AT AN AUTHOR'S DISCRETION
HOW ARE REVIEWS RUN?

- REVIEWS ARE RUN LIKE SUCCESSFUL MEETINGS; THEY ...
  - ARE PLANNED
  - HAVE PREPARED MATERIALS
  - HAVE INFORMED PARTICIPANTS
  - ARE STRUCTURED AND CONTROLLED
  - ARE RECORDED
HOW ARE REVIEWS RUN?

PLANNING

- Establishes the goal of the review
- Defines why the goal must be reached

PREPARING

- Means materials are packaged and sent to participants in a timely manner
- Means everyone can prepare for the review
- Means everyone knows...
  - why the review is being held
  - where it will take place
  - when it will occur
  - what will be covered
  - who will be there

INFORMING

- Means that there are procedures that guide everyone's thoughts to a common goal...
- Present the evidence
- Poll for interpretations
- Define a plan of action

STRUCTURING

- Means the meeting stays on track...
  - no arguments
  - no unnecessary tangents
  - rule out the overly critical
  - rule out the overly praiseworthy
  - proper allocation of time to each matter on the agenda

CONTROLLING

- Means everyone remembers...
  - resolutions
  - action items
  - the meeting's history

RECORDING

V5 778.1

29-13
INSTRUCTOR NOTES

SINCE NO ONE LIKES TO REPEAT REVIEWS, THE PARTICIPANTS MUST BE ENCOURAGED TO SAY "OKAY, IT'S GOOD UP TO THIS POINT. FIX THE REST, AND WE'LL REVIEW IT AGAIN."

INDIVIDUAL REVIEWS NORMALLY ONLY CONCENTRATE ON SMALL PORTIONS OF AN AUTHOR'S WORK.
SHOULD REVIEWS REPEAT?

TEAM REVIEWS REPEAT ONLY IF ...

- A QUORUM IS NOT ACHIEVED
- TIME LIMIT WAS EXCEEDED
- THE PRODUCT REQUIRES MAJOR WORK
- A CONSENSUS WAS NOT ACHIEVED

INDIVIDUAL REVIEWS REPEAT ...

- RARELY ONCE
- USUALLY BETWEEN 2 AND 4 TIMES
- UNTIL THE AUTHOR HAS ACHIEVED CONSENSUS WITH REVIEWERS
INSTRUCTOR NOTES

NOTE THE MULTIPLE DYNAMICS OF THE TEAM REVIEW.
SHOULD REVIEWS REPEAT?

- **THE TEAM REVIEW CYCLE:**
  - time, place, agenda, product
  - new product
  - no quorum
  - time limit exceeded
  - revised product
  - no consensus
  - resolve conflicts
  - rework guidelines

- **THE INDIVIDUAL REVIEW CYCLE:**
  - new/revised product
  - comments
  - reactions
  - reviewers

VG 778.1 29-15
INSTRUCTOR NOTES

THE GENERIC ASPECTS OF REVIEW ARE ALWAYS PRESENT, BUT MUST BE TAILORED TO THE PRODUCTS AND CIRCUMSTANCE ENCOUNTERED.
WHAT KINDS OF REVIEWS ARE THERE?

- AS MANY AS THERE ARE PRODUCTS PRODUCED BY THE PROJECT, AT LEAST ...
  - SPECIFICATION REVIEWS
  - DESIGN REVIEWS
  - CODE REVIEWS
  - TEST PLAN REVIEWS
  - DOCUMENTATION REVIEWS

- FOR EACH REVIEW YOU SHOULD CONSIDER
  - WHY IT'S DONE
  - WHO ATTENDS
  - WHAT CRITERIA EVALUATES QUALITY
  - HOW ITS GOALS ARE TO BE ACHIEVED
WHAT KINDS OF REVIEWS ARE THERE?

SPECIFICATION REVIEWS

WHY
TO CATCH OMISSIONS AND MISINTERPRETATIONS OF USER REQUESTS.

WHO
USERS, ANALYSTS AND DESIGNERS ATTEND.

WHAT
COMPLETENESS, CORRECTNESS, CONSISTENCY, PRECISION, TRACEABILITY

HOW
- CONCENTRATE ON A PARTICULAR USER VIEWPOINT (I.E. FACTION).

- REMOVE AMBIGUITIES BY:
  • REREADING WITH DIFFERENT STRESS
  • SUBSTITUTING SYNONYMS
  • REDRAWING
  • WORK OUT AN EXAMPLE
INSTRUCTOR NOTES

DESIGN REVIEWS ARE USUALLY THE LAST PLACE TO CATCH ERRORS BEFORE THE "RUBBER MEETS THE ROAD."

VG 778.1

29-181
WHAT KIND OF REVIEWS ARE THERE?

DESIGN REVIEWS

WHY
TO VERIFY ALL REQUIREMENTS ARE SATISFIED IN A FEASIBLE WAY.

WHO
ANALYSTS, DESIGNERS AND IMPLEMENTERS.

WHAT
SUBSYSTEM AND USER INTERFACES, UNNECESSARY CONSTRAINTS, INFEASIBLE STRATEGIES.

HOW
- COMPARE ALTERNATE STRATEGIES.
- ESTABLISH EVALUATION METRICS.
- RECORD ISSUES AND POSSIBLE ALTERNATE SOLUTIONS.
INSTRUCTOR NOTES

BY THIS TIME, THE REVIEW SHOULD ONLY BE CONCERNED WITH THE CODE, NOT THE DESIGN. IF DESIGN PROBLEMS APPEAR, ANOTHER DESIGN REVIEW IS PROBABLY NECESSARY.
WHAT KIND OF REVIEWS ARE THERE?

**CODE REVIEWS**

**WHY**
- TO VERIFY CORRECT IMPLEMENTATION OF THE DESIGN.

**WHO**
- DESIGNERS, IMPLEMENTERS, MAINTAINERS

**WHAT**
- SUBPROGRAM INTERFACES, DATA STRUCTURES, MACHINE UTILIZATION, STYLE, COMMENTS.

**HOW**
- ADHERENCE TO STANDARDS.
- DESCRIPTIONS (E.G. PACKAGE Specs) BEFORE WORKINGS (E.G. PACKAGE BODIES).
- TRACE BACK TO DESIGN AND TO SPECIFICATION.
WHAT KIND OF REVIEWS ARE THERE?

TEST PLAN REVIEWS

WHY
TO VERIFY TEST DETAILS SATISFY ACCEPTANCE CRITERIA.

WHO
ANALYSTS, USERS, DESIGNERS, IMPLEMENTERS, MAINTAINERS.

WHAT
A TEST FOR: EVERY REQUIREMENT, SYSTEM BEHAVIOR, SYSTEM FUNCTION, ERROR SITUATION, BACKUP AND RECOVERY SITUATION

HOW
- TRACE BACK TO SPECIFICATION, DESIGN AND CODE.
- CLUSTER TESTS INTO CATEGORIES (TO VERIFY COVERAGE).
INSTRUCTOR NOTES

These reviews are often skimmed over, but are very necessary. If the documentation is poor or incomplete, at maintenance time the cost of fixes will assuredly skyrocket.
WHAT KIND OF REVIEWS ARE THERE?

DOCUMENTATION REVIEWS

WHY

TO ENSURE TECHNICAL CORRECTNESS ON HOW THE SYSTEM IS OPERATED AND MAINTAINED.

WHO

USERS, MAINTAINERS, IMPLEMENTERS, TECHNICAL WRITERS.

WHAT

EVIDENCE OF OPERATIONAL TESTS, ERROR MESSAGES, USER INTERFACE DESCRIPTION, INSTALLATION.

HOW

- TRACE BACK TO SPECIFICATION.
- VERIFY TEST CASE FOR EACH USER INTERFACE FEATURE.
- FORM ON ERROR LIST AND ENFORCE ASSOCIATED CORRECTIONS.
INSTRUCTOR NOTES

THERE IS TOO MUCH TO COVER IN THIS LAST SECTION, SO HIT WHATEVER IMPORTANT POINTS:
(1) YOU WANT TO MAKE, AND (2) THAT ARE APPROPRIATE FOR YOUR AUDIENCE. TELL STUDENTS TO
CONSULT THE FREEDMAN AND WEINBERG TEXT FOR MORE DETAILS.

POINT OUT THAT MOST ORGANIZATIONS WILL DEVELOP FROM THESE TYPES OF CHECKLISTS A UNIQUE
SET OF GUIDELINES.
WHAT SHOULD WE LOOK FOR?

SPECIFICATION REVIEW CHECKLIST/TECHNIQUES...

1. VARY THE STRESS PATTERN IN A SENTENCE TO REVEAL POSSIBLE ALTERNATIVE MEANINGS.
2. WHEN A TERM IS DEFINED EXPLICITLY SOMEWHERE, TRY SUBSTITUTE IT DEFINITION IN PLACE OF THE TERM.
3. WHEN A STRUCTURE IS DESCRIBED IN WORDS, TRY TO SKETCH A PICTURE OF THE STRUCTURE BEING DESCRIBED.
4. WHEN A STRUCTURE IS DESCRIBED BY A PICTURE, TRY TO REDRAW THE PICTURE IN A FORM THAT EMPHASIZES DIFFERENT ASPECTS.
5. WHEN THERE IS AN EQUATION, TRY EXPRESSING THE MEANING OF THE EQUATION IN WORDS.
6. WHEN A CALCULATION IS SPECIFIED OR IMPLIED IN WORDS, TRY EXPRESSING IT IN AN EQUATION.
7. WHEN A CALCULATION IS SPECIFIED, WORK AT LEAST TWO EXAMPLES BY HAND AND GIVEN THEM AS EXAMPLES IN THE SPECIFICATION.
8. LOOK FOR STATEMENTS THAT IN ANY WAY IMPLY CERTAINTY AND THEN ASK FOR PROOF.
9. WHEN YOU ARE SEARCHING BEHIND CERTAINTY STATEMENTS, PUSH THE SEARCH BACK AS MANY LEVELS AS ARE NEEDED TO ACHIEVE THE KIND OF CERTAINTY A COMPUTER WILL NEED.
10. BE ON THE LOOKOUT FOR WORDS THAT ARE ASSUMED TO BE PERSUASIVE, SUCH AS CERTAINLY, THEREFORE, CLEARLY, OBVIOUSLY, OR AS ANY FOOL CAN PLAINLY SEE.
11. WATCH FOR VAGUE WORDS, SUCH AS SOME, SOMETIMES, OFTEN, USUALLY, ORDINARILY, CUSTOMARILY, MOST, OR MOSTLY.
12. WHEN LISTS ARE GIVEN, BUT NOT COMPLETED, MAKE SURE THAT THERE IS A COMPLETE UNDERSTANDING OF THE NATURE OF THE SUBSEQUENT ITEMS. WATCH OUT FOR ETC., AND SO FORTH, AND SO ON, OR SUCH AS.
13. IN ATTEMPTING TO CLARIFY LISTS, AS IN (12), WE SOMETIMES STATE A RULE. BE SURE THAT THE RULE DOESN'T CONTAIN UNSTATED ASSUMPTIONS.
14. THE ABOVE PROBLEMS (11, 12, AND 13) MIGHT HAVE BEEN CAUGHT BY LOOKING FOR LISTS WITHOUT EXAMPLES OR EXAMPLES THAT ARE TOO FEW OR TOO SIMILAR TO EACH OTHER TO EXPLICATE THE RULE.
15. BEWARE OF VAGUE VERBS, SUCH AS HANDLED, PROCESSED, REJECTED, SKIPPED, OR ELIMINATED.
16. PASSIVE VOICE CONSTRUCTIONS ARE ALSO TRAPS. SINCE THE PASSIVE VOICE DOESN'T NAME AN ACTOR, IT'S EASY TO OVERLOAD HAVING ANYBODY DO THE WORK.
17. BE ESPECIALLY ON THE LOOKOUT FOR COMPARATIVES WITHOUT REFERENCES. ALWAYS ASK, "COMPARED WITH WHAT?"
18. PRONOUNS ARE OFTEN CLEAR TO THE WRITER AND NOT TO THE READER. KEY ON THE OCCURRENCES OF IT, HE, SHE, THEY, HIS, HERS, ITS, THEIR, WE, US, OUR, YOU, YOUR, AND TRY TO FIND THE REFERENT AND STATE IT EXPLICITLY.
WHAT SHOULD WE LOOK FOR?

ARCHITECTURAL DESIGN REVIEW CHECKLIST ...

1. ARE THE PRELIMINARY DESIGN OBJECTIVES CLEARLY STATED?
2. DOES THE PRELIMINARY DESIGN DOCUMENT CONTAIN A DESCRIPTION OF THE PROCEDURE THAT WAS USED TO DO PRELIMINARY DESIGN OR IS THERE A REFERENCE TO SUCH A PROCEDURE?
3. IS THERE A LIST OF THE FUNCTIONS THAT ARE TO BE PROVIDED BY THE COMPUTING SYSTEM?
4. IS THERE A MODEL OF THE USER INTERFACE TO THE COMPUTING SYSTEM?
5. ARE THERE MODELS AND/OR DESCRIPTIONS OF ALL OTHER INTERFACES TO THE COMPUTING SYSTEM?
6. IS THERE A HIGH LEVEL FUNCTIONAL MODEL OF THE PROPOSED COMPUTING SYSTEM?
7. ARE THE MAJOR IMPLEMENTATION ALTERNATIVES AND THEIR EVALUATIONS REPRESENTED IN THE DOCUMENT?
8. IS THERE A RECOMMENDATION FROM THE PRELIMINARY DESIGN TEAM TO IMPLEMENT ONE OF THE ALTERNATIVES?
9. IS THE RECOMMENDATION OF THE PRELIMINARY DESIGN TEAM ADEQUATELY SUPPORTED?
10. DOES THE INFORMATION PRESENTED IN THE PRELIMINARY DESIGN DOCUMENT AND DURING THE PRELIMINARY DESIGN REVIEW GIVE YOU CONFIDENCE THAT THE COMPUTING SYSTEM CAN BE IMPLEMENTED TO SATISFY THE REQUIREMENTS TO SUCH AN EXTENT THAT YOU WOULD USE THE PROPOSED SYSTEM?
WHAT SHOULD WE LOOK FOR?

DESIGN REVIEW CHECKLISTS

1. BOUNDARY OVERSIGHTS
   a. Is anything going to fall between the cracks of "mine" versus "yours."
   b. Is anything going to be claimed by two or more parties?
   c. Is each input, function, and output specifically addressed by a specific, identifiable part of
      the system? Can you prove it?
   d. Is there any misinterpretation of the person-machine interface -- either by person or machine?

2. OVERADAPTATION
   a. Has any portion of this design received more emphasis than it seems to deserve? Can you
      explain why that happened, and what effects it has had?
   b. Is the design overly constrained, perhaps by paying too much attention to one part at the
      expense of others?
   c. If you could relax any single constraint, which would it be? How would the design be affected?

3. AFTERTHOUGHTS
   Examine the last three things added to the design and answer the following questions for each:
   a. What has been crammed in?
   b. Could someone tell that the change was not part of the original conception -- that it is a
      patch to the design?
   c. What wasn't considered when this change was made?
   d. What would happen if this change were undone and left out of the final design?

4. VESTIGES
   a. What things are in the design because "we've always done it that way?" Why are you doing it
      that way?
   b. Does the design reflect the machine on which it will operate? If so, why?
   c. Is your design independent of the programming language that will be used? If not, why not?

5. MISTAKES
   a. What have you forgotten?
   b. What has been done wrong?
   c. Did you do the I's?
   d. Did you ever go back and correct that problem you found when you were busy with something more
      important?
   e. Do you have any notes on scraps of paper?

6. INSENSITIVITY
   a. Have you remembered the people who will have to use this system?
   b. Have you remembered the people who will have to operate it?
   c. Have you remembered the people who will have to repair it?
   d. If you were one of these people, what one thing would you change in the design to make your
      life easier? Why wasn't that change made?
WHAT SHOULD WE LOOK FOR?

• CODE REVIEW CHECKLIST/QUESTIONS

FUNCTION
1. IS THERE A CONCEPT, AN UNDERLYING IDEA, THAT CAN BE EXPRESSED EASILY IN
PLAIN LANGUAGE? IS IT EXPRESSED IN PLAIN LANGUAGE IN THE IMPLEMENTED CODE?
2. DOES THE FUNCTION OF THIS PART HAVE A CLEAR PLACE IN THE OVERALL FUNCTION
OF THE WHOLE, AND IS THIS FUNCTION CLEARLY EXPRESSED?
3. IS THE ROUTINE PROPERLY SHELTERED, SO THAT IT MAY PERFORM ITS FUNCTION
RELIABLY IN SPITE OF POSSIBLE MISUSE?

FORM
1. WHATEVER STYLE IS ADOPTED, IS IT CLEAN AND CLEAR WHEN TAKEN AS A WHOLE?
2. IS IT MEANINGFUL TO ALL CLASSES OF READERS WHO WILL SEE IT?
3. ARE THERE REPEATED CODE SEGMENTS, WHETHER WITHIN OR BETWEEN ROUTINES?
4. ARE COMMENTS USEFUL OR ARE THEY SIMPLY ALIBIS FOR POOR CODING?
5. IS THE LEVEL OF DETAIL CONSISTENT?
6. ARE STANDARD PRACTICES USED?
7. IS INITIALIZATION PROPERLY DONE, AND DOES THE ROUTINE CLEAN UP AFTER ITSELF?

ECONOMY
1. ARE THERE REDUNDANT OPERATIONS FOR WHICH THERE IS NO COMPENSATING BENEFIT?
2. IS STORAGE USE CONSISTENT, BOTH INTERNALLY AND WITH EXTERNAL SPECIFICATIONS?
3. HOW MUCH WILL IT COST TO MODIFY? (CONSIDER THE THREE MOST LIKELY FUTURE
MODIFICATIONS.)
4. IS IT SIMPLE?
WHAT SHOULD WE LOOK FOR?

- **CODE REVIEW STYLE GUIDELINES**
  - WRITE CLEARLY -- DON'T BE TOO CLEVER.
  - SAY WHAT YOU MEAN, SIMPLY AND DIRECTLY.
  - USE LIBRARY FUNCTIONS.
  - AVOID TEMPORARY VARIABLES.
  - WRITE CLEARLY -- DON'T SACRIFICE CLARITY FOR "EFFICIENCY.
  - LET THE MACHINE DO THE DIRTY WORK.
  - REPLACE REPEITIVE EXPRESSIONS BY CALLS TO A COMMON FUNCTION.
  - PARENTHESIZE TO AVOID AMBIGUITY.
  - CHOOSE VARIABLE NAMES THAT WON'T BE CONFUSED.
  - AVOID UNNECESSARY BRANCHES.
  - DON'T USE CONDITIONAL BRANCHES AS A SUBSTITUTE FOR A LOGICAL EXPRESSION.
  - IF A LOGICAL EXPRESSION IS HARD TO UNDERSTAND, TRY TRANSFORMING IT.
  - USE DATA ARRAYS TO AVOID REPEITIVE CONTROL SEQUENCES.
  - CHOOSE A DATA REPRESENTATION THAT MAKES THE PROGRAM SIMPLE.
  - WRITE FIRST IN AN EASY-TO-UNDERSTAND PSEUDO-LANGUAGE; THEN TRANSLATE INTO WHATEVER LANGUAGE YOU HAVE TO USE.
  - MODULARIZE. USE SUBROUTINES.
  - USE GOTOS ONLY TO IMPLEMENT A FUNDAMENTAL STRUCTURE.
  - AVOID GOTOS COMPLETELY IF YOU CAN KEEP THE PROGRAM READABLE.
  - DON'T PATCH BAD CODE -- REWRITE IT.
  - WRITE AND TEST A BIG PROGRAM IN SMALL PIECES.
  - USE RECURSIVE PROCEDURES FOR RECURSIVELY-DEFINED DATA STRUCTURES.
  - TEST INPUT FOR PLAUSIBILITY AND VALIDITY.
  - MAKE SURE INPUT DOESN'T VIOLATE THE LIMITS OF THE PROGRAM.
  - TERMINATE INPUT BY END-OF-FILE OR MARKER, NOT BY COUNT.
  - IDENTIFY BAD INPUT; RECOVER IF POSSIBLE.
  - MAKE INPUT EASY TO PREPARE AND OUTPUT SELF-EXPLANATORY.
  - USE UNIFORM INPUT FORMATS.
  - MAKE INPUT EASY TO PROOFREAD.
  - USE FREE-FORM INPUT WHEN POSSIBLE.
  - USE SELF-IDENTIFYING INPUT. ALLOW DEFAULTS.
  - ECHO BOTH ON OUTPUT.
  - MAKE SURE ALL VARIABLES ARE INITIALIZED BEFORE USE.
  - DON'T STOP AT ONE BUG.
  - TAKE CARE TO BRANCH THE RIGHT WAY ON EQUALITY.
  - BE CAREFUL WHEN A LOOP EXITS TO THE SAME PLACE FROM SIDE AND BOTTOM.
  - MAKE SURE YOUR CODE "DOES NOTHING" GRACEFULLY.
  - TEST PROGRAMS AT THEIR BOUNDARY VALUES.
  - CHECK SOME ANSWERS BY HAND.
  - 10.0 TIMES 0.1 IS HARDLY EVER 1.0.
  - DON'T COMPARE FLOATING POINT NUMBERS SOLELY FOR EQUALITY.
  - MAKE IT RIGHT BEFORE YOU MAKE IT FASTER.
  - MAKE IT FAIL-SAFE BEFORE YOU MAKE IT FASTER.
  - MAKE IT CLEAR BEFORE YOU MAKE IT FASTER.
  - DON'T SACRIFICE CLARITY FOR SMALL GAINS IN "EFFICIENCY."
WHAT SHOULD WE LOOK FOR?

**DOCUMENTATION REVIEW CHECKLIST ...**

1. HAVE ALL PHASES OF THE DOCUMENT'S LIFE CYCLE BEEN CONSIDERED?
   a. Is there provision for user feedback?
   b. Is there provision for making changes?
   c. Will changes in the system cause difficult or expensive changes in the documentation?
   d. Is there adequate provision for distribution of the documents?
   e. Is there adequate provision for the distribution of changes to the documents?
   f. Can documents be reproduced easily?
   g. Can copying be prevented or controlled?
   h. Are there available people to supplement documents?
   i. Do the user and creators agree on the purpose of the documents?
   j. Is there adequate provision for keeping support people current and informed?
   k. Are tools available (e.g., fiche readers, terminals) for reading/accessing/storing these materials?
   l. Have the documents been properly approved?
   m. Do these documents show where they fall in the total plan?
   n. Do the documents indicate other documents that may be used as follow-up?

2. ARE THE CONTENTS OF THE DOCUMENTS ADEQUATE?
   a. Coverage of topics
   b. Correctness
   c. Evidence

3. ARE THE MATERIALS IN THE DOCUMENTS CLEAR?
   a. Are examples clear?
   b. Are diagrams, pictures, or other visual materials clear?
   c. Is terminology clear?
   d. Is writing style clear?

4. ARE THE DOCUMENTS ADEQUATELY SUPPLIED WITH REFERENCING AIDS?
   a. Is there a table of contents, if appropriate?
   b. Is the table of contents well placed?
   c. Is there an index, if appropriate?
   d. Is the index well placed?
   e. Is the table of contents correct?
   f. Is the index correct?
   g. Is there a bibliography of prerequisite publications?
   h. Is there a bibliography of related publications which may contain further information?
   i. Does the organization of the documents themselves contribute to the ease of finding information?
WHAT SHOULD WE LOOK FOR?

SYSTEM TESTING REVIEW CHECKLIST ...

I. TEST PREPARATION

1. TEST TEAM MEMBERS GIVEN ASSIGNMENTS.
2. FINAL OPERATING ENVIRONMENT DEFINED.
3. ALL CONDITIONS TO BE TESTED, IDENTIFIED, AND DOCUMENTED.
4. TEST FILES CREATED FOR EXECUTION OF ALL SYSTEM FUNCTIONS.
5. TEST TRANSACTIONS DEVELOPED TO DEMONSTRATE ALL TEST CASES.
6. TEST RESULTS PREDICTED AND DOCUMENTED FOR LATER COMPARISON WITH COMPUTER OUTPUT.
7. PARALLEL SYSTEM OUTPUT IDENTIFIED FOR STANDARD OF CORRECTNESS.
8. ACCEPTABILITY AND ACCURACY STANDARDS ESTABLISHED AND DOCUMENTED FOR LATER ASSESSMENT OF RESULTS.

II. TEST OPERATIONS

1. TEST TRANSACTIONS ENTERED BY DATA ENTRY DEPARTMENT USING STANDARD PROCEDURES.
2. TESTS EXECUTED BY OPERATIONS PERSONNEL WITHOUT PROGRAMMER SUPERVISION.
3. TRANSACTION LISTING OBTAINED BEFORE TEST EXECUTION.
4. FILE DUMPS OBTAINED PRIOR TO PROGRAM EXECUTION.
5. SYSTEM EXECUTED TO DEMONSTRATE ALL TEST CASES. (MAY REQUIRE MULTIPLE EXECUTIONS OF THE SYSTEM OR OF INDIVIDUAL MODULES.)
6. ALL HARDCOPY (REPORTS, CONTROL TOTALS) OBTAINED.
7. TRANSCRIPTION OF CONSOLE MESSAGES (AND TIMING INFORMATION) OBTAINED.
8. FILE DUMPS TAKEN AFTER TEST EXECUTION.
9. OUTPUT FROM PARALLEL RUN AVAILABLE.

III. UNIT-TEST EVALUATION

1. JCL EXECUTES PROGRAM WITHOUT ERRORS OR UNNECESSARY OPERATOR INTERVENTION, AND FILE LABELS ARE CORRECT.
2. JCL-DEFINED FILE CAPACITIES ADEQUATE, ASSUMING MODERATE EXPANSION.
3. EXECUTE TIME ACCEPTABLE ACCORDING TO PREESTABLISHED STANDARDS.
4. SUFFICIENT CORE AVAILABLE FOR EXECUTION AND MODERATE EXPANSION OF PROGRAM SIZE.
5. INPUT DATA ACCEPTED AS FORMATTED WITHOUT DATA EXCEPTION.
6. ALL LOGICAL PATHS EXECUTED CORRECTLY.
7. CREATED FILES CONFORM TO DESIGN SPECIFICATIONS.
8. FIELD SIZES IN FILES ADEQUATE. NO UNEXPECTED TRUNCATION, LOSS OF SIGNIFICANT DIGITS.
9. MATHEMATICAL ACCURACY, ROUNDDING.
10. REPORTS CONFORM TO DESIGNED LAYOUT.
11. REPORT INFORMATION CONTENTS AGREES WITH TITLES, HEADERS.
12. REPORT PAGE NUMBERING ACCURATE.
13. SPELLING CORRECT ON REPORT TITLES, HEADERS.
14. FIELD SIZES ON REPORTS ADEQUATE.
15. AUDIT TRAILS (CONTROL TOTALS) ACCURATE.
WHAT SHOULD WE LOOK FOR?

IV. SYSTEM TEST EVALUATION

1. JCL EXECUTES SYSTEM WITHOUT ERRORS OR UNNECESSARY OPERATOR INTERVENTION, AND FILE LABELS ARE CORRECT.
2. JCL-DEFINED FILE CAPACITIES ADEQUATE, ASSUMING MODERATE EXPANSION.
3. EXECUTE TIME ACCEPTABLE ACCORDING TO PREESTABLISHED STANDARDS.
4. FILE SEQUENCES (SORT SPECS) ACCURATE THROUGHOUT SYSTEM EXECUTION.
5. NO-FILE CASE EXECUTING THROUGHOUT ALL SYSTEM STEPS.
6. EMPTY-FILE CASE EXECUTING THROUGHOUT ALL SYSTEM STEPS.
7. AVAILABLE MEANS FOR CROSS-CHECKING SYSTEM RESULTS EXHAUSTED.
8. INTERFACES WORKING ACCURATELY.
9. FILES CLOSED AT EOJ (NORMAL OR DUE TO ERROR).
10. EXECUTION SEQUENCE (SYSTEM FLOW) ACCURATE AND WORKABLE.
11. AUDIT TRAILS (CONTROL TOTALS) ACCURATE.
12. ERROR-CORRECTION PROCEDURE LOOPS WORK.
13. SYSTEM RESTART PROCEDURES SATISFACTORY.
14. FILE RECOVERY PROCEDURES SATISFACTORY.
15. FILES AND REPORTS CONFORM TO OUTPUT FROM PARALLEL TEST, IF APPLICABLE.
16. OUTPUT ACCURATE AFTER MORE THAN ONE EXECUTION CYCLE.
17. END-OF-PERIOD OUTPUT ACCURATE, AND PERIOD-TOTALS RESET TO ZERO.

V. ACCEPTABILITY TEST

1. USER REVIEW OF OUTPUT.
2. JUDGEMENT OF ACCEPTABILITY ACCORDING TO PREDETERMINED STANDARDS.
3. SERVICES PROVIDED BY THE SYSTEM CONFORM TO ORIGINALLY STATED USER REQUIREMENTS.
4. CHANGES IN USER REQUIREMENTS SINCE ORIGINAL STATEMENT.
5. SYSTEM TESTING JUDGED COMPLETE.
6. USER SIGN-OFF ON OUTPUT.
INSTRUCTOR NOTES

- GOALS - WHAT YOU'RE SHOOTING FOR (THE "ENDS")
- REQUIREMENTS - THE MAJOR CONSTRAINTS (THE "WHYS")
- DESIGN - THE BIG PICTURE (THE "WHATS")
- HARDWARE - THE POSSIBLE WAYS THINGS CAN BE DONE (THE "HOWS")
INSTRUCTOR NOTES

ENTERTAIN QUESTIONS, BUT DON'T GET STUCK ON DETAILS. KEEP TALK ON A MORE GENERAL LEVEL RATHER THAN SPECIFIC.

REMARK THAT Ada RELATIONSHIPS WERE HANDLED IN INDIVIDUAL SECTIONS.
<table>
<thead>
<tr>
<th>CHARACTERISTIC</th>
<th>TECHNIQUE</th>
<th>EMPHASIZES</th>
<th>GOAL</th>
<th>APPROPRIATE AUDIENCES</th>
<th>COVERAGE OF THE IMPLEMENTATION PHASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL FLOW</td>
<td>STRUCTURED PROGRAMMING</td>
<td>SIMPLICITY</td>
<td>CODERS</td>
<td>THROUGHOUT</td>
<td></td>
</tr>
<tr>
<td>COMPLEXITY</td>
<td>TIME/SPACE TRADEOFF</td>
<td>EFFICIENCY</td>
<td>CODERS</td>
<td>THROUGHOUT</td>
<td></td>
</tr>
<tr>
<td>HIERARCHICAL</td>
<td>STEP-WISE REFINEMENT</td>
<td>UNDERSTANDABILITY</td>
<td>DESIGNERS CODERS</td>
<td>THROUGHOUT</td>
<td></td>
</tr>
<tr>
<td>DECOMPOSITION</td>
<td>FINITE-STATE MAPS</td>
<td>UNDERSTANDABILITY</td>
<td>ANALYSIS DESIGNERS</td>
<td>CODERS</td>
<td>EARLY</td>
</tr>
<tr>
<td>SYSTEM BEHAVIOR</td>
<td>DECISION TABLES &amp; TREES</td>
<td>SIMPLICITY</td>
<td>CODERS</td>
<td>THROUGHOUT</td>
<td></td>
</tr>
<tr>
<td>DECISION</td>
<td>ASSERTIONS</td>
<td>UNDERSTANDABILITY</td>
<td>CODERS</td>
<td>THROUGHOUT</td>
<td></td>
</tr>
<tr>
<td>LOGIC COMPLEXITY</td>
<td>INVARIANTS &amp; GUARDS</td>
<td>RELIABILITY</td>
<td>CODERS</td>
<td>THROUGHOUT</td>
<td></td>
</tr>
<tr>
<td>LOGIC IMPLICATIONS</td>
<td>TESTING</td>
<td>RELIABILITY</td>
<td>TESTERS</td>
<td>LATE</td>
<td></td>
</tr>
<tr>
<td>REPETITION</td>
<td>INTEGRATION</td>
<td>RELIABILITY</td>
<td>TESTERS</td>
<td>LATE</td>
<td></td>
</tr>
<tr>
<td>COMPLEXITY</td>
<td>REVIEWS</td>
<td>RELIABILITY</td>
<td>EVERYONE</td>
<td>THROUGHOUT</td>
<td></td>
</tr>
<tr>
<td>VERIFICATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VG 778.1

30-2
INSTRUCTOR NOTES

- STRUCTURE - DATA AND ALGORITHMS HAVE THE SIMPLEST STRUCTURES

- INEFFICIENCY - GOES AWAY WITH SMART USE OF TIME/SPACE TRADEOFF

- CORRECTNESS - IS AN IDEAL, NEVER ATTAINABLE

- ASSERTIONS - ARE YOUR ASSUMPTIONS, SO DOCUMENT YOUR THOUGHTS

- GUARDS - MAKE LOOPS RELIABLE

- TEST - INTEGRATE TOP-DOWN = 90%, BOTTOM-UP = 10%, PHASED = 0%

- REVIEW - ALL PRODUCTS, WITH PEERS, AS MUCH AS POSSIBLE
REMINDERES

• DURING IMPLEMENTATION ...
  - STRUCTURE SIMPLY
  - BE AWARE OF BLATANT INEFFICIENCY
  - DON'T INSIST ON CORRECTNESS
  - DOCUMENT ASSERTIONS
  - GUARD AGAINST UNRELIABLE LOOPS
  - TEST IN A RELIABLE MANNER
  - CONSTANTLY REVIEW YOUR WORK

• IMPLEMENTATION TECHNIQUES CAN'T BE EVALUATED USING
  SAME CRITERIA AS FOR ANALYSIS AND DESIGN
THIS LAST TOPIC IS RUN AS A DISCUSSION.

IF THE COURSE HAS BEEN RECEIVED WELL, THEN TRY TO HAVE A LIVELY DISCUSSION.

OTHERWISE, KEEP THE LECTURE SHORT AND TO THE POINT.

FIRST POLL THE CLASS FOR COMMENTS AND GENERAL QUESTIONS. GIVE THEM SOME TIME TO
RESPOND. YOU CAN EVEN GIVE THE CLASS A COUPLE OF MINUTES TO REFLECT ON THE COURSE
AND JOT DOWN COMMENTS AND QUESTIONS. IF THINGS ARE GOING WELL, JUST LET THE
DISCUSSION FLOW. USE UP THE ALLOCATED TIME, THEN FORMALLY "SIGN OFF."

SELECT SLIDES FOR THE INTRODUCTION SECTION (1-4) AND PHASE WRAP-UP SECTIONS IF YOU
NEED TO TALK TO A SLIDE.
INSTRUCTOR NOTES

If the comments and questions of the class don't stimulate enough discussion, ask a few of your own. The following are three questions that are likely to make most of the class start participating:

1. Are there any neglected life cycle phases?

Yes, there are two: installation and maintenance. These phases are still in a state of having a bunch of heuristics, with no methodology. The problems faced in these phases are very project specific, and the sharing of techniques has not been high ("software practice and experience" being one of the only professional forums). So much work is needed to better understand the problems in these areas.
INSTRUCTOR NOTES

2. HOW WELL DO THESE METHODOLOGIES INTEGRATE WITH EACH OTHER?

A BRIEF ANSWER: SOME WELL, SOME NOT SO WELL, SOME NOT AT ALL. INTEGRATION OF TWO METHODOLOGIES IS INFLUENCED BY THE COVERAGE OF THE LIFE CYCLE BY EACH METHODOLOGY, AND BY WHAT IS EMPHASIZED BY EACH METHODOLOGY. LET'S GIVE AN EXAMPLE OF EACH CASE:

SOME WELL

STRUCTURE DESIGN STOPS AT THE MODULE LEVEL; IT IS CONCERNED WITH ARCHITECTURAL DESIGN ISSUES. NSSF DESCRIBES WELL THE LOGIC FLOW OF MODULE. SO, NSSF DIAGRAMS COULD BE DRAWN FOR EACH STRUCTURE CHART BOX. HERE, THE FIT IS EXACT, FOR THE CONTEXT FOR EACH NSSF DIAGRAM IS A BOX ON THE ("BIG PICTURE") STRUCTURE CHART.

ASK THE CLASS FOR METHODOLOGY COMBINATION THEY THINK WOULD WORK WELL TOGETHER.
SADT DIAGRAMS MODEL THE FUNCTIONS AND DATA OF A SYSTEM; SO DO THE BUBBLE CHARTS OF STRUCTURED DESIGN. BUT SADT MODELS EMPHASIZE THE CONSTRAINTS AMONG THE FUNCTIONS OF A SYSTEM, WHILE BUBBLE CHARTS EMPHASIZE DATA FLOW AMONG SYSTEM FUNCTIONS. SO, AN SADT DIAGRAM CANNOT BE CONSIDERED A BUBBLE CHART FROM WHICH A STRUCTURED DESIGN STRUCTURE CHART CAN BE DRAWN. A BUBBLE CHART MUST BE DRAWN FROM THE SADT MODEL, BEFORE A STRUCTURED DESIGN CAN BEGIN. HERE, THE FIT IS NOT EXACT, BUT INTEGRATION CAN BE SUCCESSFUL (THOUGH NOT STRAIGHT-FORWARD).

ASK THE CLASS FOR COMBINATIONS THEY THINK THAT DO NOT FIT.
INSTRUCTOR NOTES

SOME NOT AT ALL

TRYING TO INTEGRATE PARNAS (SCRP) WITH STRUCTURED DESIGN WOULD BE DISASTERIOUS. THE MAJOR REASON BEING WHAT EACH METHODOLOGY EMPHASIZES: PARNAS -- DATA HIDING RESULTS IN PACKAGES, STRUCTURED DESIGN -- DATA FLOWS (TRANSFORMS/TRANSACTION CENTERS) RESULT IN SYSTEM STRUCTURE. COMMON SERVICE PACKAGES ARE DEVELOPED EARLY WITH PARNAS AND LATE WITH STRUCTURED DESIGN. THE TWO METHODS WORK CROSS PURPOSES, SO THEY WOULD BE EXTREMELY DIFFICULT, IF NOT IMPOSSIBLE, TO INTEGRATE.
INSTRUCTOR NOTES

3. HOW DOES ADA FIT WITH THE METHODOLOGIES YOU'VE SEEN?

ASK THE CLASS TO IDENTIFY SOME PROBLEM AREAS IN APPLYING ADA IN THE CONTEXT OF THE METHODOLOGIES WE DISCUSS.

YOU MAY NEED THE ADA RELATIONSHIP SLIDES FROM THE VARIOUS SECTION WRAP-UPS HERE.
INSTRUCTOR NOTES

• NOW IT'S TIME TO SAY "GOODBYE." SO GIVE AN APPROPRIATE FAREWELL, LET EACH INSTRUCTOR HAVE THEIR SAY, LET THE CLASS MAKE ANY PARTING REMARKS.
We would appreciate your comments on this material and would like you to complete this brief questionnaire. The completed questionnaire should be forwarded to the address on the back of this page. Thank you in advance for your time and effort.

1. Your name, company or affiliation, address and phone number.

2. Was the material accurate and technically correct?
   Yes [ ] No [ ]
   Comments: 

3. Were there any typographical errors?
   Yes [ ] No [ ]
   If yes, on what pages?

4. Was the material organized and presented appropriately for your applications?
   Yes [ ] No [ ]
   Comments: 

5. General Comments:
COMMANDER
US ARMY MATERIEL COMMAND
ATTN: AMCDE-SB (OGLESBY)
5001 EISENHOWER AVENUE
ALEXANDRIA, VIRGINIA 22233
DTIC
END
4-86