6. Preparation for Practice: Translation to Programming Languages

Aims

- Mapping of architectural concepts to mainstream programming languages
- Translation can be done by tools as part of an integrated Software Development Environment

  Architecture → Module Code Frames

  then using Programming Environment or

  Architecture Tool – Integrator – Programming Tool
- Do translation by hand

  thereby extending architectural knowledge

Contents of Chapter 6

Level of Programming Language

- High-level languages with rich architectural concepts (e.g. Ada 95):
  translation easy, but Programming Language ≠ Architectural Language

  Low-level (FORTRAN 66, C, Assembler):
  translation difficult but possible
- First question: Has programming language module construct ?
  i.e. aggregated interface, hidden bodies

  else aggregate mentally
  corresponding layout and comments
  discipline
- Second question: Has language import relations
  else simulation as above
- Third question: Has language consistency constraints ?
  else discipline

Clarifying Topics

- Module construct of programming language:
  aggregated interface + body
  module types f, ado, adt are “semantical“ uses
- Import construct of programming language
  import relation “types“ are “semantical“ uses
- Translation never impossible
  but more or less difficult
- However: range of mapping depends on language
  not possible to map recursion on FORTRAN 66
  not possible to map OO on C
  So we have to regard “restricted“ architectures

”Classes” of Programming Languages

Regard in this chapter 3 classes:

- languages without module construct
  flat / no hierarchies, arbitray relations, without checking consistency,
  mostly static storage allocation
  representatives: FORTRAN 6x, C, Cobol 6x / 7x, Basic, Assembler
- block-structured languages without module construct
  locality by nesting, visibility / scope, no independent compilation
  dynamic storage allocation (stack, heap)
  representatives: Standard-Pascal, Algol 60, Algol 68 etc.
- new programming languages with module concepts
  independent compilation, more or less checking between units, multi-paradigmatic
  representatives: Ada 83 and 95, Modula-2 / Modula-3, Turbo-Pascal, C++ etc.

Translating to FORTRAN IV

- FORTRAN IV as representative of languages with independent units, independently compileable
- Units are not modules but units from which a module body is composed of
- These languages have subprograms working a global data space (in FTN separable to different COMMOMs)
- Now: Translation of f, ado, adt modules
  and module relations locality / general usability

Interface f Module

```
C   ********************************************************************************
C   * functional module Graph_of_Functions is                                      *
C   *      Input data in parameter list                                            *
C   *      Output datum is the plotter file                                        *
C   *                                                                              *
C   *      procedure POLYLN (X, Y: inARRAY_RT;                                     *
C   *                                XTEXT, YTEXT, UETEXT: in STRING);             *
C   *      procedure INTLN (...);                                                  *
C   *      ...                                                                     *
C   *      Semantics of Graph_of_Functions: ...                                    *
C   *      ...                                                                     *
C   * end Graph_of_Functions;                                                      *
C   *------------------------------------------------------------------------------*
```

So, interface is just imagination

Body f Module

```
C       module body Graph_of_Functions is ----------------------------------------------------*
C
        SUBROUTINE POLYLN (X, Y, XTEXT, YTEXT, UETEXT)
                REAL X(100), Y(100)
                INTEGER XTEXT (20), YTEXT(20), UETEXT (20)
                ...
                RETURN
        END
C                                                                       only units of
        SUBROUTINE INTLN (X, Y, XTEXT, YTEXT, UETEXT)                   module body
                REAL X(100), Y(100)                                     are FORTRAN
                INTEGER XTEXT (20), YTEXT(20), UETEXT (20)
                ...
                RETURN
        END

C       ...     further “local“ subroutines: sealing, plotting axes etc.

C       end Graph_of_Functions;                                                 *
C       ***************************************************************************************
```

Interface ado Module

```
C   ********************************************************************************
C   abstract data object module INTEGER_STACK is                                   *
C   *       procedure PUSH (X: in INTEGER);                                        *
C   *       procedure POP;                                                         *
C   *       function READ_TOP return INTEGER: - -- - RDTOP                         *
C   *       function IS_EMPTY return BOOLEAN; - -- - ISEMTY                        *
C   *       function IS_FULL return BOOLEAN;     - -- - ISFULL                     *
C   *       ...                                                                    *
C   *       Semantics of operations                                                *
C   *       ...                                                                    *
C   end INTEGER_STACK;                                                             *
C   *-------------------------------------------------------------------------------*
```

Body ado Module (1)

```
C       module body INTEGER_STACK is ----------------------------------------------------*
        SUBROUTINE PUSH (ELEMENT)
                INTEGER STACK (100), POINTR, ELEMNT
                COMMON / STDATA / STACK, POINTR
                ...
                RETURN
        END
C
        SUBROUTINE POP
                INTEGER STACK (100), POINTR
                COMMON / STDATA / STACK, POINTR
                ...
                RETURN
        END
```

Body ado Module (2)

```
C
      INTEGER FUNCTION RDTOP
            INTEGER STACK (100), POINTR
            COMMON / STDATA / STACK, POINTR
            ...
            RETURN
      END
C
      ...
      BLOCKDATA
            INTEGER STACK (100), POINTR
            COMMON / STDATA / STACK, POINTR
            ...
      END
C     end INTEGER_STACK; ************************************************
```

Mapping Locality

FORTRAN programs are an unstructured heap of units (which compose module bodies)

⇨ no module relation contains, local usability, general usability

⇨ simulation by layout, comments, and discipline

```
C     ****************************************************************************************
C     * functional module USERINPUT is                                                       *
C     *         procedure INPUT (...);                                                       *
C     *                                                                                      *
C     *         The semantics of interface operation is: ...                                 *
C     *         ...                                                                          *
C     *                                                                                      *
C     * end USERINPUT; ------------------------------------------------------------------------*
C     *                                                                                      *
C     * module body USERINPUT is --------------------------------------------------------------*
C     *         local import from CHECKINPUT                                                 *
C     *                         using CHECKSTRING, CHECKNUM;                                 *
C     *         Realization of above operations:                                             *
              *         ...                                                                  *
C     * end USERINPUT;                                                                       *
C     *--------------------------------------------------------------------------------------*
```

the locally importing module

(not recursive situations !)

The Local Module

```
C    ********************************************************************************* *
C    functional module CHECKINPUT is                                                   *
C    *      is contained in USERINPUT;                                                  *
C    *      procedure CHECKSTRING (...);                                                *
C    *      procedure CHECKNUM (...);                                                   *
C    *      ...                                                                         *
C    *      The semantics of interface operations: ...                                  *
C    *      ...
C    end CHECKINPUT; ------------------------------------------------------------------ *
C    * module body CHECKINPUT is ------------------------------------------------------ *
C    *                                                                                  *
C           ...
C    end CHECKINPUT;                                                                   *
C    *---------------------------------------------------------------------------------- *
```

General Usability & Other Concepts

- general usability nearer to FORTRAN IV concepts
- analogous to above
  general import clause at the interface or
  body of using component
- subsystems by two-fold simulation
- genericity only by macro mechanisms
- OO not available: cannot easily be simulated
- recursion not available

Translation to C

- every module to 2 files
  interface: header file
  body: implementation file
- No module concept but rigid help on
  program file management
- simulation (layouting, comments, discipline) similar to FORTRAN
- subsystems via double simulation
- genericity only by macro
- OO not available
- concentrate on ado mudules, f analogous
  adt modules analogous, better with classes from C++

Ado Interface in C

```
// abstract data object module Buffer ***********************************************

# ifndef __ INC_BUFFER_H __

# define__ INC_BUFFER_H __


void enqueue (item x);

void deque (item &x);

bool nonempty ();

bool nonfull ();


// item is the type of buffer elements.

// semantics: enqueue adds an element at the backend if buffer is nonfull,

// ...

// ---------------------------------------------------------------------------------
```

Ado Body in C

```
# include “Buffer.h“
//general import Virt_array using all
// abstract data object module body Buffer is ----------------------------------------------------
// realization: circular storage using another ado module Virt_Array
    //administration info for the data within Virt_Array container
    static int In = 0
    static int Out = 0
    static int n = 0
    static bool nempty = false
    static bool nfull = true
    // static data scope is restricted to the implementation file

    void enqueue (item x) {
       if (n < N) {
          storeVA (In, x); In = (In +1) % N;
          n ++; nfull = (n < N); nempty = true; }
       else { ...}
    ...
// ***************************************************************************************
```

The Used Component Ado

```
// abstract data object module Virt_Array ************************************

# ifndef __ INC_VIRTARRAY_H __

# define __ INC_VIRTARRAY_H __


void storeVA (int loc, item x);

void readVA (int loc, item &x);

...


// ------------------------------------------------------------------------------
```

Consistency Conditions (Discussion for FORTRAN and C)

- All consistency constraints of chapter 4 by discipline:
  - for structure and usability relations (graphic level)
  - for composing export interfaces to a module of a certain kind
  - for having consistent import interfaces
  - for consistency export – import
  - for consistency import – body
- Furthermore as “modules“ are “simulated“
  - every module is of one above kinds
  - bodies are hidden:

    local procedures not used outside

    encapsulated data structures not directly read or changed

    local type declarations not used outside

Translation to Standard Pascal

- Translation similar
  - » aggregate mentally
  - » compose textually, layout, and use comments
  - » apply discipline
- Thereby translation of a class of programming languages
  - » block-structured
  - » without modules
  - » locality (nesting) the only vehicle to structure programms
  - » other representatives Algol 60, 68, etc.
- Translation simpler , as Pascal is 10 years younger ? No!
- Translation of module “types“ similar,

  have some translation experience

  ⇨ discuss only adt translation

Specific Difficulty: Order of Declarations

```
anywhere in constant declarations part
const STACKMAX = ... ;                          (* only for internal use in module Stack *)
...
anywhere in type declaration part
type StackT =                                   (* only for internal use in module Stack *)
      record
            SPACE : array [ 1 ... STACKMAX ] of item
            INDEX : integer
      end;
anywhere in procedure declaration part: interface and body of Stack
(*****************************************************************************************
data type module Stack is
      type StackT is private
      procedure PUSH (St: in out StackT; EC: in Item);
        ...
 end Stack; ------------------------------------------------------------------------------- *)
(*    module body Stack is ------------------------------------------------------------------ *)
      procedure PUSH (St: in out StackT; EC: in Item);
      ...
      begin ...
      end;
      ...
(* end Stack; ***************************************************************************)
```

Import Translation of Relations to Standard Pascal

- Locality Translation Idea:
  contains-relation by nesting
  local usability by scope / visibility rules

  } not possible Pascal has no nesting for modules

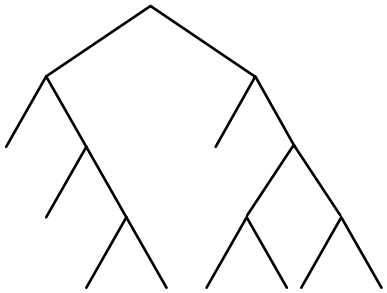
  ⇨ locality expressed only by comments

- General Usability:
  Pascal has no import relations
  ⇨ general usability by comments

Resulting Program Structure

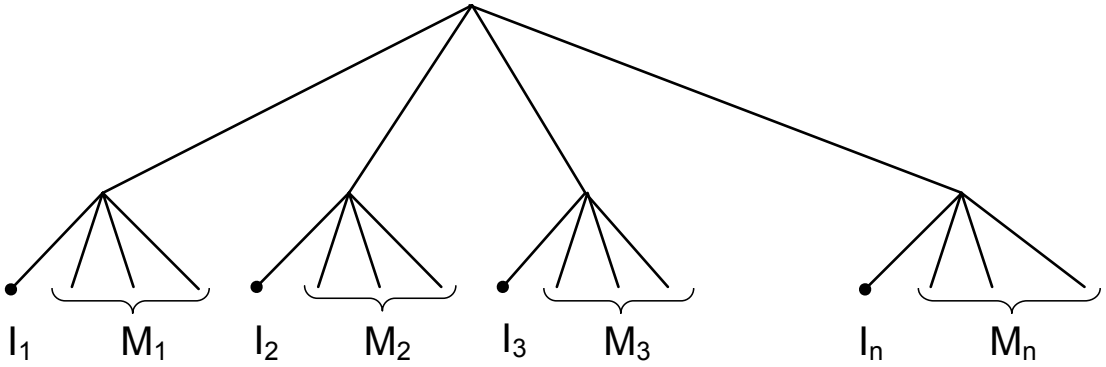
- Where are the “modules“ connected by local and general usability:
  In the upmost block
- Structure of a translated program

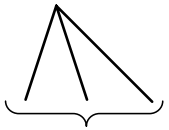
typical Pascal
program structure



nested
procedures

structure of program
translated from an arch. diagram





Mi

units of a
module body

Ii interface by
comments

+ mutual module relations by comment clause

Example Ada

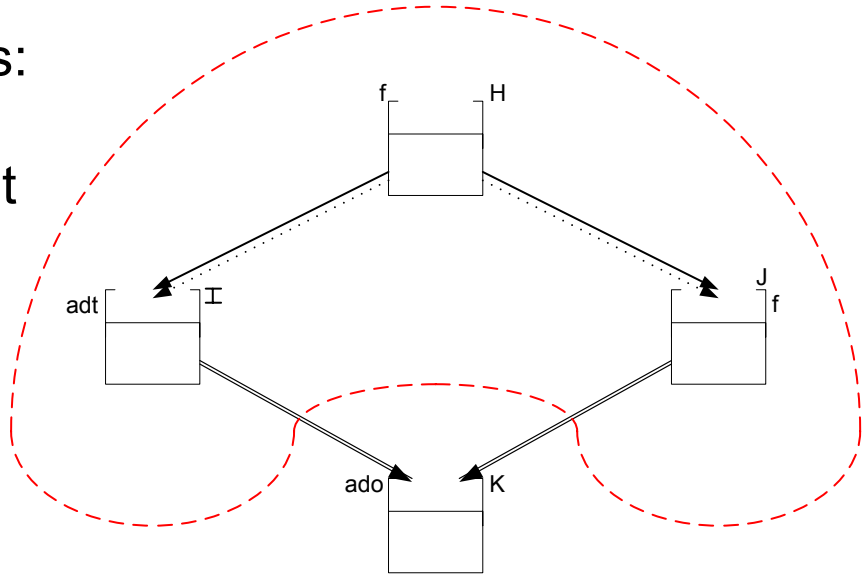
- Class of new programming languages with modules, imports, multiparadigmatic language
- Representatives:
  Ada 83, Modula-2, Pascal dialects
- In addition with OO: C++, Modula-3, Ada 95
- Language for textual architecture specifications in this lecture
  for export interfaces, imports
  was already Ada 83 or Modula-2 like
- Discussing translation of module types we again restrict ourselves to adt modules in following example

Adt Module in Ada 83

```
-- data type ************************************************************************************************************************************************
package ITEM_STACK_STENCIL is                                                                                                                                --
        type ITEM_STACK_TYPE is private;                                                                                                                     --
        procedure INITIALIZE (ST: in out ITEM_STACK_TYPE);                                                                                                   --
        procedure PUSH (EL: in ITEM_TYPE; ST: in out ITEM_STACK_TYPE);                                                                                       --
        procedure POP (ST: in out ITEM_STACK_TYPE);                                                                                                          --
        function READ_TOP (ST: in ITEM_STACK_TYPE) return ITEM_TYPE;                                                                                         --
        function IS_EMPTY (ST: in ITEM_STACK_TYPE) return BOOLEAN;                                                                                           --
        function IS_FULL (ST: in ITEM_STACK_TYPE) return BOOLEAN;                                                                                            --
        ST_UNDERFLOW, ST_OVERFLOW; ST_NOT_INITIALIZED: exception;                                                                                            --
        -- semantics: ...                                                                                                                                    --
        private                                                                                                                                              --
            SIZE: constant INTEGER := 100;                                                                                                                   --
            type SPACE_T is array (1..SIZE) of ITEM_TYPE;                                                                                                    --
            type ITEM_STACK_TYPE is                                                                                                                          --
                record                                                                                                                                       --
                    SPACE: SPACE_T;                                                                                                                          --
                    INDEX: INTEGER range 0..SIZE := 0;                                                                                                       --
            end record;                                                                                                                                      --
end ITEM_STACK_STENCIL;  ---------------------------------------------------------------------------------------------------------------------------------------------
package body ITEM_STACK_STENCIL is ------------------------------------------------------------------------------------------------------------------------------
        -- implementation of above interface  ...                                                                                                            --
end ITEM_STACK_STENCIL; -- ********************************************************************************************************************************
```

Transformation of Locality

- contains relation by nesting:
  in the declaration part of a module body another one is nested
- big text units result:
  we have bodies separately: body stubs and separate subunits
  (to be developed, compiled)
  export interface as comments for subunits
  is – contained clause as comment
- nesting ⇨ scope / visibility rules:
  potential local usability
  local import clause as comment



The Example 1

```
-- functional ***************************************************************************************************************************************************
package H is                                                                                                                                                    --
      ...                                                                                                                                                       --
end H;  ------------------------------------------------------------------------------------------------------------------------------------------------------
package body H  is ---------------------------------------------------------------------------------------------------------------------------------------------
      -- local import from I using ...;                                                                                                                        --
      -- local import from J using ...;                                                                                                                        --
      -- abstract data type  ***************************************************************************                                                    --
      package I is                                                                  --                                                                      --
          ...                                                                       --                                                                      --
      end I;  ---------------------------------------------------------------------------------------------                                                  --
      package body I  is separate;  --*****************************************************************                                                    --
       -- functional  ********************************************************************************                                                     --
      package J is                                                                  --                                                                      --
          ...                                                                       --                                                                      --
      end J; ----------------------------------------------------------------------------------------------                                                  --
      package body J  is separate;  --*****************************************************************                                                    --
      ...                                                                                                                                                   --
end H;  --*************************************************************************************************************************************************--

separate (H);*************************************************************************************************************************************************
-- abstract data type package I is                                                                                                                          --
--      is contained in H;                                                                                                                                  --
--      ...                                                  export interface as comment                                                                    --
-- end I;  ---------------------------------------------------------------------------------------------------------------------------------------------------
package body I  is ---------------------------------------------------------------------------------------------------------------------------------------------
         ...                                                                                                                                                --
end I;  --*************************************************************************************************************************************************--

--analogously for J                                                                                                                                          --
```

Translation of General Usability

- general components: library units
- general usability: Ada import clauses
- import only of specific resources: comment

The Example 2

```
-- abstract data object ********************************************************************************************************************************
package K is                                                                                                                                          --
      ...                                                                                                                                             --
end K;  ---------------------------------------------------------------------------------------------------------------------------------------------
package body K  is  ---------------------------------------------------------------------------------------------------------------------------------
      ...                                                                                                                                             --
end K;  --********************************************************************************************************************************************* --
```

```
-- H as above                                                                                                                                         --
```

```
with K; use K; separate (H) --********************************************************************************************************************** --
-- abstract data type package I is                                                                                                                    --
--      is contained in H;                                                                                                                            --
--      ...                                                  export interface as comment                                                          --
-- end I;  ---------------------------------------------------------------------------------------------------------------------------------------------
package body I  is ----------------------------------------------------------------------------------------------------------------------------------
      -- general import from K using ...;                                                                                                           --
        ...                                                                                                                                           --
end I;  --********************************************************************************************************************************************* --
```

```
with K; use K; separate (H) --********************************************************************************************************************** --
-- functional package J is                                                                                                                            --
--      is contained in H;                                                                                                                            --
--      ...                                                  export interface as comment                                                          --
-- end J;  ---------------------------------------------------------------------------------------------------------------------------------------------
package body J is ----------------------------------------------------------------------------------------------------------------------------------
      -- general import from K using ...;                                                                                                           --
        ...                                                                                                                                           --
end J;  --********************************************************************************************************************************************* --
```

Ada Translation Contd.

- consistency rules:
  most of them by discipline
- so Ada 83 ≠ Architectural Language
  Architectural Language conceptually much richer
- above discussion assumes:
  locality → nesting and stubs
- if locality is mapped on library units:
  even more simulation and discipline

Subsystems as Aggregated Packages of Ada 83

```
-- data object subsystem *****************************************************************************************************************
   package Entry_Coll is                                                                                                               --
--          data type **********************************************************************************************                 --
            package Entry  is                                                                                    --                  --
                      type Entry_Den_Type is private;                                                            --                  --
                      procedure Create_and_Init  (El: out Entry_Den_Type);                                       --                  --
                      ...                                                                                        --                  --
                      -- semantics: ...                                                                          --                  --
             private                                                                                             --                  --
                      type Entry_Den_Type is ...                                                                 --                  --
            end Entry; --************************************************************************************                        --
--          data object *********************************************************************************************                 --
            package Coll  is                                                                                     --                  --
                      procedure Store (El: in Entry_Den_Type);                                                   --                  --
                      ...                                                                                        --                  --
                      -- semantics: ...                                                                          --                  --
             end Coll; --*****************************************************************************************                    --
  end Entry_Coll; ---------------------------------------------------------------------------------------------------------------   --

  package body Entry_Coll is --------------------------------------------------------------------------------------------------------   --
            ...                                                                                                  --
            package body Entry  is --*****************************************************************************      --
                 -- Translation of Entry as usual                                                                --                  --
                 -- Body can be developed separately (here only stub)                                            --                  --
                 ...                                                                                             --                  --
            end Entry; --*****************************************************************************************      --
            package body Coll  is *****************************************************************************      --
                  -- Translation of Collection as usual                                                          --                  --
                  -- Body can be developed separately (here only stub)                                           --                  --
                  ...                                                                                            --                  --
            end Coll; --******************************************************************************************      --
   begin                                                                                                                              --
       ...                                                                                                                            --
   end Entry_Coll;  --***************************************************************************************************************   --
```

Subsystems as Public Library Units of Ada 95

- Subsystems interfaces are aggregated
- Any extension of the interface ⇨ all clients have to be recompiled
- Hierarchical libraries of Ada 95 to handle this problem:
  extension: no recompilation is necessary if client only uses the old interface
- Mechanism of public child library units can be used
  - » for extending an interface
  - » especially, for extending inheritance hierarchies:

    inheritance hierarchy = subsystem

Translation of Genericity

- simple genericity
  - discrete types
  - simple types with operations
  - ⇨ generic clauses of Ada 83
  - generic instantiation of generic units

restrictions:
no general arch.
within body of
generic unit

- “complex“ types: main application of genericity
  - complex type with operations ⇨ adt

  - ⇨ formal generic packages of Ada 95
    - define the properties which have to be fulfilled

Translation of OO

- most languages of this class have OO
  Modula-3, Ada 95, C++

- in Ada 95 inheritance is orthogonal to modularization
  ⟹ inheritance hierarchies in the interface of a package:
  not good if inheritance changes

- better one type (class) = one package
  organization of inheritance structure by public child libraries

What We Have Learned

- Translation discussion valuable:
  - » can architecture language apply to all relevant progr. languages
  - » have deepened understanding of arch. language
- Architeture Language ≠ modern Programming Language
  - » Types of components
  - » Types of relations
  - » Consistency constraints
- Have learned about Progr. Language Classes
- Translation can be mechanized by tools