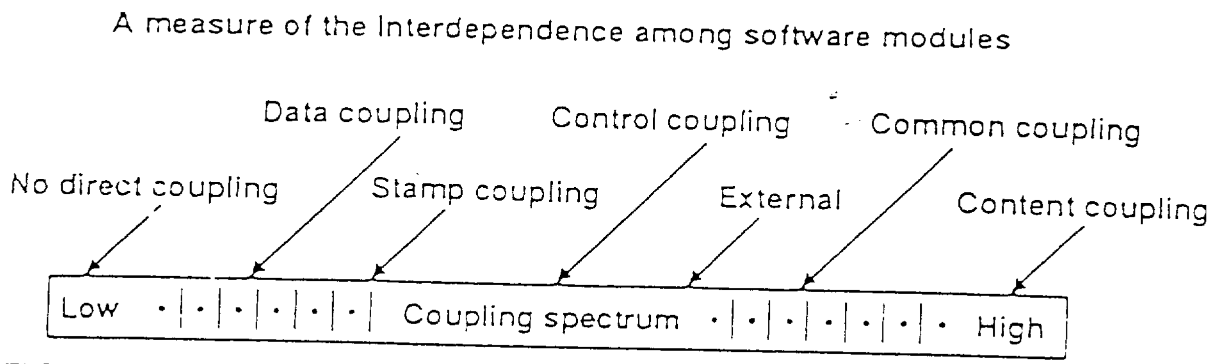


FIGURE 6.12
Coupling spectrum.

- Named collections of declarations.
- Groups of related program units.
- Abstract data types.
- Abstract-state machines.

- *Named collections of declarations*
  Export objects and types.
  Do not export other program units.
- *Groups of related program units*
  Do not export objects and types.
  Export other program units.
- *Abstract data types*
  Export objects and types.
  Export other program units.
  Do not maintain state information in the body.
- *Abstract-state machines*
  Export objects and types.
  Export other program units.
  Maintain state information in the body.

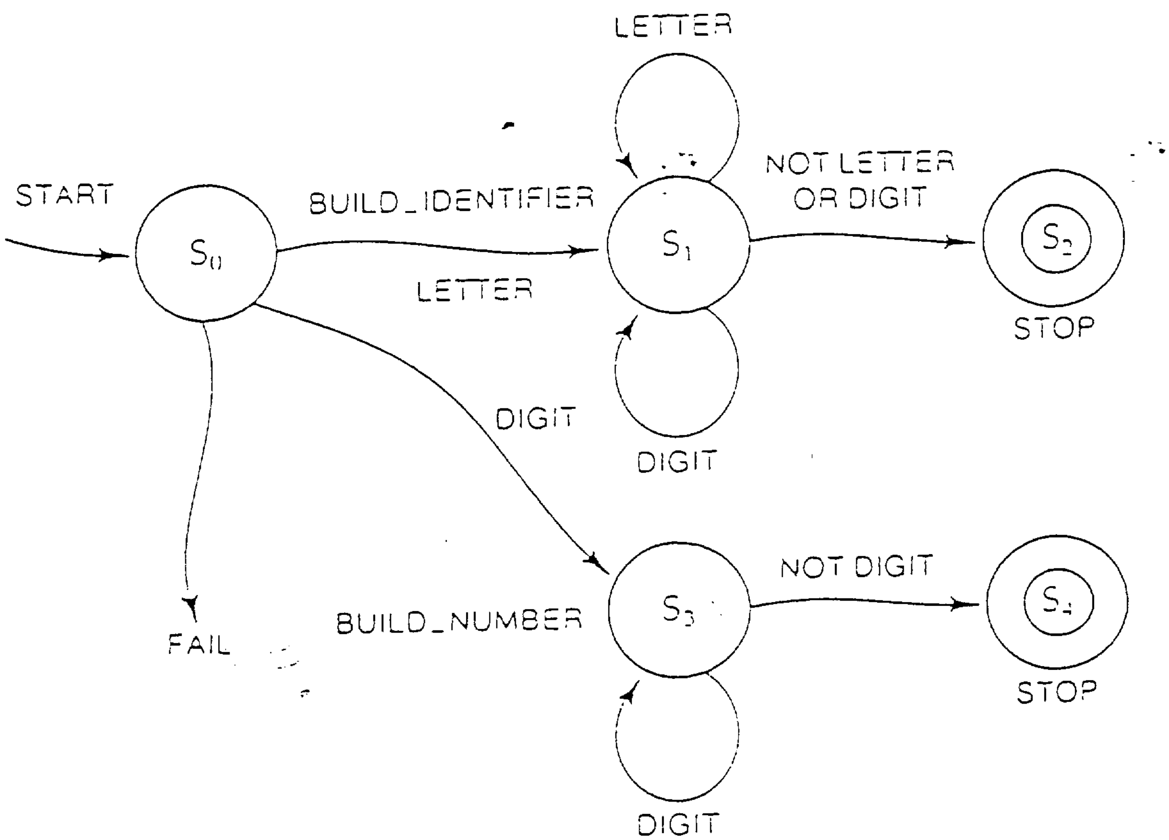
```
package METRIC_EARTH_CONSTANTS is
  EQUATORIAL_RADIUS      : constant := 6_378.145;        -- km
  GRAVITATION_CONSTANT   : constant := 3.986_012e5;      -- km**3/sec**2
  SPEED_UNIT             : constant := 7.905_368_28;     -- km/sec
  TIME_UNIT              : constant := 806.811_874_4;    -- sec
end METRIC_EARTH_CONSTANTS;
```

```
package TRANSCENDENTAL_FUNCTIONS is
  function COS (ANGLE : in FLOAT) return FLOAT;
  function SIN (ANGLE : in FLOAT) return FLOAT;
  function TAN (ANGLE : in FLOAT) return FLOAT;
end TRANSCENDENTAL_FUNCTIONS;
```

```
package QUEUES is
  type QUEUE (SIZE : POSITIVE) is limited private;
  procedure CLEAR (THE_QUEUE         : in out QUEUE);
  procedure ADD    (THE_ITEM          : in     INTEGER;
                    TO_THE_QUEUE      : in out QUEUE);
  procedure REMOVE (THE_ITEM          : out    INTEGER;
                    FROM_THE_QUEUE    : in out QUEUE);
  function LENGTH_OF (THE_QUEUE : in QUEUE) return NATURAL;
private
  type LIST is array (INTEGER range <>) of INTEGER;

  type QUEUE (SIZE : POSITIVE) is
    record
      THE_ITEMS : LIST(1..SIZE);
      THE_BACK  : NATURAL := 0;
    end record;
end QUEUES;
```

```
package LEXICAL_ANALYZER is
  type TOKEN is (NONE, INVALID, IDENTIFIER, NUMBER);
  procedure SET_START_STATE;
  procedure RECEIVE_SYMBOL(C : in CHARACTER);
  function VALUE return TOKEN;
end LEXICAL_ANALYZER;
```



HISTORY OF ADA

- o 1975 DOD HOLWG
- o 1979 BULL HONEYWELL-FRANCE
  - ORIGINAL NAME DOD-1
  - CHANGED TO ADA

WHAT IS ADA

- o PACKAGE - PROGRAM ABSTRACTION
- o OVERLOADING - NAME MANAGEMENT
- o SEPARATE COMPILATION
- o SOFTWARE ENGINEERING
  - BOTTOM UP - PACKAGE
  - TOP DOWN - STUBS
- o PROGRAMMING IN ADA
- o REAL TIME

```
package STRUCTURE is
        --Package specification
        --This is the part of the package that is visible to the user and indicates to
        --the user the resources that are available in the package. The package
        --specification should be written prior to the programs that use the
        --package. The specification may include data type definitions, data object
        --declarations, and subprogram specifications. Subprogram specifications
        --indicate the interface mechanism to subprograms in the package that are
        --available to the user.
private
        --The private part of the package specification is optional. It is useful when
        --the data type names must be made visible to the user but the internal
        --structure of the data type remains hidden from the user. Operations on
        --private data types may be made available to the user in the package but
        --the user does not have access to their internal representation.
end STRUCTURE;


package body STRUCTURE is
        --Declaration of local variables and types. These are not known or usable
        --outside of the package body. Declaration of subprograms not visible
        --outside of the package.

        --Implementation of subprograms defined in the visible part of the
        --specification. The subprograms may be procedures and functions.
        --Implementation of auxiliary subprograms needed to implement the visible
        --subprograms.
end STRUCTURE;
```

```
package LINEAR_SYSTEMS is
        MAXSIZE: constant INTEGER:=50;
        subtype INDEX is INTEGER range 1..MAXSIZE;
        type VECTOR is array (INDEX) of FLOAT;
        type MATRIX is array (INDEX, INDEX) of FLOAT;
        procedure LU_FACTOR(N: in INTEGER; A: in out MATRIX);
        procedure SOLVE(N: in INTEGER; A: in MATRIX; C: in VECTOR; X: out
        VECTOR);
        procedure MATRIX_INVERSE(N: in INTEGER; A: in out MATRIX; B: out
        MATRIX);
end LINEAR_SYSTEMS;
```

```
with LINEAR_SYSTEMS;
procedure SIMULTANEOUS_EQUATIONS is
        --Declaration of types and variables
begin
        --Sequence of statements that create, say, a 10 × 10 matrix, A
        LINEAR_SYSTEMS.MATRIX_INVERSE(10,A,B);
        --The inverse of A is B
        --Sequence of statements that perform desired operations
end SIMULTANEOUS_EQUATIONS;
```

```
with LINEAR_SYSTEMS; use LINEAR_SYSTEMS;
procedure SIMULTANEOUS_EQUATIONS is
        --Declaration of types and variables
begin
        --Sequence of statements that create, say, a 10 × 10 matrix, A
        MATRIX_INVERSE(10,A,B);
        --The inverse of A is B
        --Sequence of statements that perform desired operations
```

```
declare --Defines a block of code
        P: INTEGER;
        Q: FLOAT;
        procedure OVER_LOAD(X: INTEGER) is
        begin
                --Sequence of statements
        end OVER_LOAD;
        procedure OVER_LOAD(X: FLOAT) is
        begin
                --Sequence of statements
        end OVER_LOAD;
        begin
                --Sequence of statements
                OVER_LOAD(P);   --Calls the version of OVER_LOAD with integer
                                --parameter; the first version
                OVER_LOAD(Q);   --Calls the version of OVER_LOAD with float parameter;
                                --the second version
        end;
```

```
package COMPLEX_NUMBERS is
        type COMPLEX is
                record
                        REAL: FLOAT;
                        IMAGINARY: FLOAT;
                end record;

        function "+"(X,Y: COMPLEX) return COMPLEX;

        function: "*"(X,Y: COMPLEX) return COMPLEX;

                --Other functions or subprograms may also be specified.

end COMPLEX_NUMBERS;
```

```
package body TOP_DOWN is
        --Declarations
procedure NOT_DONE is separate;
        --Package implementation
end TOP_DOWN;
```

```
separate (TOP_DOWN)
procedure NOT_DONE is
        --Declarations
begin
        --Statements
end NOT_DONE;
```

```
package COMPLEX is

   type NUMBER is ...

   -- abstract operations for NUMBER objects

end COMPLEX;


with COMPLEX;
procedure MAIN is
   --
   MY_NUMBER : COMPLEX.NUMBER;
   --
begin
   -- body of MAIN
end MAIN;
```

```
task MAILBOX is
        entry SEND(MAIL: in MESSAGE);
        entry RECEIVE(MAIL: out MESSAGE);
end;

task body MAILBOX is
        BOX: MESSAGE;
begin
        loop
                accept SEND(MAIL: in MESSAGE) do
                        BOX:=MAIL;
                end;
                accept RECEIVE(MAIL: out MESSAGE) do
                        MAIL:=BOX;
                end;
        end loop;
end MAILBOX;
```

PROGRAMMING IN ADA

- o PREDEFINED DATA TYPES
- o STRONGLY TYPED
- o PARAMETER PASSING-I O I/O
- o PROGRAM UNITS
  - BLOCK
  - SUBPROGRAM
  - TASK
  - PACKAGE
- o SEQUENTIAL
- o CONDITIONAL
- o ITERATIVE
- o EXCEPTIONS
- o GENERICS

```
procedure PROCESS_TEMPERATURE is
  TEMPERATURE  : FLOAT;
  OVER_TEMP    : exception;
begin
  loop
    GET(TEMPERATURE);
    SCALE(TEMPERATURE);
    if TEMPERATURE > LIMIT then
      raise OVER_TEMP;
    end if;
  end loop;
exception      -- an exception handler
  when OVER_TEMP =>
    -- sequence of statements
end PROCESS_TEMPERATURE;
```

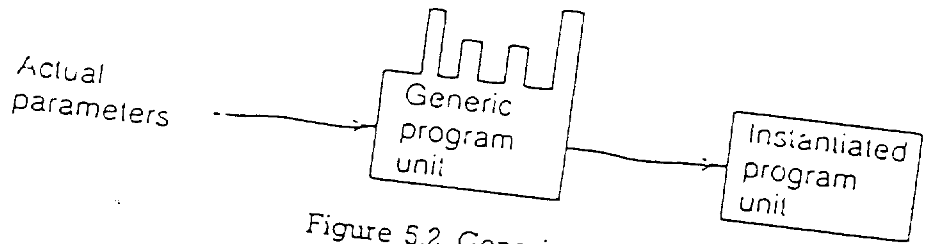


Figure 5.2 Generic Instantiation

```
generic     -- the generic part
  LIMIT : NATURAL;
  type DATA is private;
package STACKS is

  type STACK is private;

  procedure PUSH (ELEMENT : in  DATA; ON   : in out STACK);
  procedure POP  (ELEMENT : out DATA; FROM : in out STACK);

private
  type LIST is array (1..LIMIT) of DATA;
  type STACK is
    record
      STRUCTURE : LIST;
      TOP       : INTEGER range 1..LIMIT: = 1;
    end record;
end STACKS;
```

```
package INTEGER_STACK is new STACKS (100, INTEGER);
package FLOAT_STACK   is new STACKS (LIMIT => 300, DATA => FLOAT
```

V

RULES FOR WRITING EFFICIENT CODE

MODIFYING DATA STRUCTURES

- o TRADING SPACE FOR TIME
- o TRADING TIME FOR SPACE

MODIFYING CODE

- o LOOPS
- o LOGIC
- o PROCEDURES
- o EXPRESSIONS

TRADING SPACE FOR TIME

- DATA STRUCTURE AUGMENTATION
- STORE PRECOMPUTED RESULTS
- CACHING
- LAZY EVALUATION

TRADING TIME FOR SPACE

- o PACKING
- o INTERPRETERS

LOOPS

- o CODE MOTION OUT OF LOOP
- o COMBINING TESTS
- o LOOP UNROLLING
- o TRANSFER DRIVEN LOOP UNROLLING
- o UNCONDITIONAL BRANCH REMOVAL
- o LOOP FUSION

LOGIC

- EXPLOIT ALGEBRAIC STRUCTURES
- SHORT CIRCUIT MONOTONE FUNCTIONS
- REORDERING TESTS
- PRECOMPUTE LOGICAL FUNCTIONS
- BOOLEAN VARIABLE ELIMINATION

PROCEDURES

- o COLLAPSING PROCEDURE HIERARCHIES
- o EXPLOIT COMMON CASES
- o COROUTINES
- o TRANSFORMATIONS ON RECURSIVE PROCEDURES
- o PARALLELISM

EXPRESSIONS

- o COMPILE TIME INTIALIZATIONS
- o EXPLOIT ALGEBRAIC IDENTITIES
- o COMMON SUBEXPRESSION ELIMINATION
- o PAIRING COMPUTATIONS
- o EXPLOIT WORD PARALLELISM

```
procedure ApproxTSTour;
  var I, J: PtPtr;
    Visited: array [PtPtr] of boolean;
    ThisPt, ClosePt: PtPtr;
    CloseDist: real;

  begin
  (* Initialize unvisited points *)
  for I := 1 to NumPts do
    Visited[I] := false;

  (* Choose NumPts as starting point *)
  ThisPt := NumPts;
  Visited[NumPts] := true;
  writeln('First city is ', NumPts);

  (* Main loop of nearest neighbor heuristic *)
  for I := 2 to NumPts do
    begin
  (* Find nearest unvisited point to ThisPt *)
    CloseDist := maxreal;
    for J := 1 to NumPts do
      if not Visited[J] then
        if Dist(ThisPt, J) < CloseDist then
          begin
          CloseDist := Dist(ThisPt, J);
          ClosePt := J
          end;
    (* Report closest point *)
    writeln('Move from', ThisPt, 'to', ClosePt);
    Visited[ClosePt] := true;
    ThisPt := ClosePt
    end;

(* Finish tour by returning to start *)
writeln('Move from', ThisPt, 'to', NumPts)
end;
```

Fragment A1. Original code.

```
procedure ApproxTSTour;
  var
    I: PtPtr;
    UnVis: array [PtPtr] of PtPtr;
    ThisPt, HighPt, ClosePt, J: PtPtr;
    CloseDist, ThisDist: real;
  procedure SwapUnVis(I, J: PtPtr);
    var Temp: PtPtr;
    begin
    Temp := UnVis[I];
    UnVis[I] := UnVis[J];
    UnVis[J] := Temp
    end;

  begin
  (* Initialize unvisited points *)
  for I := 1 to NumPts do
  UnVis[I] := I;
  (* Choose NumPts as starting point *)
  ThisPt := UnVis[NumPts];
  HighPt := NumPts-1;
  (* Main loop of nearest neighbor tour *)
  while HighPt > 0 do
    begin
    (* Find nearest unvisited point to ThisPt
    CloseDist := maxreal;
    for I := 1 to HighPt do
      begin
      ThisDist := DistSqrd(UnVis[I],ThisPt);
      if ThisDist < CloseDist then
        begin
        ClosePt := I;
        CloseDist := ThisDist
        end
      end;
    (* Report this point *)
    ThisPt := UnVis[ClosePt];
    SwapUnVis(ClosePt,HighPt);
    HighPt := HighPt-1
    end
  end;
```

Fragment A4. Convert boolean array to pointer array.

```
function Fib(N: integer): integer;
  var A, B, C, I: integer;
  begin
  if N<1 or N>MaxFib then return 0;
  if N<=2 then return 1;
  A := 1;  B := 1;
  for I := 3 to N do
    begin
    C := A + B;
    A := B;
    B := C
    end;
  return C
  end;
```

Fragment B1. Fibonacci numbers.

```
I := 1;
while I <= N cand X[I] <> T do
    I := I+1;
if I <= N then
    (* Successful search: T = X[I] *)
    Found := true
  else
    (* Unsuccessful search: T is not in X[1..N] *)
    Found := false
```

Fragment D1. Sequential search in an unsorted table.

```
X[N+1] := T;
I := 1;
while X[I] <> T do
    I := I+1;
if I <= N then
    Found := true
  else
    Found := false
```

Fragment D2. Add sentinel to end of table.

```
function Fib(N: integer): integer;
  var A, B, C, I: integer;
  begin
  if N<1 or N>MaxFib then return 0;
  if N<=2 then return 1;
  A := 1;  B := 1;
  for I := 3 to N do
    begin
    C := A + B;
    A := B;
    B := C
    end;
  return C
  end;
```

Fragment B1. Fibonacci numbers.

```
var FibVec: array [1..MaxFib] of integer;
```

```
function Fib(N:integer):integer;
  var A,B,I: integer;
  begin
  if N < 1 or N > MaxFib then return 0;
  if N <= 2 then return 1;
  A := 1; B := 1;
  for I := 1 to (N div 2) - 1 do
    begin
    A := A+B;
    B := B+A
    end;
  if odd(N) then
    B := B+A;
  return B
  end;
```

Fragment B3. Loop-unrolled Fibonacci numbers.

```
Sum := 0;
for I := 1 to 10 do
  Sum := Sum + X[I]
```

Fragment F1. Compute the sum of X[1..10].

```
Sum := X[1] + X[2] + X[3] + X[4] + X[5]
     + X[6] + X[7] + X[8] + X[9] + X[10]
```

Fragment F2. Unrolled sum of X[1..10].