4. The Notation: An Architecture Description Language

Aims

- Components
  - » Modules for functional and data abstraction
  - » Subsystems as big components
- Relations
  - » Locality, Layering, Inheritance with corresponding usabilities
- Consistency conditions
  - » Language rules, methodology rules
- Language(s)
  - » Graphical for overview
  - » Textual for details

  } influenced by Modula-3, Ada 95 (C++)
- Bottom-up explanation starting with modules
- Examples on the level of components, two components with relation

Contents of Chapter 4

big main chapter of this lecture

What is a Module? (Result of a Students' Brainstorming)

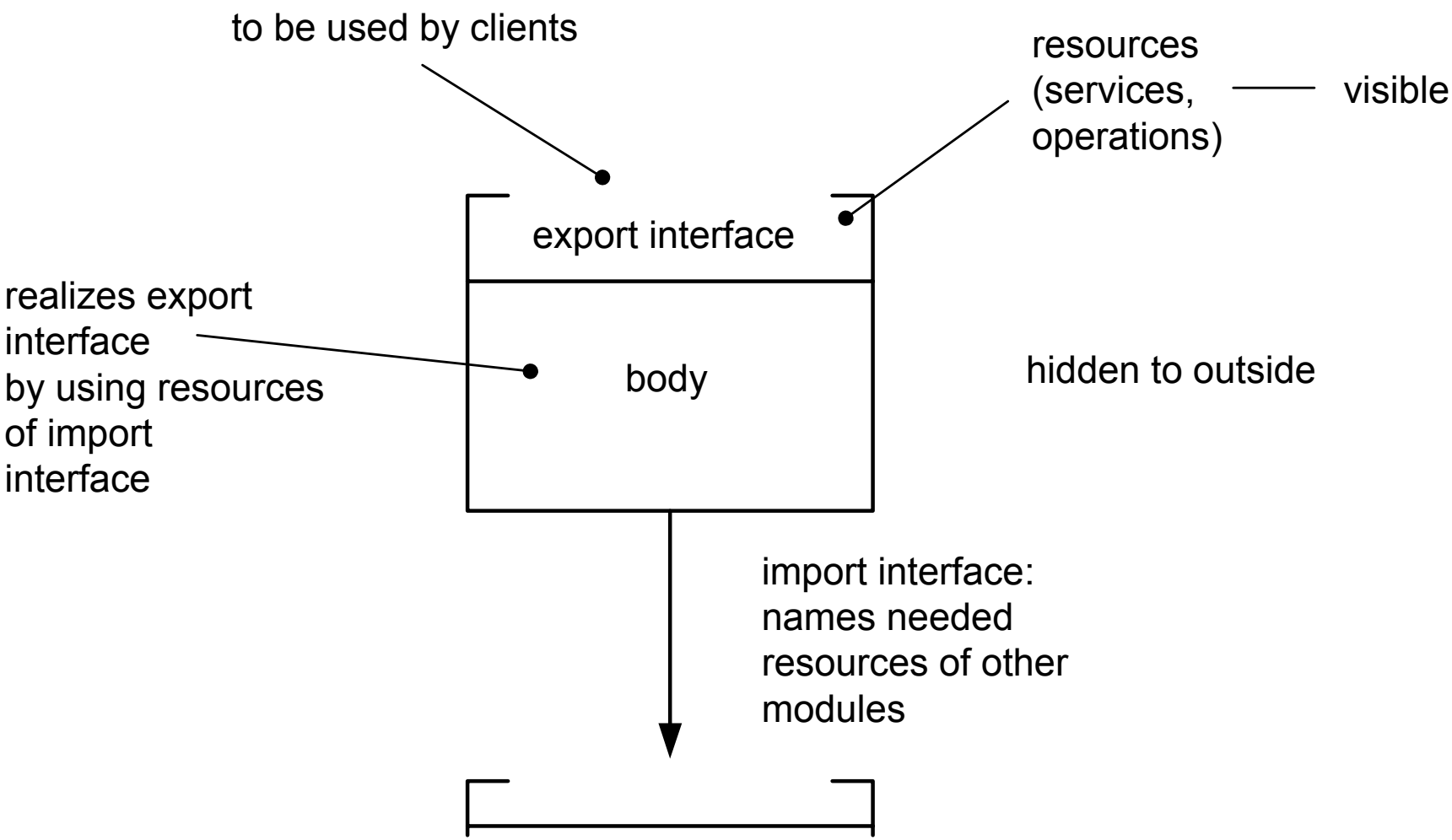
- Logical unit, one sentence for description
- Abstract machine or part of it
- Module has tight internal coupling and loose coupling to other modules
- Represents one design decision; architecture is sum of all decisions
- Unit consisting of data and operations
- Module has a certain complexity (length of spec, length of source code)
- Offers resources to clients: interface components simple and orthogonal
- Module realization (body) hidden

What is a Module? (Continued)

- For realization import from other modules
- Module is side-effect free:
  What it does can completely be seen from the resources of the interface (and imports)
- Module can be exchanged by another with the same interface:
  no influence for semantics (but for pragmatics)
- Correctness can be proven without knowledge of its use
- Unit of independent development (project organization)
- Unit of reuse
- Unit of compilation

etc.

The Parts of a Module



Modules are Logical Units

- No implementation units
  - » as subprograms, macros, …

    are (may be) used for module bodies
- No programming language units
  - » as interface module, implementation module of Modula-2

    both build up a module
- No realization details in the first step:
  - » sequential, concurrent, reentrant, distributed

    these details are regarded later

Interface and Body

```
abstract data object module ITEM_STACK is      --****Export Interface**** --
    …
    procedure PUSH (X: in ITEM_TYPE);          --access operations of    --
    procedure POP;                             --FIFO data structure:    --
    function READ_TOP return ITEM_TYPE;        --for update and read     --
    function IS_EMPTY return BOOLEAN;          --                        --
    function IS_FULL return BOOLEAN;           --                        --
    ST_UNDERFLOW, ST_OVERFLOW: exception; --                             --
--Semantics: PUSH puts an element on the stack, POP deletes the upmost element. READ_TOP reads the upmost
--element. IS_EMPTY and IS_FULL are security read ops. ST_UNDERFLOW and ST_OVERFLOW are …
end ITEM_STACK;
--= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = --
module body ITEM_STACK is                      -- Body of the module     --
    SIZE: INTEGER := 100;                      -- one very simple        --
    SPACE: array (1..SIZE) of ITEM_TYPE;       -- realization            --
    INDEX: INTEGER range 0..SIZE := 0;
    procedure PUSH (X: in ITEM_TYPE) is begin ... end;
    …
    function IS_FULL return BOOLEAN is begin ... end;
    …
end ITEM_STACK;
```

Export Interface: design part

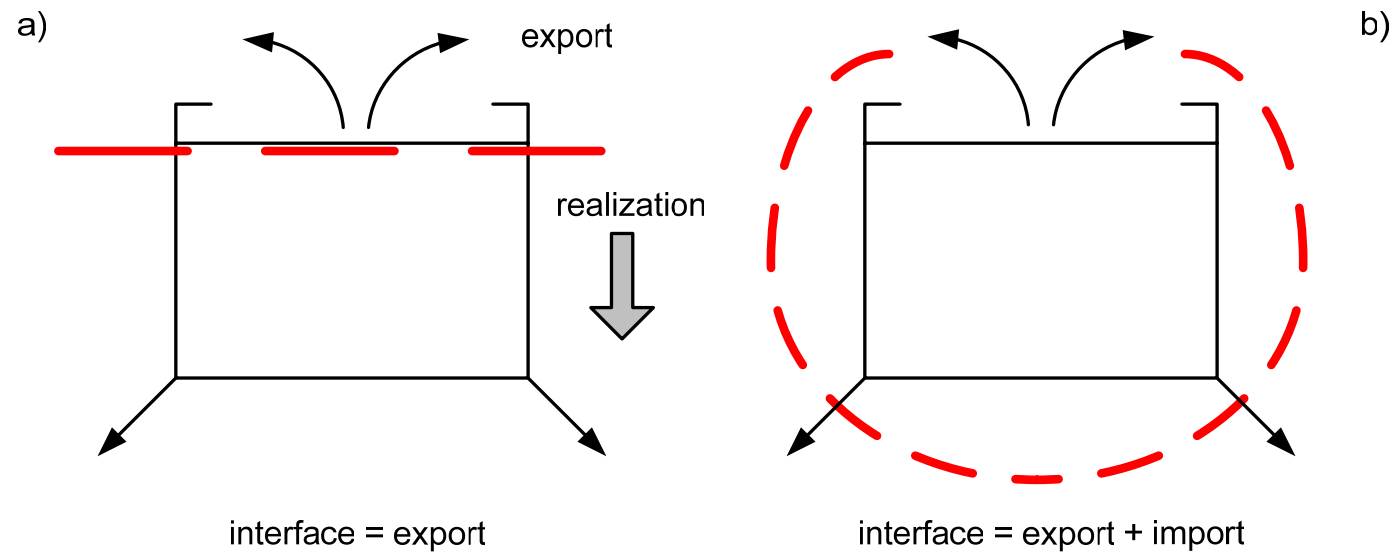
Programming in the small part of the module

Information Hiding

Purpose of hiding details of the body

- Abstraction from body details: export interface
- Basis of “Architecture Paradigm”
- Client does not want to and must not see body details
- Basis for realization exchange and reuse

Wording “Interface”



- Architecture diagram:
  - » Symbol for module + kind of module + name of module
  - » No resources of export interface given
  - » Imports as relations
- Textual description language (interface connection language)
  - » In addition:
    - ⇨ Syntactical (and semantical) description of export interfaces
    - ⇨ Import clauses

Different Abstractions

- Not just “module”, but regarding different purposes
- Classify: “types” of modules
- Modules for functional abstraction or for data abstraction
- Functional abstraction
  - » 1 “type” (but see later chapter):

    Functional module or shorter function module
- Data abstraction
  - » 2 “types”

    (abstract) data object module

    (abstract) data type module

Functional Abstraction

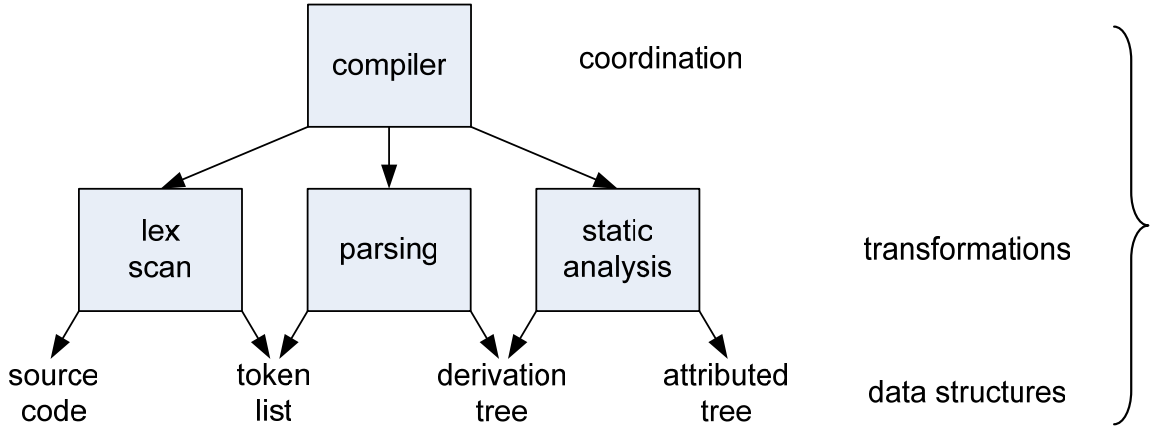
- Characterization of functional modules
  - » transform input to output data
  - » are “activity-oriented”
  - » I/O data appear in the export interface
  - » can have more than one operation in the interface
  - » no state (memory-less)
- Formalization
  - » pre and post condition for every operation
- Functional abstraction is well – known:
  - » see procedures in programming languages
  - » see task decomposition of chapter 2
  - » see wrong examples of chapter 2
  - » however: function modules may have different operations in interfaces

Use of Functional Abstraction

- Examples:
  - compiler, phases of a multi-pass compiler
  - mathematical functions
  - control of a user dialogue
  - main program
- Typical applications:
  - transformation (e.g. lexical scan)
  - computation (e.g. evaluation)
  - control (e.g. dialogue)
  - coordination (multi-pass compiler)
  - auxiliary service on data (e.g. navigation on a search tree)
  - coupling / integration (e.g. representation / logical data, see model view controller)

Use of Functional Abstraction (contd.)

- Clear architectures
  - » Demand for explicit units serving for integration or coupling (otherwise this functionality is done implicitly by other units)



- Understanding functional abstraction is easy, right application difficult
  - » distinction between functional and data abstraction
  - » right functional abstraction together with data abstraction
- Special case
  - » one operation in the interface: subprogram specification as “module interface”
    (for this special case no specific formulation is introduced)

Data Abstraction

- Data abstraction modules
  - » follow data abstraction (or data encapsulation) principle:

    See only operations on data and not the detailed representation of data or operation realization
  - » have a state (memory)
  - » are “passive”
- Formalization
  - » logical pre and post conditions

    State before, state after
  - » or algebraic equations

$$POP_2 \circ PUSH_1 = ID$$

Data Abstraction Principle

- Central idea
  - » no direct access to data structures
  - » access only via logical operations: interface
  - » operations (updates, reads) and data representation are one unit
  - » data representation details hidden
  - » details are only used by realization of operations
  - » different realizations (data representation, corresponding operations implementation) possible, realizations exchangeable

⇨ a semantical application (for the data abstraction purpose) of information hiding or body → interface abstraction

Origin, Use, Importance

- Origin
  - » algebraic specification, abstract data types from Theoretical Computer Science
- Use
  - » makes any architecture adaptable
    - changes of data
    - but also other changes easier
    - even changes of requirements

  } main message of this lecture
- Importance
  - » enormously important for Software Engineering
  - » to be explained later after having some examples

Wordings “Data Abstraction”

- Wordings
  - » data abstraction principle
  - » abstract data object (ado)
  - » abstract data object module: a module for just one ado
  - » abstract data type (adt)
  - » abstract data type module: a module for one adt

- How abstract is an ado / adt module?
  - » interface more abstract than realization
  - » abstraction from a variety of realizations
  - » data abstraction modules often in lower parts of an architecture

Functional Module Example

```
functional module DRAW_GRAPHS is

    -- Input data in the parameter lists, output is plotterfile                    --
    -- linear axes                                                                 --
    procedure POLYGON_LIN(X,Y: in FIELD; X_TEXT, Y_TEXT, CAP_TEXT: in STRING);     --
    procedure INTPOL_LIN(X,Y: in FIELD; X_TEXT, Y_TEXT, CAP_TEXT: in STRING);      --
    procedure APPROX_LIN(X,Y: in FIELD; X_TEXT, Y_TEXT, CAP_TEXT: in STRING);      --
    --                                                                             --
    -- further procedure specifications with the same parameter list               --
    procedure POLYGON_HLOG(...);                  -- semi-logarithmic              --
    procedure INTPOL_HLOG(...);                   -- mode                          --
    procedure APPROX_HLOG(...);                                                    --
    procedure POLYGON_DLOG(...);                  -- double-logarithmic            --
    procedure INTPOL_DLOG(...);                   -- mode                          --
    procedure APPROX_DLOG(...);                                                    --

    ...
```

Functional Module Example (contd.)

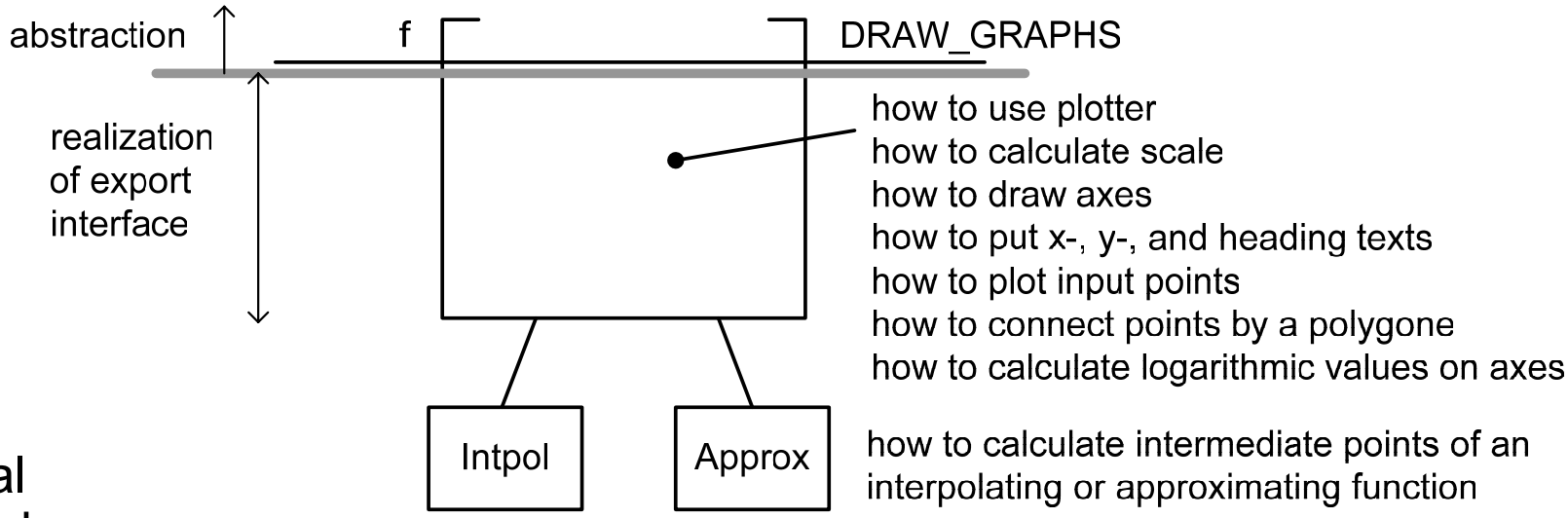
```
    -- semantics of DRAW_GRAPHS:                                                    --
    --        Let X, Y be parameters of an array type FIELD, that is               --
    --        Xi, Yi ∈ REAL, i = 1, ..., FIELDSIZE                                 --
    --        X_TEXT, Y_TEXT, CAP_TEXT ∈ STRING                                    --
    --                                                                             --
    -- global post conditions:                                                     --
    --        plotted output on a sheet with appropriately scaled axes,            --
    --        X_TEXT at the x-axis, Y_TEXT at the y-axis, and CAP_TEXT as headline. --
    --                                                                             --
    -- post conditions for operations:                                             --
    -- POLYGON_LIN connects input points by a polygon, linear  x- and y-axes       --
    -- INTPOL_LIN connects input points by a smooth spline curve, linear  x- and y-axes --
    -- …                                                                           --
end DRAW_GRAPHS;
```

Example Explained

- Typical transformation module
  - » X, Y input data
  - » output data structure is plotterfile (to be seen within architecture)
- Why one module DRAW_GRAPHS?
  - » same parameters in all interface operations
  - » for all operations similar realization:
    - ⇨ calculating scales of axes and output texts
    - ⇨ plot input points
    - ⇨ connect points by a polygon
  - » in addition for half- or double-logarithmic mode: calculate logarithmic values
  - » in addition for INTPOL: calculate intermediate points by splines
  - » in addition for APPROX: calculate approximation points
- Realization within body of DRAW_GRAPHS
  - » better: interpolation and approximation by other modules below

Functional Abstraction

- For this example



- In general
  - » f may have
    - local procedures in its body
    - use other modules for its body realization
  - » functional abstraction
    - abstracts from realization details
  - » details may be
    - in the body of f
    - in the architecture below f

Specifics

- Only one operation in the interface of the function module
  - » e.g. the numerical calculation of a mathematical function
  - » may nevertheless be complicated and use further modules
  - » for this specific case subprograms in all programming languages

- For functional abstraction we have only functional modules
  - » in sequential programs only one functional object is necessary
  - » may be used more than once by different actual parameters (later in context of concurrency we have function type modules, from which different objects can be generated)

- Aim of this section:
  - » Introduction of Data Object Modules
  - » Data Abstraction Principle and its importance for Software Engineering
- Example

```
abstract data object module PERSON_LEXIC is  --******************INTERFACE*********************--
    -- The lexicon has operations to store, change and find entries                              --
    -- for persons with the following semantics: …                                               --
    procedure FIND (DENOTATION: in STRING_S; INFO: out STRING_L);                                --
    procedure STORE (DENOTATION : in STRING_S; INFO: in STRING_L);                               --
    procedure CHANGE (DEN_OLD, DEN_NEW: in STRING_S; INFO: in STRING_L);                         --
    procedure DELETE (DENOTATION: in STRING_S);                                                  --
    function IS_EL_OF (DENOTATION : in STRING_K) return BOOLEAN;                                 --
    function IS_SPACE return BOOLEAN;                                                            --
    THERE_IS_NO_ENTRY, ALREADY_THERE, MEMORY_FULL: exception;                                    --
end PERSON_LEXIC ; ----------------------------------------------------------------------------------------
```

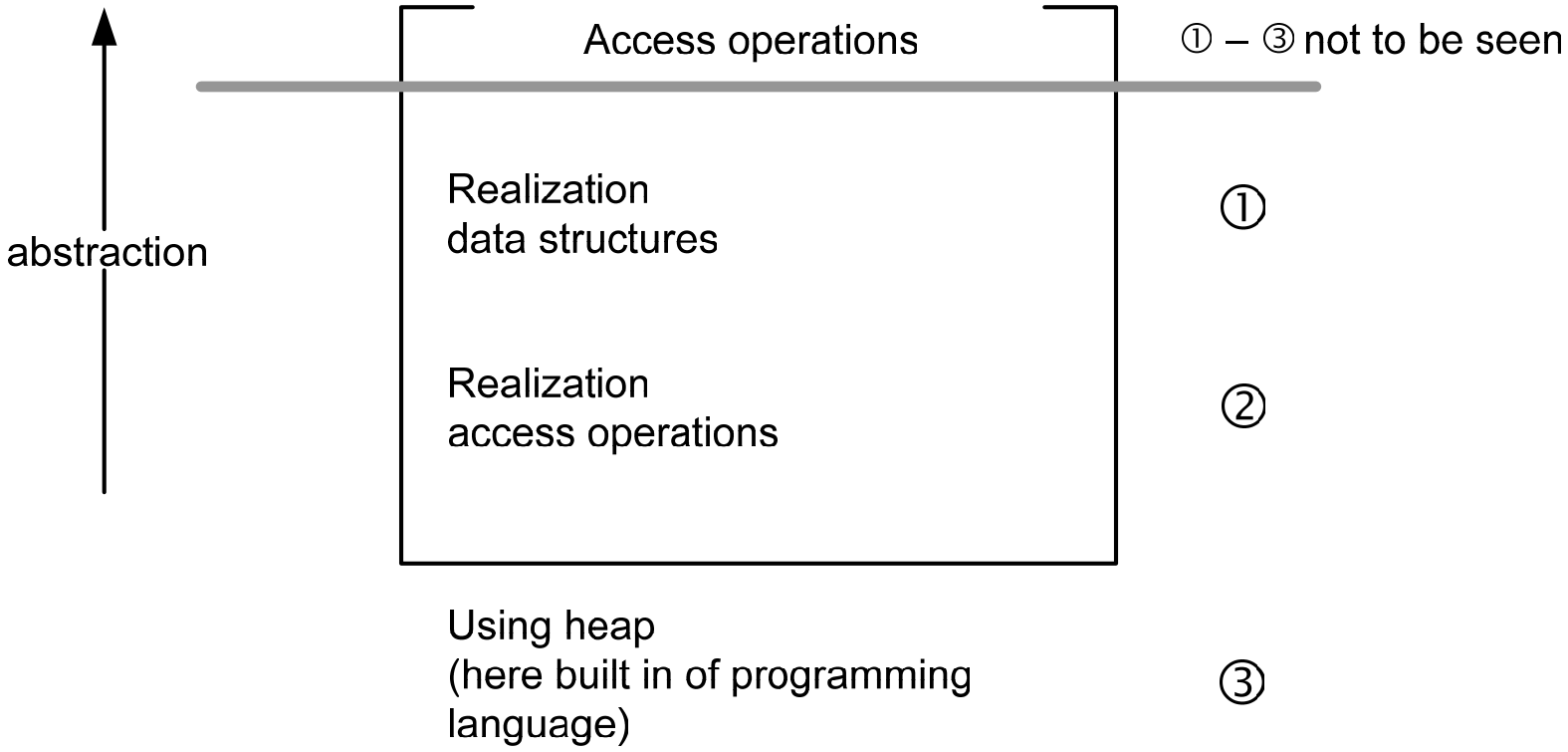
```
module body PERSON_LEXIC is ----------------------------------------------------------------------------------------
    -- declarations for heap:                                                                                    --
    type KEY_VAL is INTEGER range 1..100_000;                                                                    --
    type TREE_LE; type P_TREE_LE is access TREE_LE:                                                              --
    type TREE_LE is                                                                                              --
        record                                                                                                   --
            KEY: KEY_VAL;                                                                                        --
            INFO: STRING_L;                                                                                      --
            LEFT_SON, RIGHT_SON: P_TREE_LE := null;                                                              --
        end record;                                                                                              --
    REF_ROOT, FOUND_NODE: P_TREE_LE; CONTAINED: BOOLEAN;                                                         --

    -- local procedures:                                                                                         --
    function PRIM_KEY(DENOTATION: in STRING_S) return KEY_VAL is …;                                              --
    procedure SEARCH_IN_TREE(KEY: in KEY_VAL; START_NODE: in P_TREE_LE:= REF_ROOT;--
            SUCCESS: out BOOLEAN; END_NODE: out P_TREE_LE) is …:                                                 --

    -- realization of interface ops:                                                                             --
    procedure FIND(DENOTATION: in STRING_S; INFO: out STRING_L) is …                                             --
    …                                                                                                            --
    function IS_SPACE return BOOLEAN is … ;                                                                      --
begin -- statement part, for initialization only                                                                 --
    …                                                                                                            --
end PERSON_LEXIC; --************************************************************************************** --
```

Summary ado Module Example

- Interface
  - » Access operations FIND, STORE, CHANGE
- Body: realization
  - » realization of data structures
    here on heap
  - » realization of access operations
  - » thereby using other components
    internal, or external

} mutually dependent

Data Abstraction: Information Hiding for a Semantical Purpose



Telegram Example Revisited: Exercise

- Reformulate Interface Telegram Input

Telegram Example Revisited: Exercise

- Formulate Interface of Triple List
- Specific question: What's about sort-operation?

Card-Box Example Revisited: Namelist

- Application of data abstraction: Card-Box Namelist

```
abstract data object module CB_Namelist is --*****************************************************--
    procedure Open_CB_NL;                                    -- operations on          --
    procedure Close_CB_NL;                                   -- the whole              --
    function Is_CB_NL_Empty return BOOLEAN;                  -- list                   --
    function Is_CB_NL_Nonfull return BOOLEAN;                                          --
    function Does_CB_Name_Exist return BOOLEAN;              -- dealing with           --
    procedure Insert_CB_Name(Name: in CN_Type);               -- elements              --
    procedure Delete_CB_Name(Name: in CN_Type);                                        --
    procedure Reset_CB_NL;                                    -- moving                --
    procedure Output_Current_CB_Name(Name: out CN_Type);      -- in the                --
    procedure Next_CB_Name;                                  -- list                   --
    procedure Pred_CB_Name;                                                            --
    function Does_Succ_Exist return BOOLEAN;                                           --
    function Does_Predec_Exist return BOOLEAN;                                         --
    CB_NL_empty, CB_NL_full, CB_Name_not_ex, CB_Name_succ_not_ex,                      --
    CB_Name_pred_not_ex: exception;                                                    --
end CB_Namelist; ------------------------------------------------------------------------------------------
```

Card-Box Example Revisited: Entry

- Application of data abstraction: Complex entry
  (new: up to now only collection applications)

```
abstract data object module CARD is --*********************************************************--
    -- The parameter types KEY_T, NAME_T etc. are visible                                      --
    procedure init_Card;                                                                       --
    procedure delete_Card_contents;                                                            --
    procedure insert_Key(KEY: in KEY_T);                                                       --
    function read_Key return KEY_T;                                                            --
    procedure insert_Name(NAME: in NAME_T);                                                    --
    function read_Name return NAME_T;                                                          --
    ...                                                                                        --
    -- A read and write operation per entry component                                          --
    function is_consistent(KEY: in KEY_T) return CASE_T;                                       --
end CARD; ------------------------------------------------------------------------------------------
```

Check is_consistent depends on entry: syntactical, semantical etc. checks.

Security operations, exceptions seldom in entry application.

Broad Entry

- Many components ⇨ Large interface
- One unspecific read and write op?

```
abstract data object module COMPLEX_RECORD is --************************************--
    procedure READ(COMP: in COMP_DENOTATION; INFO: out INFO_TYPE);        --
    procedure STORE(COMP: in COMP_DENOTATION; INFO: in INFO_TYPE);        --
    ...                                                                   --
end COMPLEX_RECORD; ---------------------------------------------------------------------------
```

- Discussion:
  - \+ less effort for developing interface
  - \- unspecific uses: loss of security
  - \- unspecific parameter type (string, variant record): loss of security
  - \- more difficult for implementor and client: conversions
- Other solution later: a broad entry may aggregate different entries (name, address, employment data, etc.)

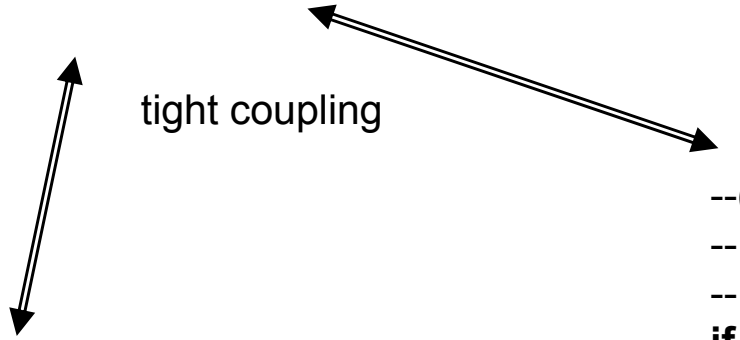
Localize Dangerous Situations

- e.g. pointers
  - » aliasing
  - » inaccessible objects
  - » dangling references
- within one data abstraction module
  - » from outside not to be seen, that pointers are used
  - » corresponding module developer is hopefully careful

Application of Data Abstraction Produces Adaptability (1)

Situation without data abstraction

```
--(1)
-- Realization of data structure
SPACE: array (1..SIZE) of ITEM_TYPE;
INDEX: INTEGER
```



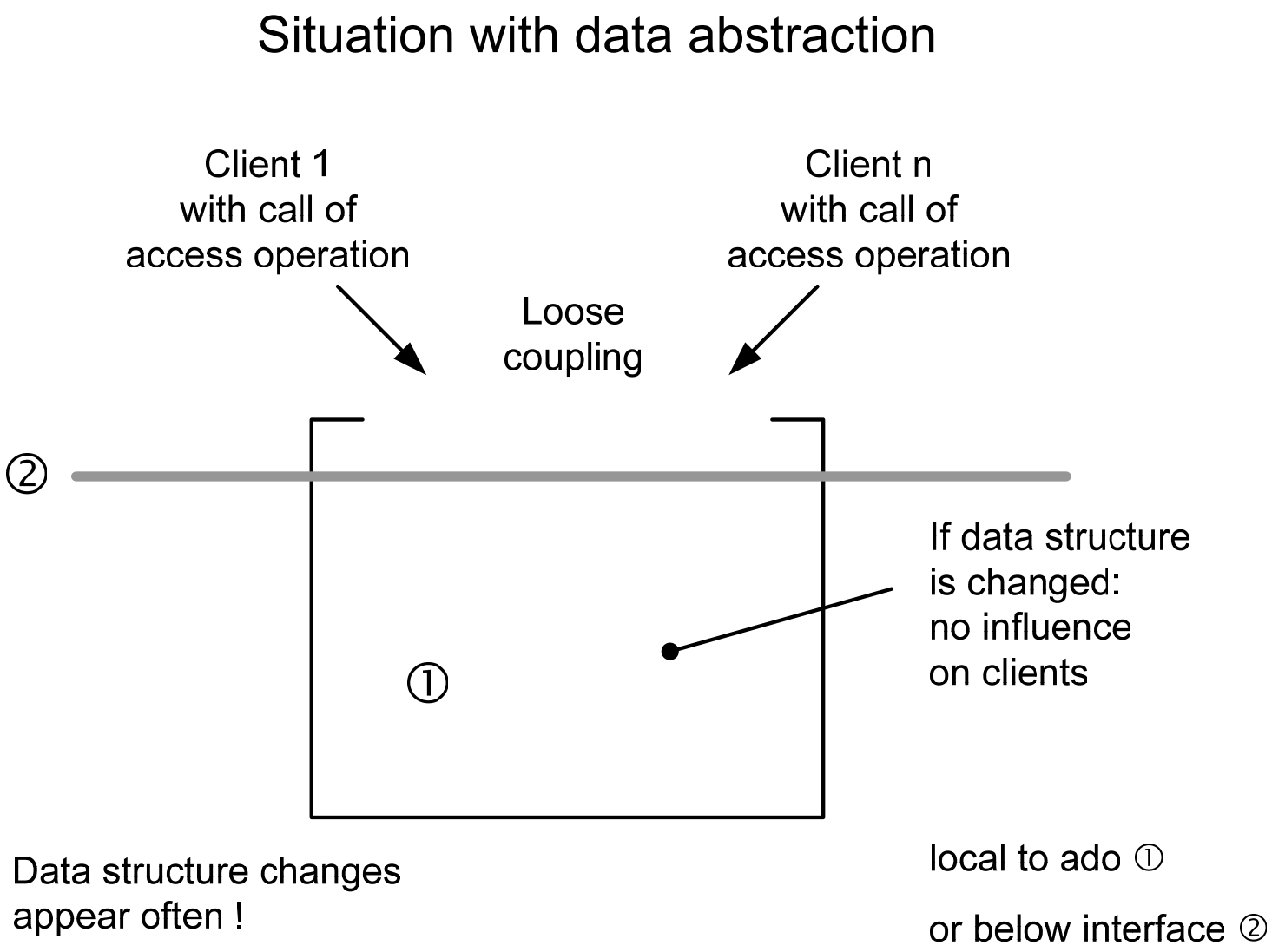

```
--(3)
-- Realization of PUSH
-- procedure PUSH(X: in ITEM_TYPE);
if INDEX < SIZE then
        INDEX:= INDEX +1;
        SPACE(INDEX):= X;
else
        …
end if;
```

```
--(2)
-- Realization of POP
-- procedure POP;
if INDEX >= 1 then
        INDEX:= INDEX -1;
else
        …
end if;
```

any change produces
many other changes

if there is a possibility
of direct access: it is used

Application of Data Abstraction Produces Adaptability (2)



Data Abstraction Allows Different Realizations

- Statically addressable storage
- Run-time stack
- Heap

Thereby sequential lists, chained lists, binary trees, etc.

- Persistent storage

B-tree, B*-tree, etc.

- Node in a network: access via remote procedure call

Data Abstraction and Unification

- No longer distinction between programs and data:
  complex data appear as modules (ado or adt) in the architecture
- Architecture contains the complete structure of the program system
  (the whole complexity story)

Ado Interfaces and Security

- Systematical interfaces
  - » Change operations
  - » Query operations
  - » Security operations
  - » Exceptions

	(a)	(b)
Application of access operation	**if** IS_SPACE **and** **not** IS_EL_OF(…) **then** STORE(…, …) **end if**;	STORE(…, …);
Execution of access operation	regular execution	eventually raising MEMORY_FULL or ALREADY_THERE

this way is more secure:
exceptions, if client not careful

Alternative: Return (Success) Parameter

```
-- changed interface
procedure STORE(DENOTATION: in STRING_S; INFO: in STRING_L;
        OK: out BOOLEAN);


-- application
STORE(ACT_DENOT, ACT_INFO, SUCCESS);
if SUCCESS then
        -- proceed
        …
else
        -- do something else
        …
end if;
```

Less secure: if success parameter is not inspected, something might go wrong

success parameters: efficiency or in concurrency context

Standard Data Abstraction Applications (up to now only 2)

Application class of data abstraction	Abstraction: logical view	Realization: physical view
Complex entry	aggregate comp 1 ... comp n Read- and write access on “logical” components	Component order sequentially, scattered Realization of order by sequential storage, using indices, or pointers Packed components, single component must be computed Additional components for internal use, e.g. statistics
Collection	Store, find, change and delete elements of the collection set of entries +access method Access method (FIFO, LIFO, RANDOM, etc.)	Realization of the (ordered) set: sequential, by indices, pointers, etc. Storage localization: static memory area, runtime stack, heap, secondary storage, network node, etc. Realization of access method: calculation, construction of access paths Additional components for internal use

Motivation

- Need more than one abstract data object:
  - » Multiplying source code?
  - » Template: Abstract data type module
- Difference: abstract (opaque, encapsulated) type
  open (transparent, normal) type

```
-- (a) use of normal data types and objects
type SPACE_T
      is array (1..N) of ITEM_TYPE;   -- (1)
SPACE_1: SPACE_T;   -- (2)
…
if … then
      SPACE_1(INDEX):= X
else …
end if;   -- (3)
```

```
-- (b) use of abstract data types and objects:
STACK_1: STACK_TYPE;   -- (4)
…
if not IS_FULL(STACK_1) then   -- (5)
      PUSH(STACK_1, X);
else …
end if;
```

Stack as adt Module

Interface

```
abstract data type module ITEM_STACK is  --************************************************** --
    ...                                                                                        --
    type  ITEM_STACK_TYPE is private;                                                          --
    procedure INITIALIZE (ST: in out  ITEM_STACK_TYPE);                                        --
    procedure PUSH (ST: in out ITEM_STACK_TYPE; EL: in ITEM_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: …                                                                            --
end ITEM_STACK; ----------------------------------------------------------------------------------------------
```

Use in the body of a client module:

```
ST_1: ITEM_STACK_TYPE;
EI_1 : ITEM_TYPE;
if not IS_FULL (ST_1) then PUSH (ST_1, EL_1) else ... end if;
```

Body of the Stack adt (very simple solution)

```
module body ITEM_STACK is ---------------------------------------------------------------------------
    -- Representation of data type                                                              --
    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;                                                                             --
    …                                                                                           --
    -- Realization of interface operations (depending on the data type representation)          --
    …                                                                                           --
end ITEM_STACK; --************************************************************************** --
```

in many languages: data description in the private part of interface

Task of an adt Module

- Declaring an abstract data type in the interface:
  - » type identifier plus access operations
- Realizing the abstract data type in the body:
  - » declaring the corresponding data structures
  - » realizing the access operations according to declarations
  - » thereby using other components

ado ↔ adt Module

- ado module is an ado
  adt module is a template for ados
- ado module is a part of the architecture
  adt module is a part of the architecture (template for ados)
- The ados of an adt module are not to be seen on architecture level (“generated” in the body of the client)
- Use of adt module in a client
  - » to generate ados
  - » to query / update ados (eventually generated by another client)

adts with Variable Semantics

- ados are generated when a declaration is elaborated in the body of a client module

  ST1, ST2: ITEM_STACK_TYPE

  ado variables

- Prerequisite:
  - » underlying programming language knows type declarations (old FORTRAN and COBOL versions and assemblers do not)
  - » the number of ados is known at compile time

adts with Reference Semantics

- Interface:
  - » Creation (and deletion) of ados
  - » Creation together with initialization or separate, if an ado is used more than once
  - » Type identifier: for references to ados

- Client:
  - » use in the statement part (creation, initialization, … , deletion)
  - » number of generated ados is determined at runtime

Stack with Reference Semantics

```
abstract data type module ITEM_STACK is  --*******************************************--
    ...                                                                                  --
    type STACK_DENOTER_TYPE is private;                                                  --
    procedure CREATE_AND_INIT(ST: out STACK_DENOTER_TYPE);                               --
    procedure DELETE(ST: in STACK_DENOTER_TYPE);                                         --
    procedure PUSH(ST: in STACK_DENOTER_TYPE; EL: in ITEM_TYPE);                         --
    procedure POP(ST: in STACK_DENOTER_TYPE);                                            --
    function READ_TOP(ST: in STACK_DENOTER_TYPE) return ITEM_TYPE;                       --
    function ST_IS_EMPTY(ST: in STACK_DENOTER_TYPE) return BOOLEAN;                      --
    function ST_IS_FULL(ST: in STACK_DENOTER_TYPE) return BOOLEAN;                       --
    function ANOTHER_ST_POSSIBLE return BOOLEAN;                                         --
    ST_UNDERFLOW, ST_OVERFLOW, NO_ST_AVAILABLE, ST_IS_NIL:                               --
        exception;                                                                       --
    -- semantics: …                                                                      --
end ITEM_STACK; ------------------------------------------------------------------------------
```

Must Client Programmer Know whether adt Has Variable or Reference Semantics?

Yes: different situations for created objects

- Variable semantics
  - » Objects on runtime stack
  - » Language implementation creates and deletes
  - » Assignment: copy
  - » Equality: values
- Reference semantics
  - » Objects on a “heap”
  - » Client programmer creates and deletes
  - » Assignment: another reference onto on object
  - » Equality: same reference

- Semantics of different use
  - » to be seen from adt interface

    Syntax

    Comments
- Realization is different
  - » to be seen in the architecture

Data Abstraction and Software Engineering Impact to be Repeated for adts

- Localize dangers
- Loose coupling between modules
- Adaptability
- Many different realization possibilities
- Applications: entries, collections
- How to group interface operations
- Interfaces and security
- Information Hiding (body → interface) is used for a semantical purpose

In Sequential Systems f, ado, adt Modules to be Used for

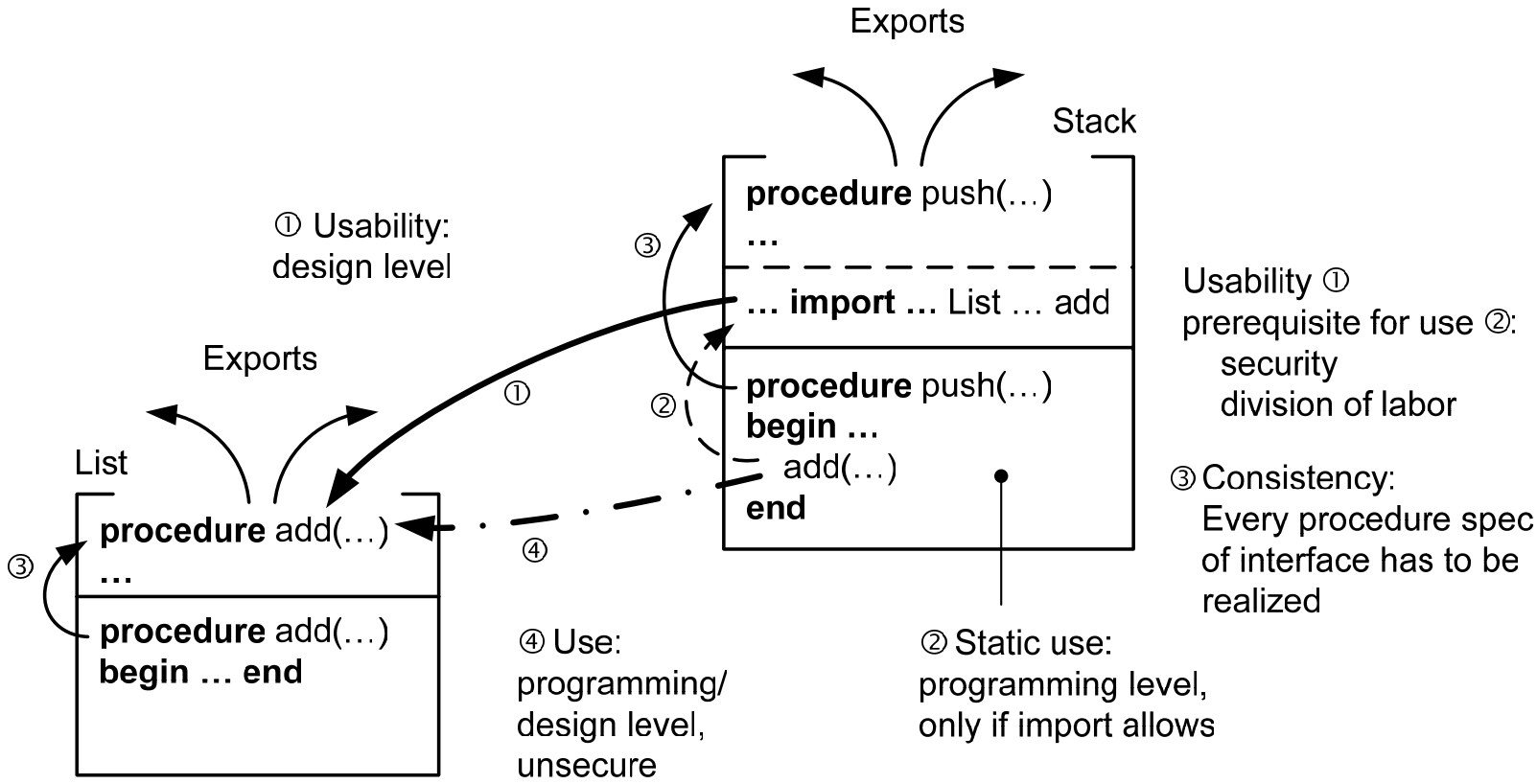
Module type	Functional module	Abstract data object module	Abstract data type module
Where to use	❑ Coordination and control ❑ Transformation ❑ Complex evaluation Functional intermediate layer between data abstraction layers	Single entry ❑ Single collection with specific access operations	❑ Complex entries ❑ Collections with specific access operations
		❑ main applications	

Motivation for Second Part of this Chapter

- Even more important than module “types”
- Not one relation but different relations corresponding to purpose of use
- Logical relations, not relations between programming language units, as e.g. separate compilation
- Some relations are bound to underlying “structure” relations

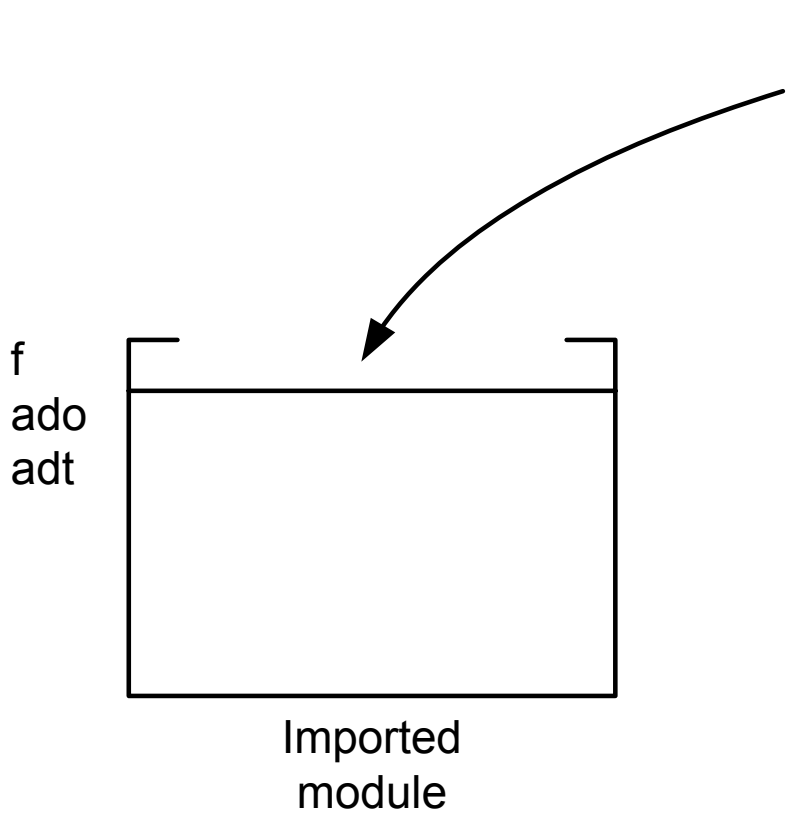
Relations on which Level?

- Usability or import level: design ✓
- Statical use level: programming
- Dynamical use level: runtime execution



What is Made Usable?

- Depends on the “type” of the imported module:



If imported module is

f: usability of functions

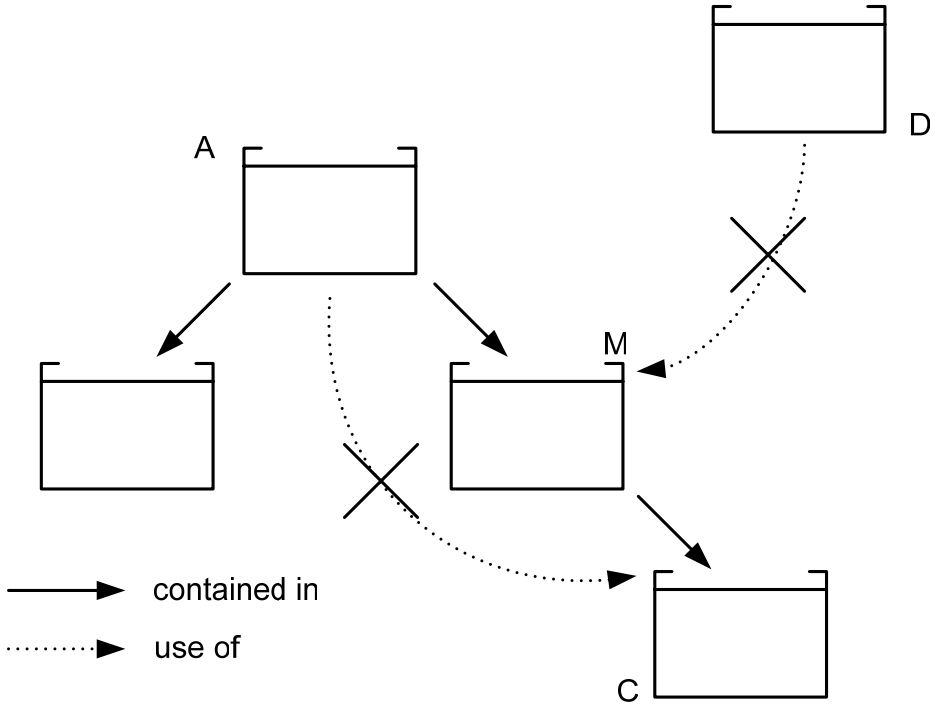
ado: usability of access operation

adt: usability of type / creating operations and / or access operations

Origin: Block-oriented Languages

- Other kinds of relations will follow
- Distinguish from the “logics” of use on architecture level, not from the programming language onto we map
- Origin of local usability:
  Nesting, block structures, scope, visibility from Algol-type languages
- First definition
  A module M is “contained” in some other one and, therefore, only usable in a local context. There, we specify where M is usable

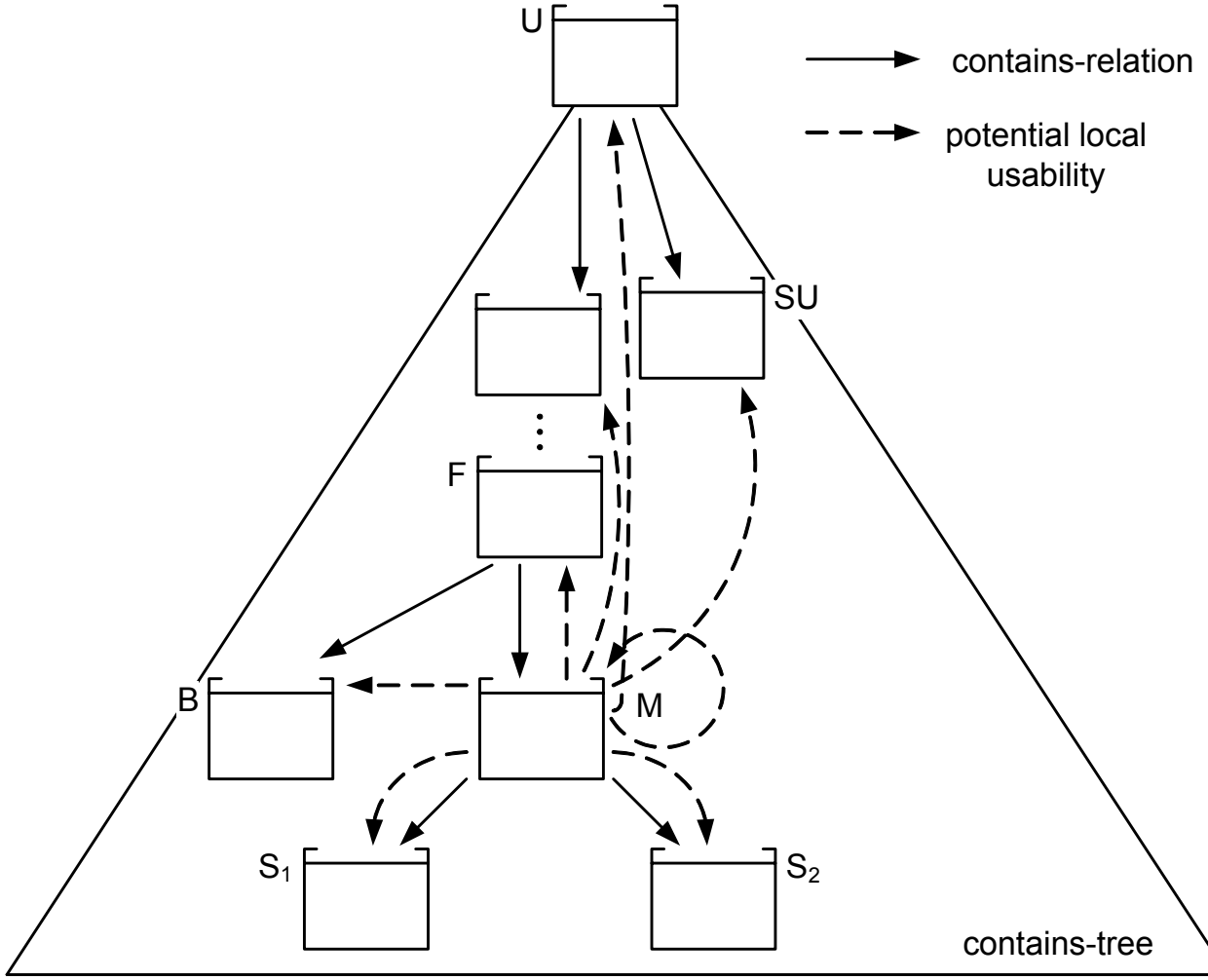
What Do We Express ?



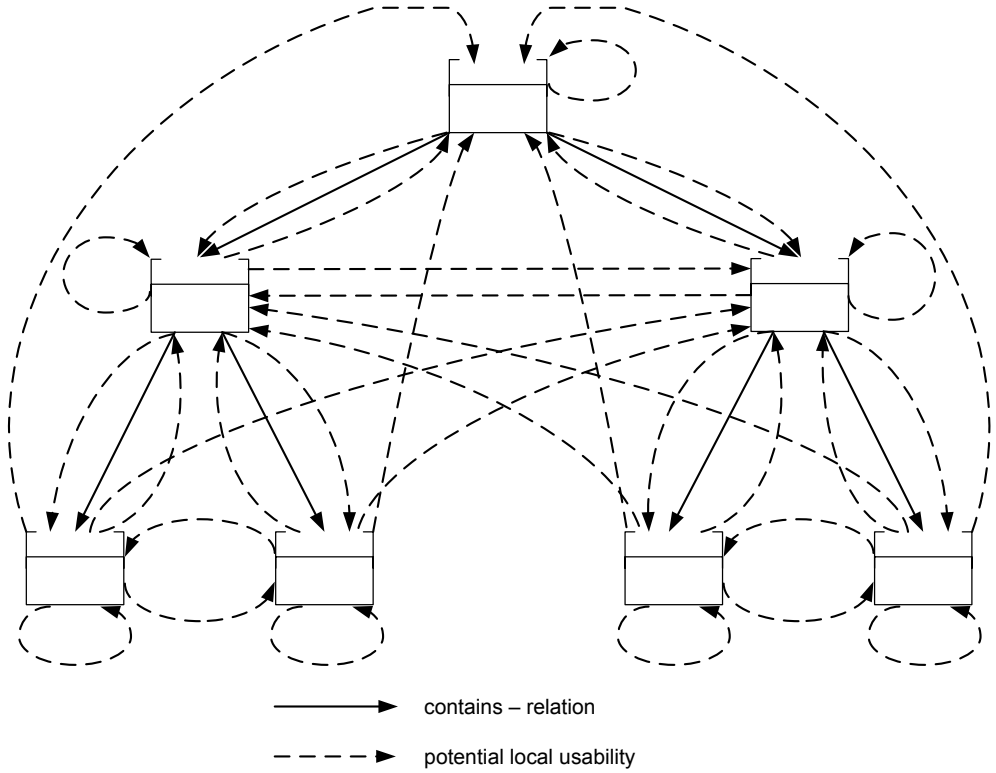
- M contained in A:
  Realization of A can make use of M
  Local importance of M
- Contains-relation
  - » an arrow in graphical description
  - » a clause in textual description (not by nesting)

Usability in a Local Context

- Do not regard “Who can use M” (scope, visibility)
- but inverse relation “What can M use”



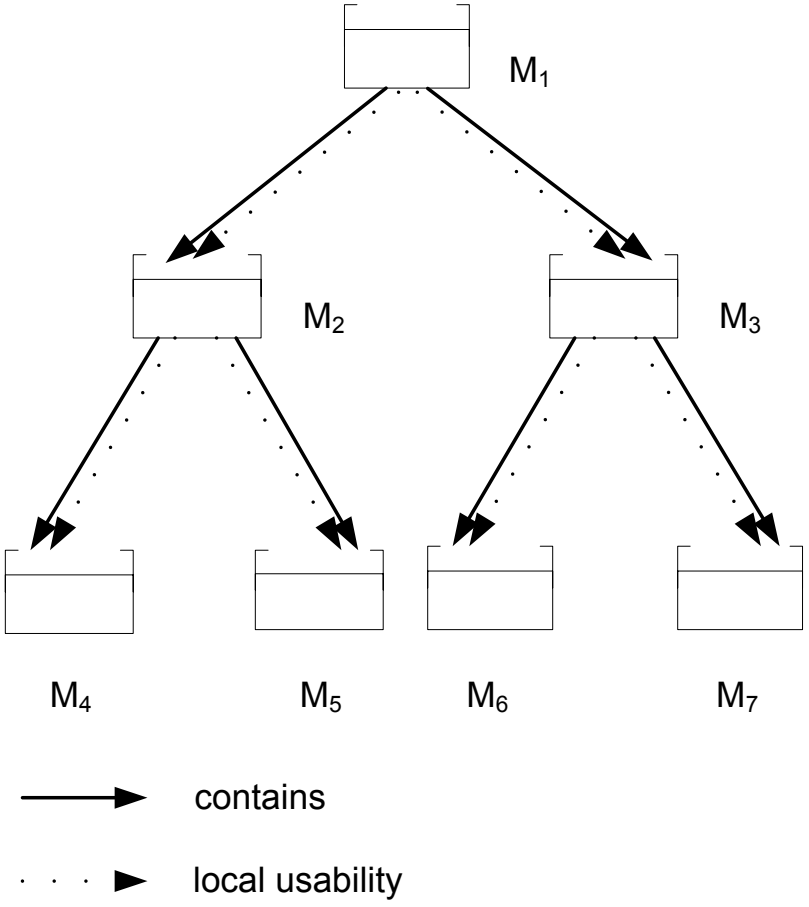
An Example with only 7 Modules: Chaos



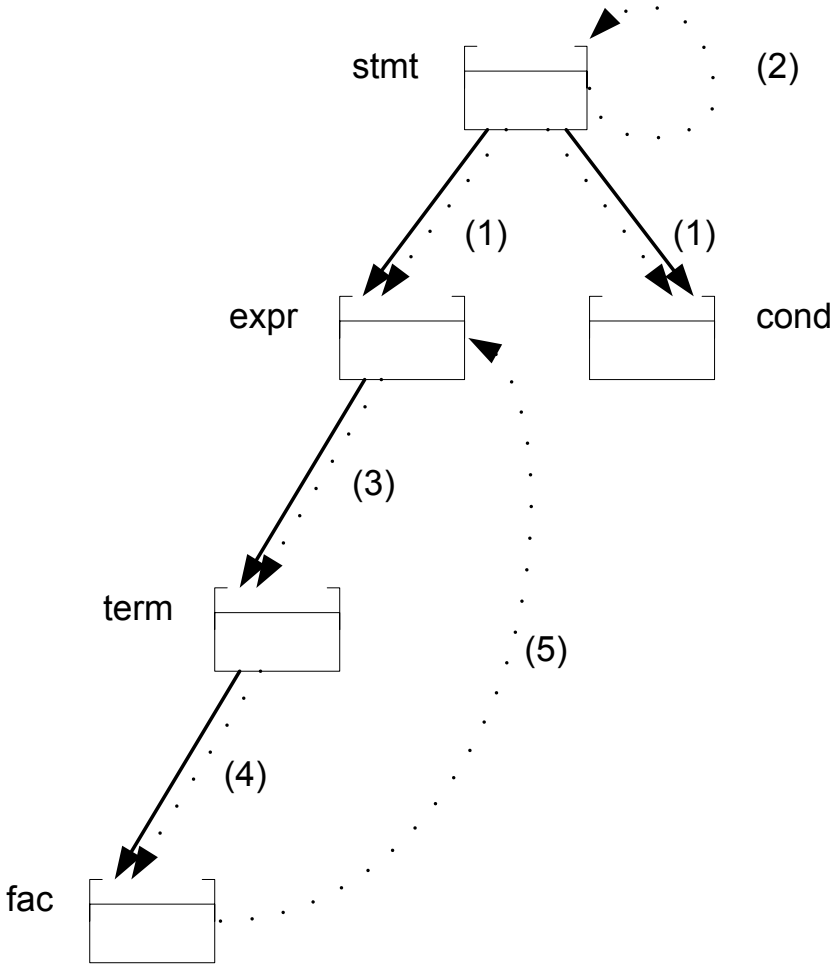
- Consequence:
  - » explicit determination of usability (specific import)
  - » only from possibilities of potential local usability
- Denotation
  - » edge of a specific kind in architecture diagrams
  - » specific clause in text notation

Standard Situations

a) standard case



b) example: recursive descent compiling



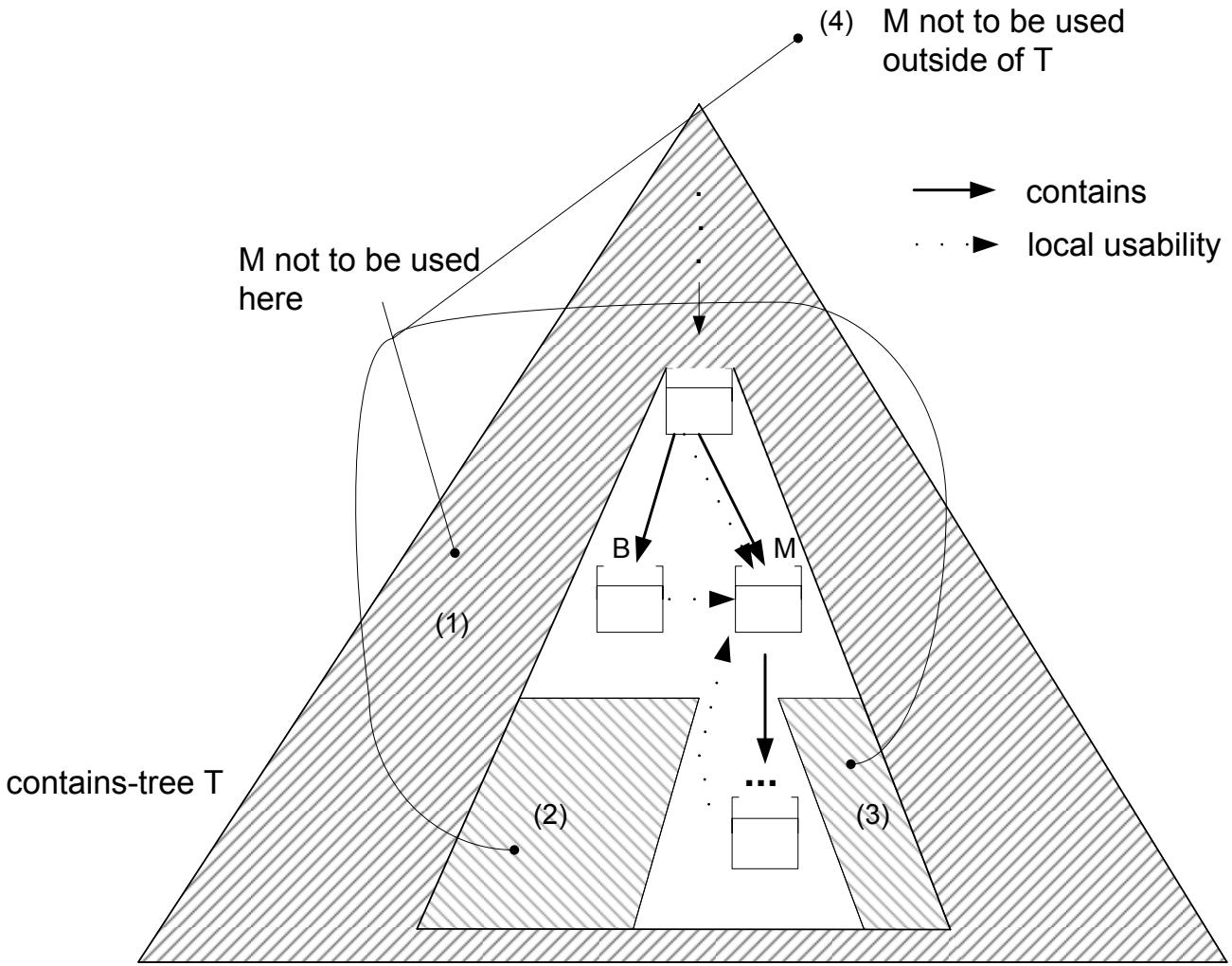
What Have We Gained ?

- Local importance by contains-relation
  - » restrictions of use: potential local usability
  - » explicit determination by local usability

- Therefore
  - » which modules are allowed to use M
  - » what M can use

Local Usability and Security

Regard now “Who can use M” and not “Whom can M use”



(1) – (4) no use, not even usability

Resumee

Locality principle

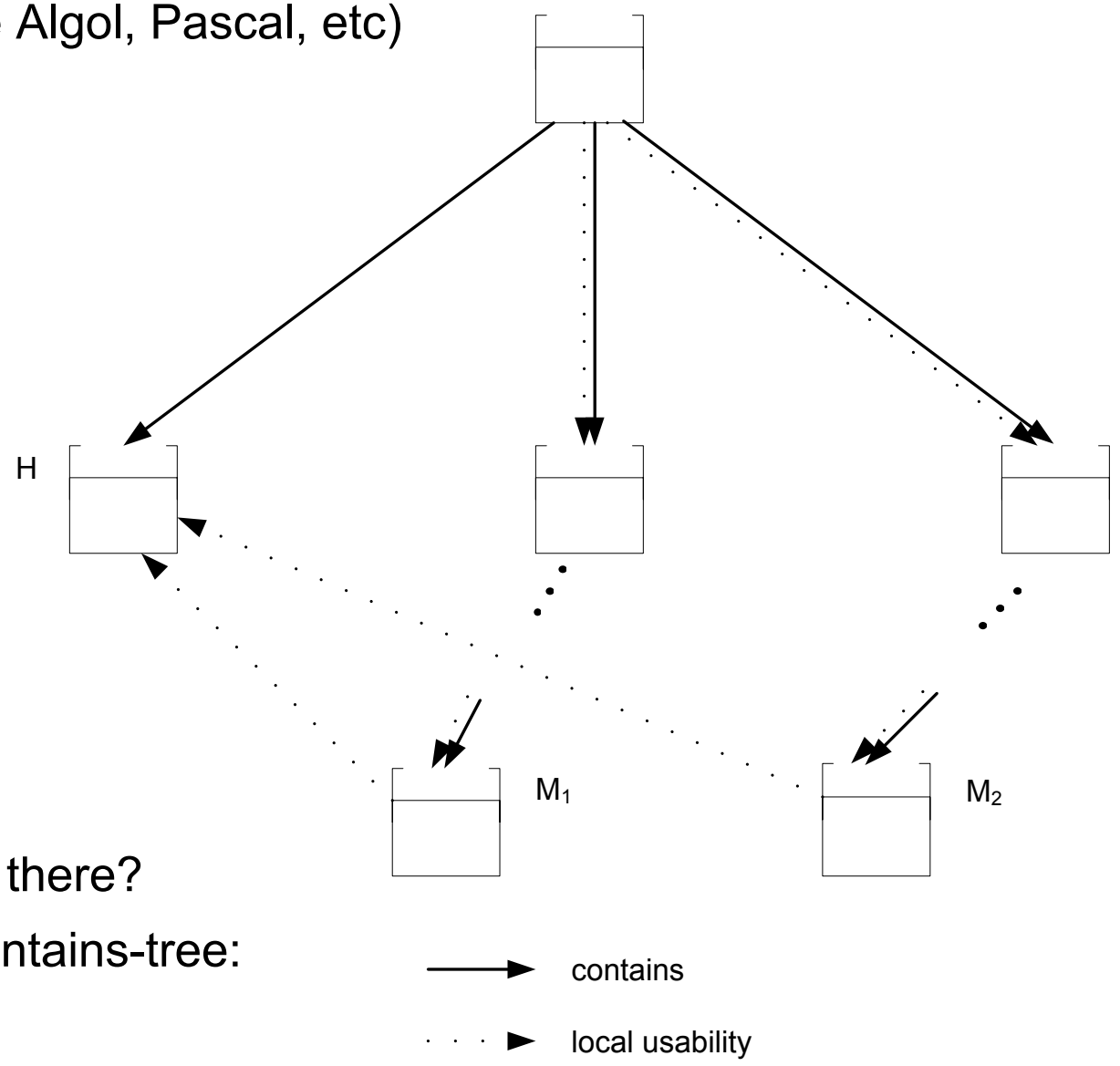
- Comes from block-structural languages
- Nesting as structure relation “contains”
- We do not accept use in scope (potential local usability):
  - » explicit determination as design decision
- We do not express locality by nesting

Using Common Components

- Second kind of relation
- Origin: import clauses in programming languages
  - » Modula-2, Ada 83, etc.
- First definition:
  - » a module (later also a subsystem), if regarded to be a common component for the realization of other modules, is usable where explicitly determined

Why Does Locality not Suffice?

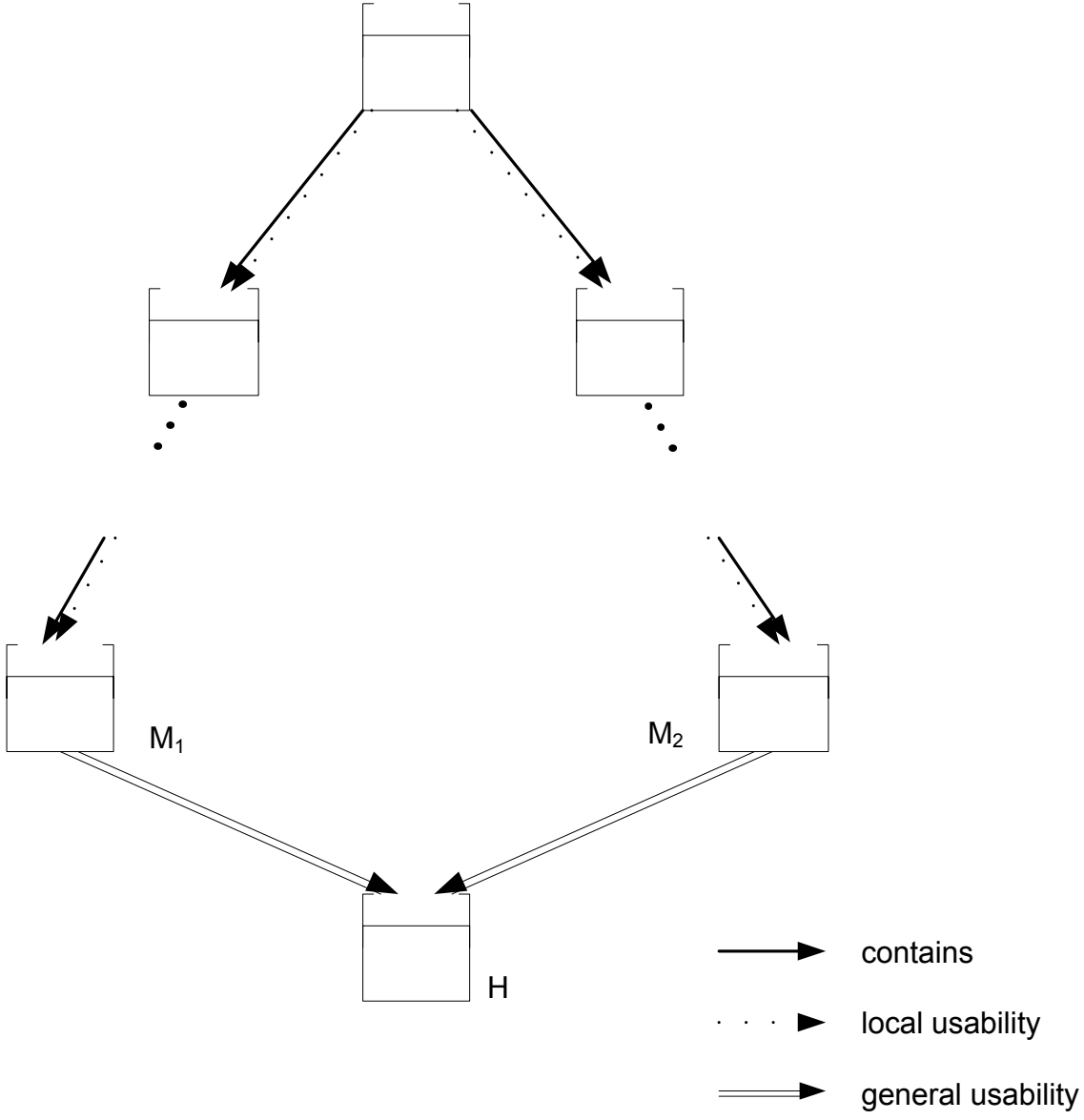
- In principle it does (see Algol, Pascal, etc)



- Where to hang H?
- Can one understand H there?
- If needed in another contains-tree: even move up

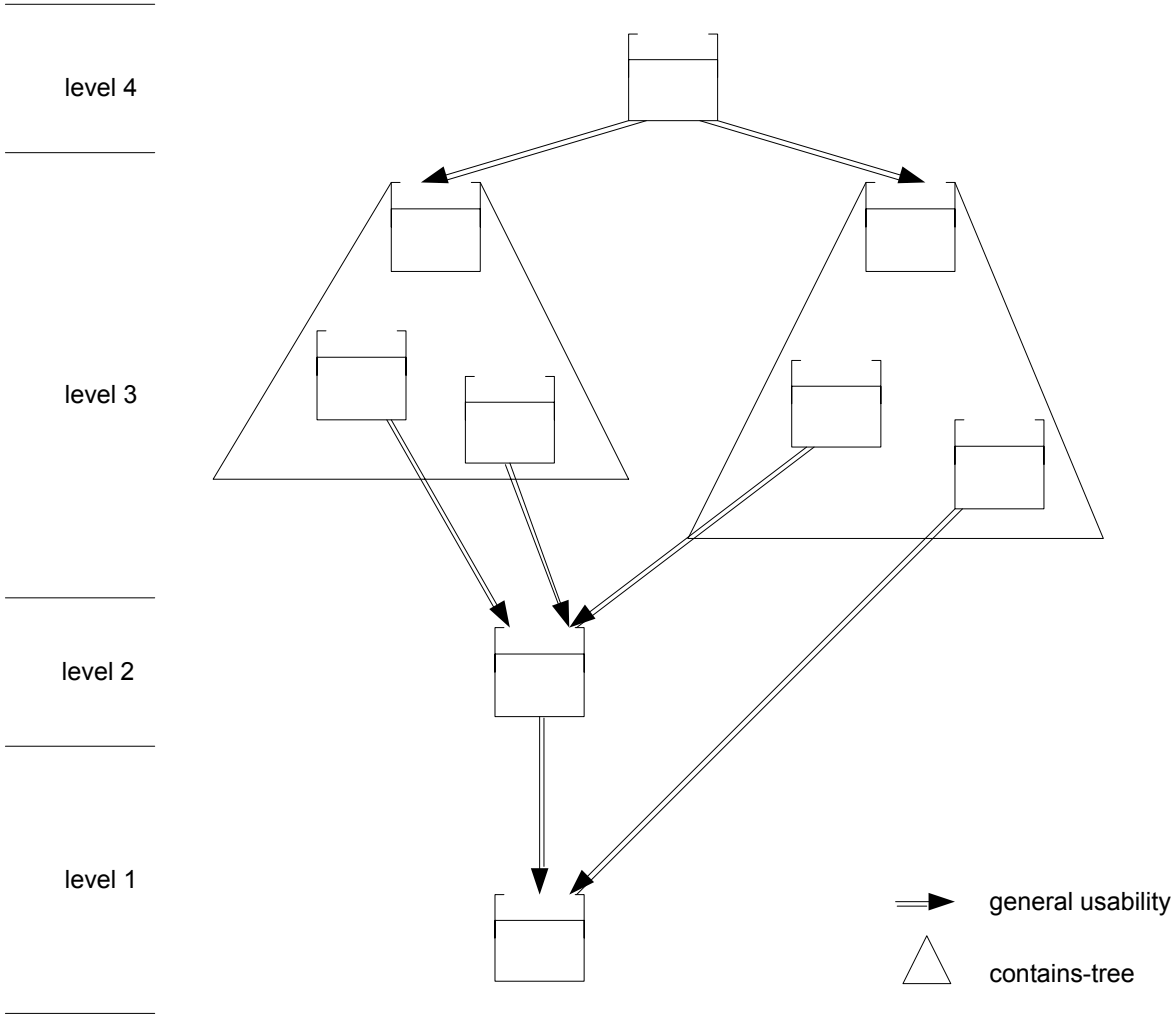
What We Would Like to Express

- In architecture diagram: different and new kind of edges
- In textual notation: different import clause



Layers by General Usability

- Usable module is logically deeper than using module: (not necessarily strict) hierarchy



More Precise

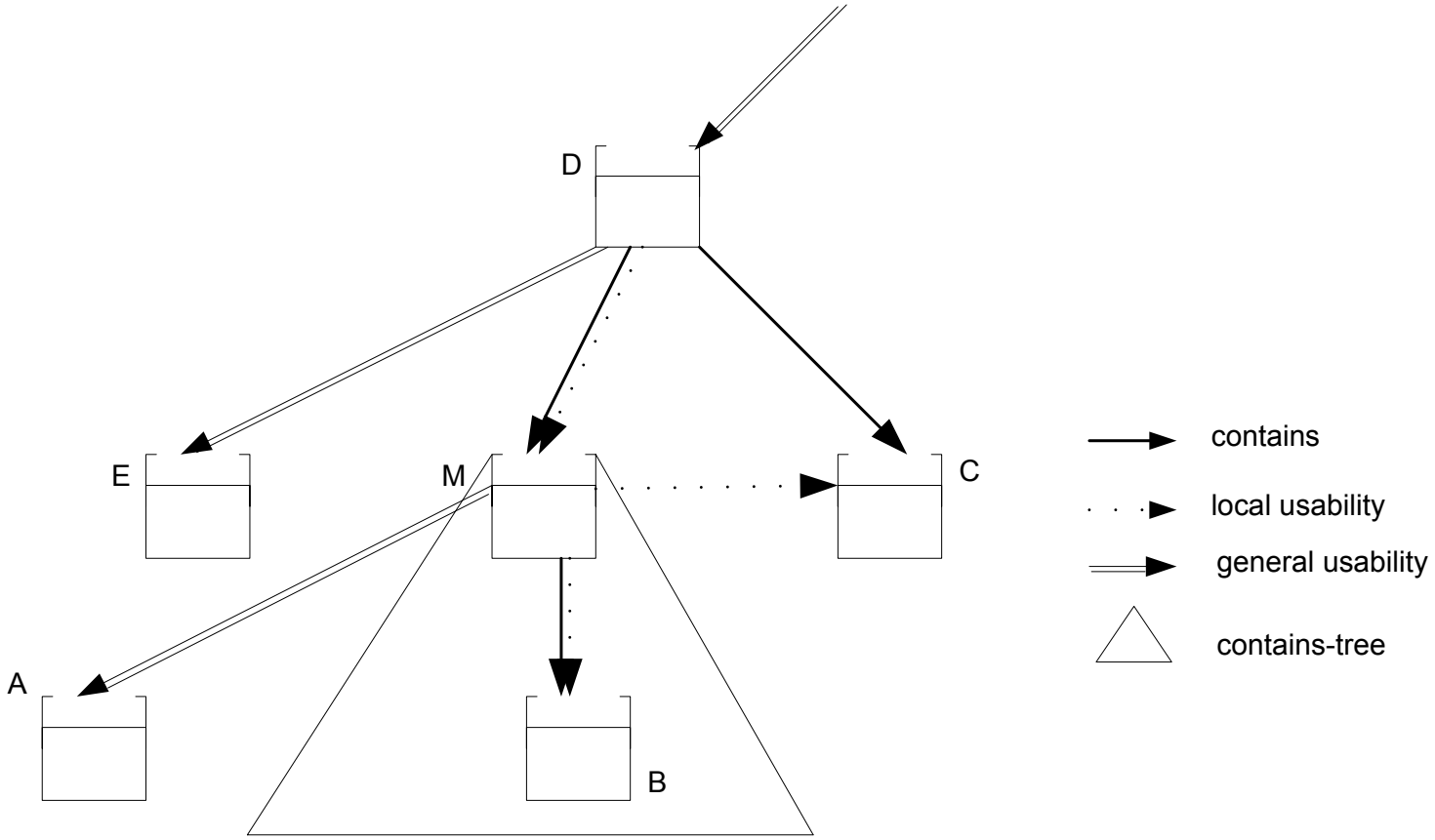
- General (or common) module
  - » designed with the purpose of multiple use
  - » or later generalized
  - » more often data abstraction than functional module
- Hang a common module into an architecture
  - » need not know realization:

    Information hiding on architecture level
  - » has to know whether local or general use

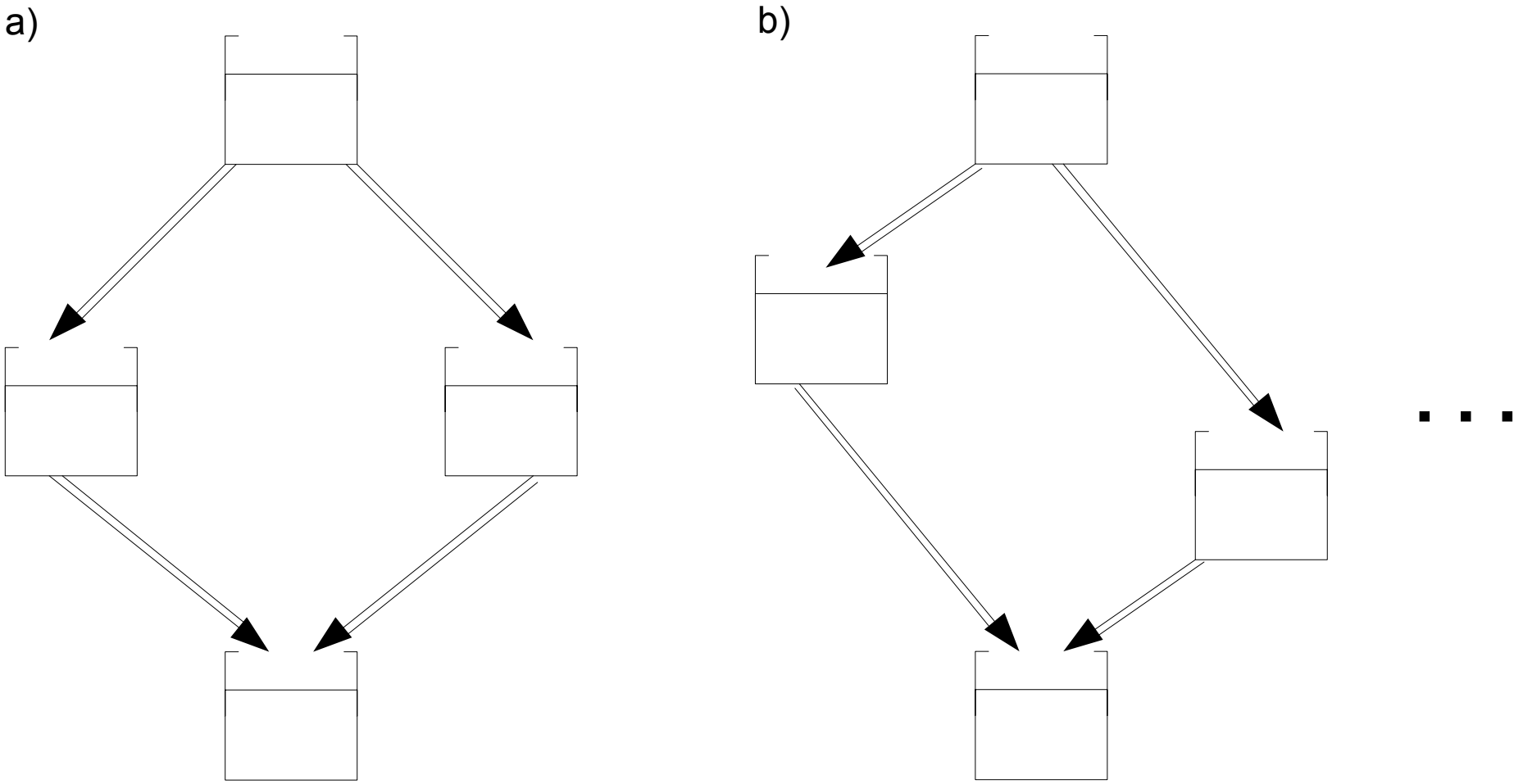
Different Module Relations: Export / Import

- What can M, D use?
- Import only for importing module

  not for its export (latter possible for subsystems)



General Usability and Layers: not Unique



Different Degrees of Generality

bound to one system:

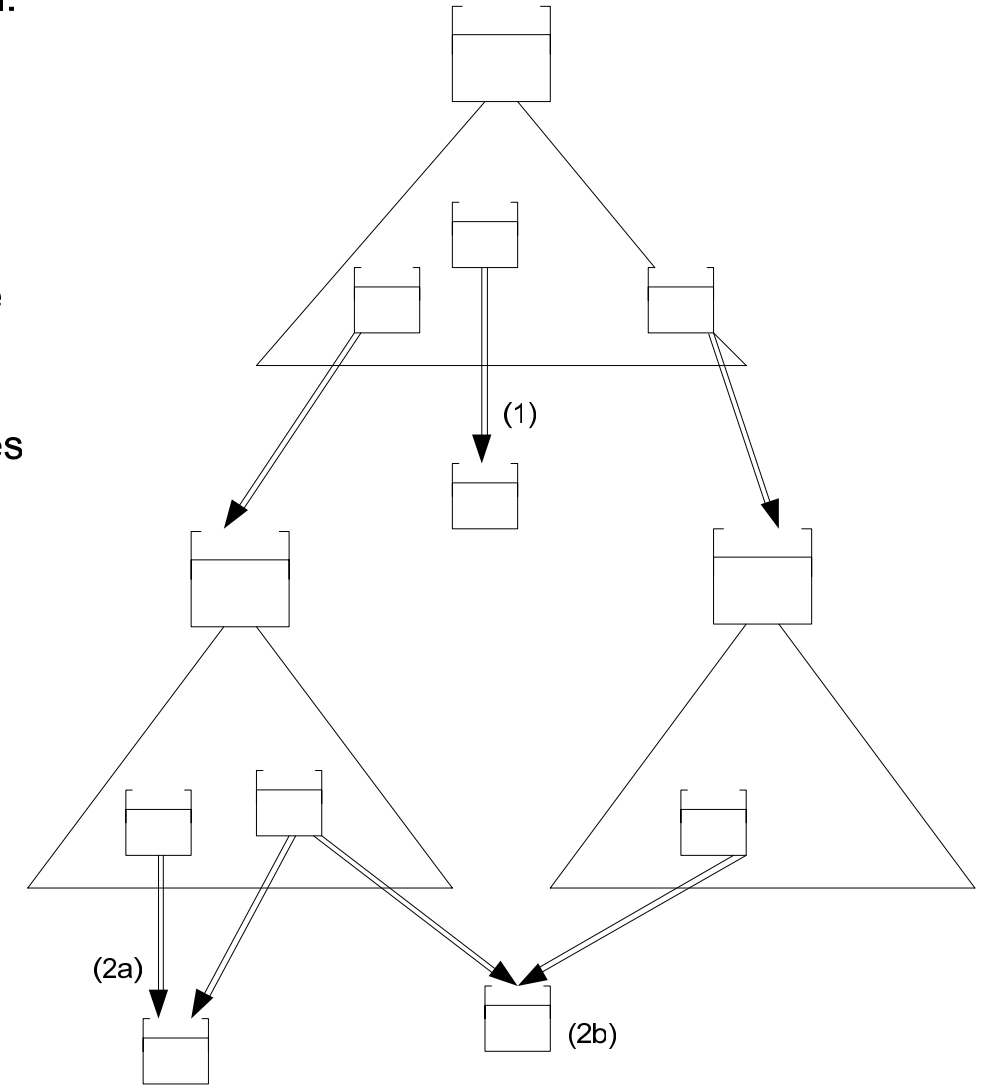
(1) one use
multiple use

(2a) in the same
subarchitecture

(2b) in different
subarchitectures

bound to different
systems

one process for
different systems



Is General Usability Enough ?

(in principle yes, see FORTRAN, Assembler etc.)

No, would like to express:

- component has only local importance
- need not know it from outside
- forbidden to use it
- then component library only contains reusable components

local usability: bound to underlying contains-relation

general usability: no underlying structure relation

(however general components are “ordered” in some library)

Aims of OO

- Modelling types and not single objects
- Modelling similarities between types
- Model processing of objects with similar types
- Achieving “reuse”

3 Concepts Necessary for OO

- extending interfaces (programming by extension): is_a – relation
- variant (or difference) programming
- uniform processing elements of similar type: dispatching

Distinction

- object-based: use of data abstraction: ados
- adt-based: use of adts
- object-oriented: use of inheritance and dispatching

Integration

- OO is one valuable design concept
- to be integrated with others (locality, general usability etc.)
- in literature OO is a separate world

Other Terminology in OO (e.g. Smalltalk)

object	is	abstract data object
class		adt module from which other classes may be derived
method		access operations of an adt module
message		call of access operation

Extending Interfaces

```
abstract data type module Employee is  --***************SUBCLASS*******************************  --
    a Person;                                                                                      --
    type Employee_T extends Person_T with private;                  -- further internal components --
    procedure Write_Bonus (E: in out Employee_T; B: in Currency_T);      -- further operations     --
    procedure Read_Bonus (E: in Employee_T; B: out Currency_T);                                    --
    procedure Calc_Salary (E: in out Employee_T; Sal: out Currency_T);  -- redefining operations  --
end Employee; ------------------------------------------------------------------------------------  --


abstract data type module Person is  --******************SUPERCLASS************************  --
    type Person_T is tagged private;                                                        --
    procedure Write_Name (P: in out Person_T; N: in Name_T);                                --
    procedure Read_Name (P: in Person_T; N: out Name_T);                                    --
    …                                                                                       --
    procedure Calc_Salary (P: in out Person_T; Sal: out Currency_T);                        --
end Personnel; -----------------------------------------------------------------------------  --
```

Variant Programming

```
module body Employee is  --*************************************************************************************** --
      procedure Calc_Salary (E: in out Employee_T; Sal: out Currency_T)is                                           --
            B, S1: Currency_T;                                                                                      --
      begin                                                                                                         --
            Calc_Salary (Person_T(E); S1);  -- make use of Calc_Salary of Person; program only the difference       --
            Read_Bonus (E;B);                -- in a further class Tariff_Employee the salary may even regard       --
            Sal :=  B + S1;                  -- different tariff groups etc                                         --
      end Calc_Salary;                                                                                              --
      ….                                                                                                            --
end Employee; ------------------------------------------------------------------------------------------------------ --


module body Personnel is  --************************************************************************************** --
      procedure Calc_Salary (P: in out Person_T; Sal: out Currency_T)is                                             --
      begin Sal := 1000;               -- base salary for all company members                                       --
     end;                                                                                                           --
      …                                                                                                             --
end Personnel; ----------------------------------------------------------------------------------------------------- --
```

Classes in OO Language Are Mostly Denoted Differently (As only this Kind of Module Exists in OO)

extension revisited (first parameter is missing):

```
class Employee is a Person;
    method Write_Bonus (B: in Currency_T);
    method Read_Bonus (B: out Currency_T);
    method Calc_Salary (Sal: out Currency_T);
    …
end Employee;
```

applying method to objects of a class with dot notation

```
method Calc_Salary (Sal: out Currency_T) is          -- of Class Employee --
    B; S1: Currency_T;
begin
    self.Person.Calc_Salary (S1);
    self.Read_Bonus (B);
    Sal := B + S1;
end Calc_Salary;
```

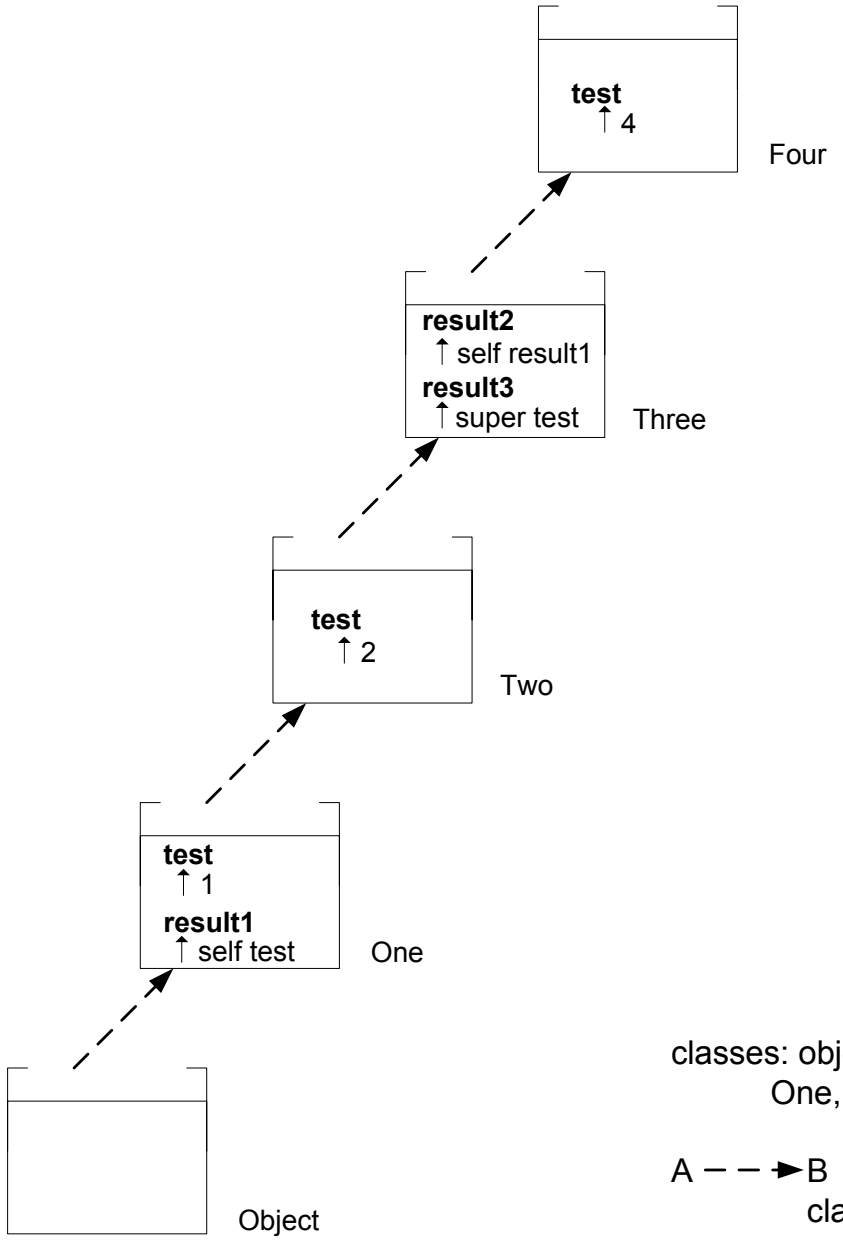
Solely Dynamic Binding in Smalltalk

a)
example1 ← One new.
example2 ← Two new.
example1 test. 1
example1 result1. 1
example2 test. 2
example2 result1. 2

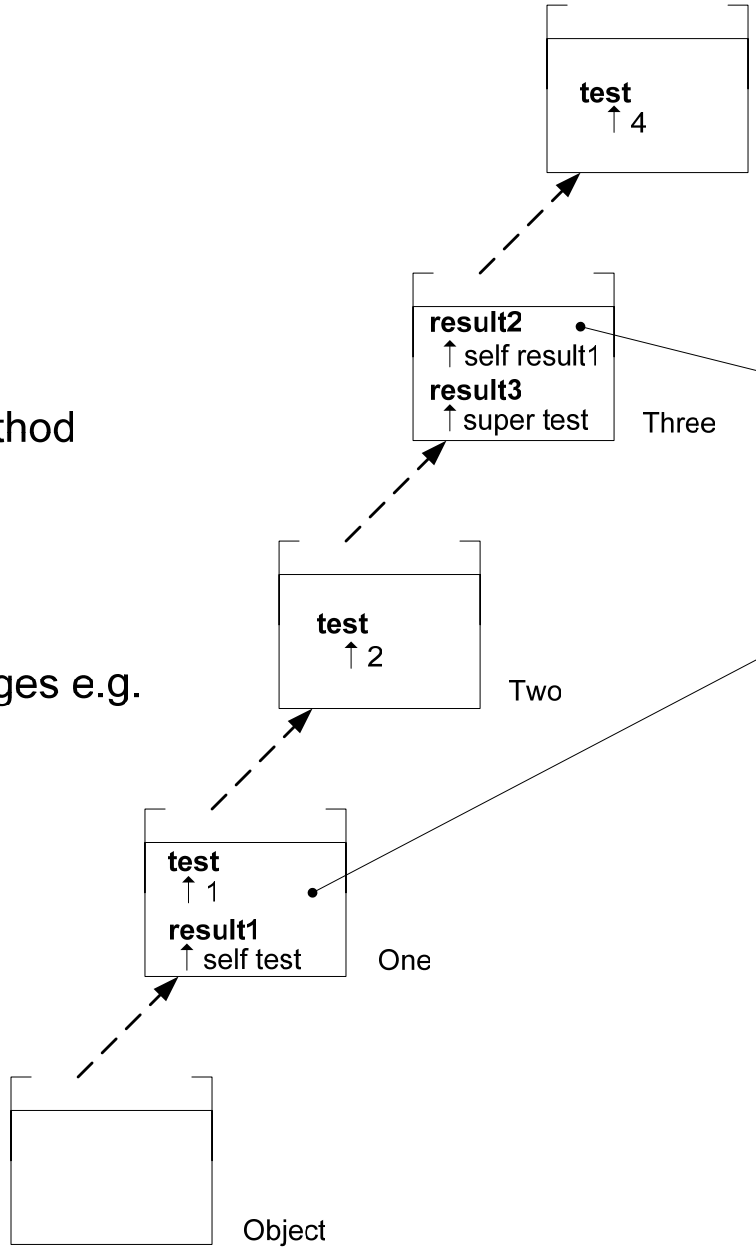
example3 ← Three new.
example4 ← Four new.
example3 test. 2
example4 result1. 4
example3 result2. 2
example4 result2. 4

b)
example3 result3. 2
example4 result3. 2

output results



this / self and super



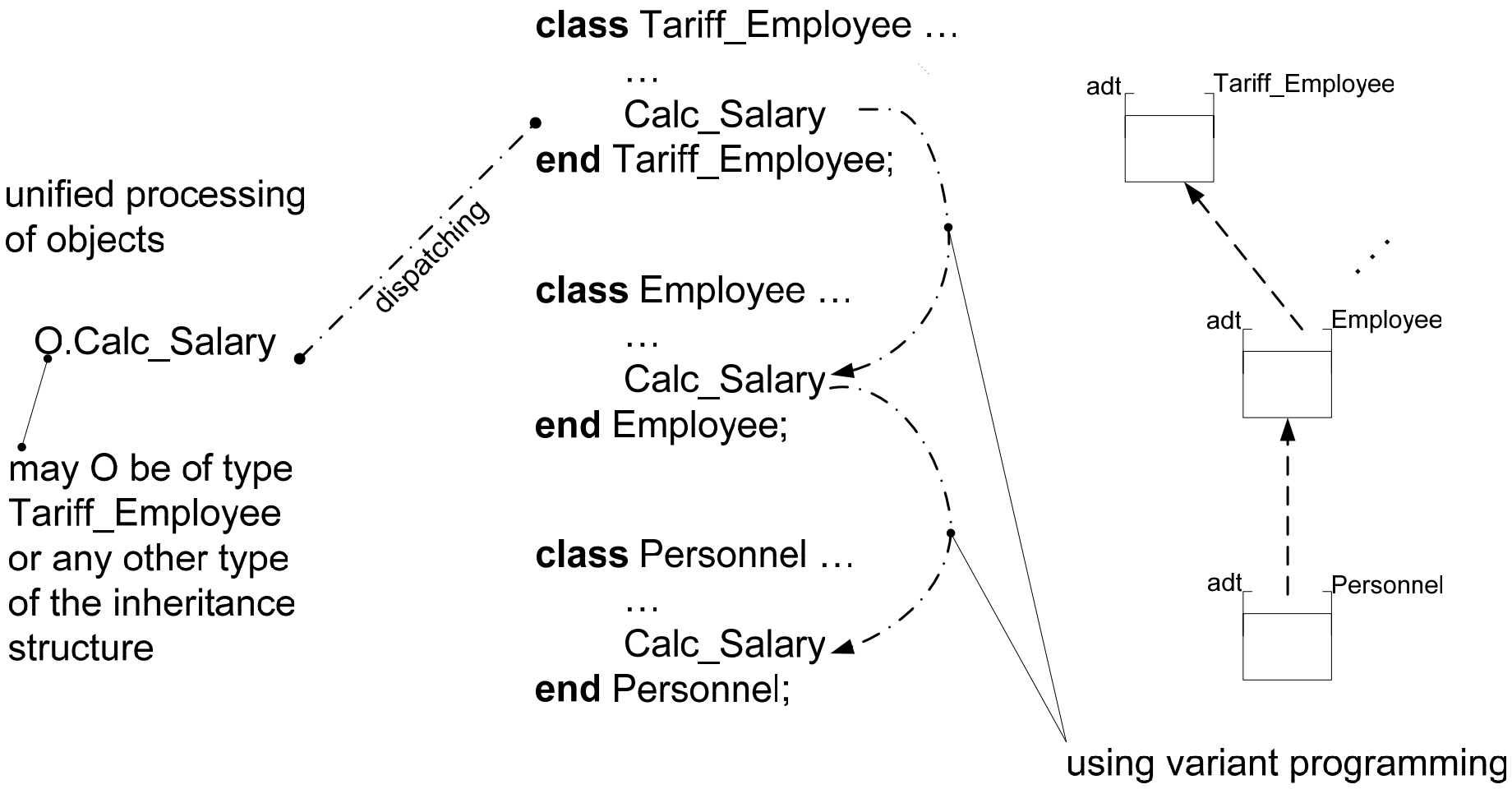
super: start method search in superclass

in other languages e.g. Two.test

To apply a method result1 / test to the complete object to which result2 / result1 was applied

All OO languages with special notation of method application have such a pseudo variable this, self etc.

Processing Different Objects via Dispatching



avoiding nested case - statements

Structure, Static vs. Dynamic Binding

- Type
  - » dynamic: every object at a certain time of program execution belongs to exactly one type (class)
  - » static: the type of an object is determined by a class together with its subclasses

- Binding
  - » static binding of method: at compile time (not everywhere possible)
  - » dynamic binding of method: at runtime (necessary for dispatching) in a certain part of hierarchy
  - » some languages (e.g. Smalltalk): everything at runtime, no static checks (doesNotUnderstand)
  - » some languages (e.g. Ada 95): mostly statical binding

Abstract Classes

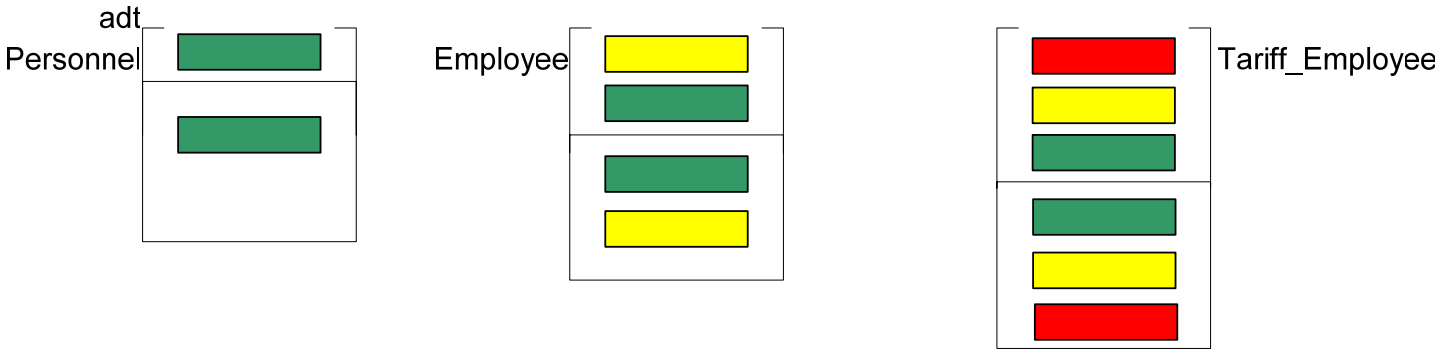
- abstract class has no objects
- “abstract” different from “abstract” in adts
- common protocol for an inheritance structure
- abstract procedures have no body

```
abstract data type module Person_Protocol is          -- common protocol        --
    type PP_T is abstract tagged;                     -- abstract type          --
    procedure Calc_Salary (…); is abstract            -- abstract operations    --
    …                                                 -- needed for dispatching --
end Person_Protocol;
```

advantage: uniform processing of objects can be formulated independent of class hierarchy

OO Means Classification (1)

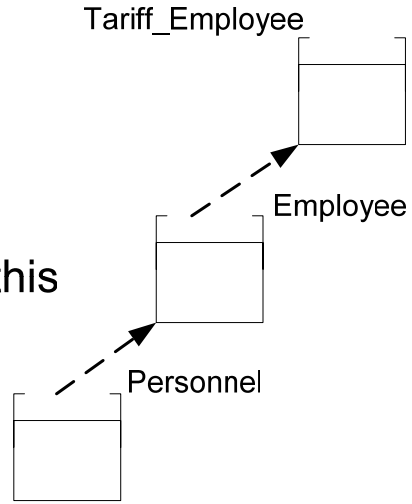
without inheritance: redundancy



Disadvantages:

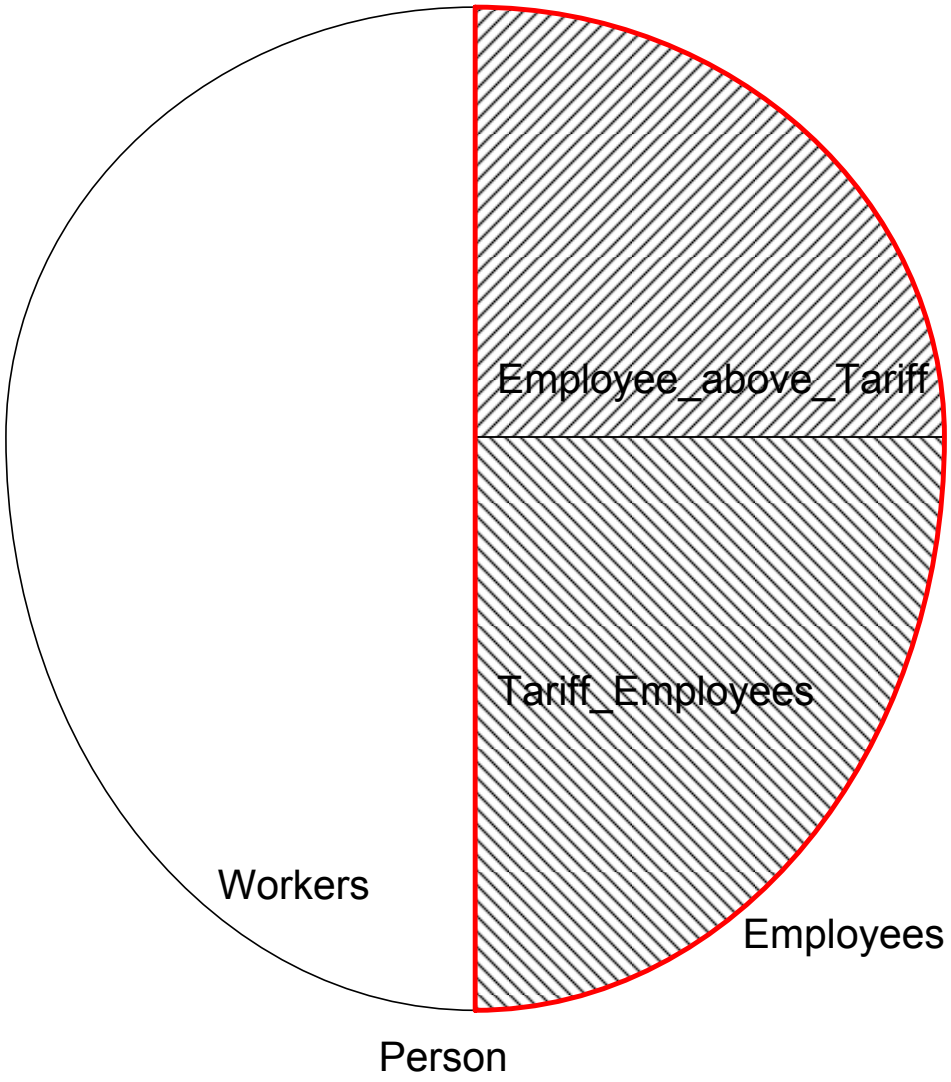
- redundancy in interfaces
- redundancy in realization
  - data declarations
  - corresponding operations code
- processing of elements of either type
  - case - statements

OO avoids this



OO Means Classification (2)

universe is partitioned: class of mathematics



- spezialization of a class: further refinement of partition
- uniform processing of all representatives of all classes via dispatching (class-wide programming, Ada 95)
- abstract class: no representatives

Person = Worker U Employees

no object of type Personnel (abstract class)

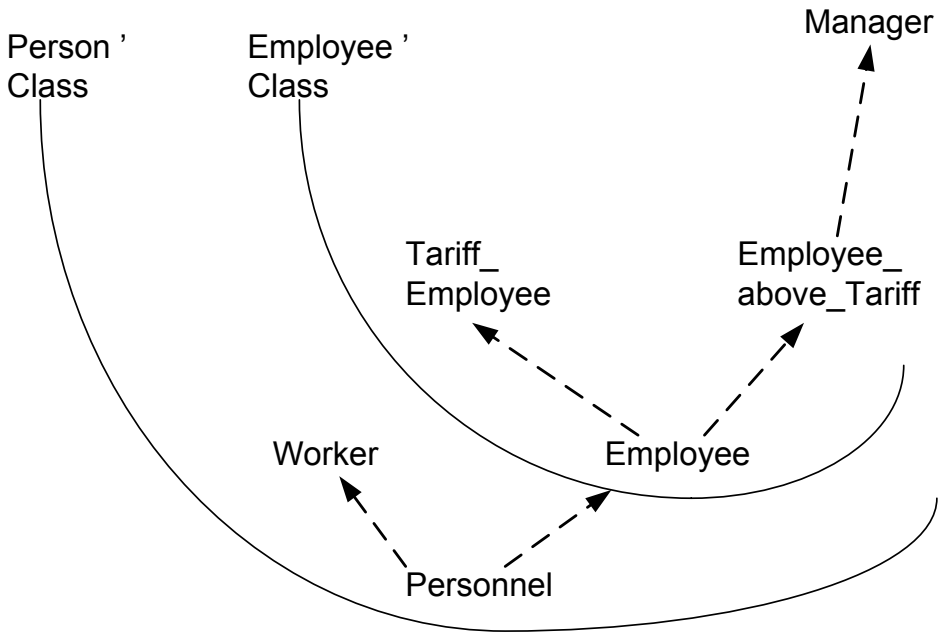
Term “class”

common OO languages

Ada 95: a class is a type subtree

object declared or generated of Type Personnel

at runtime object can also be of a subclass of Personnel

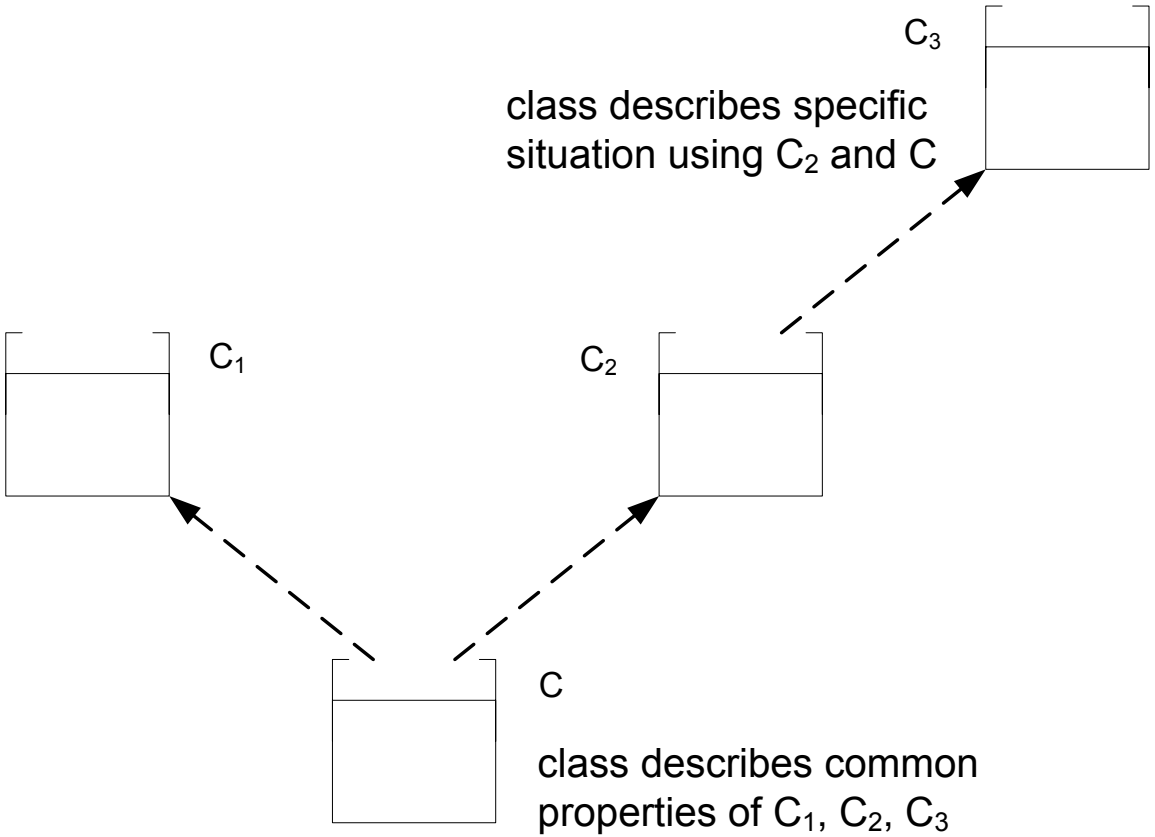


OO and Reuse

- extension: reusing interface of superclass, difference to specify
- variant programming: reuse of method code, difference to program
- dispatching / dynamic binding: common scheme for processing elements without regarding specific types (by case – statements)
- reusing classes for defining objects
- reusing class hierarchies by extending them specifically

(but: later discussion about limits of reuse)

Inheritance A - - ► B, Semantics



subclass C_3 inherits all properties of superclass C_2

properties are available for objects of C_3 if not redefined

objects of C_3 have further properties described in the interface of C_3

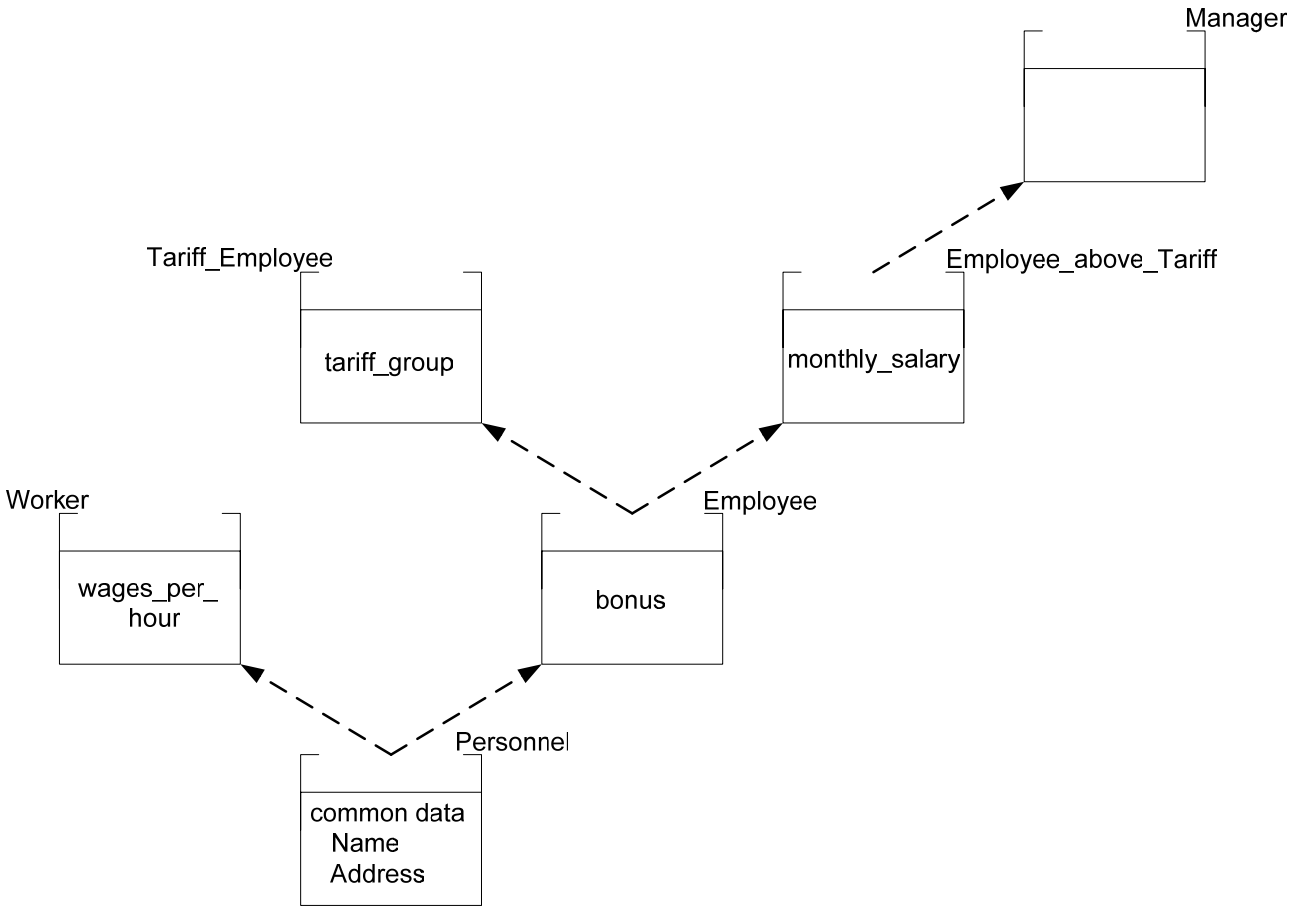
inheritance is transitive

Wording:

C_2 — ► C_3 C_3 is specialization of C_2, C_2 is generalization of C_3

Good use: only opaque types and extensions in interfaces

Example Inheritance Structure



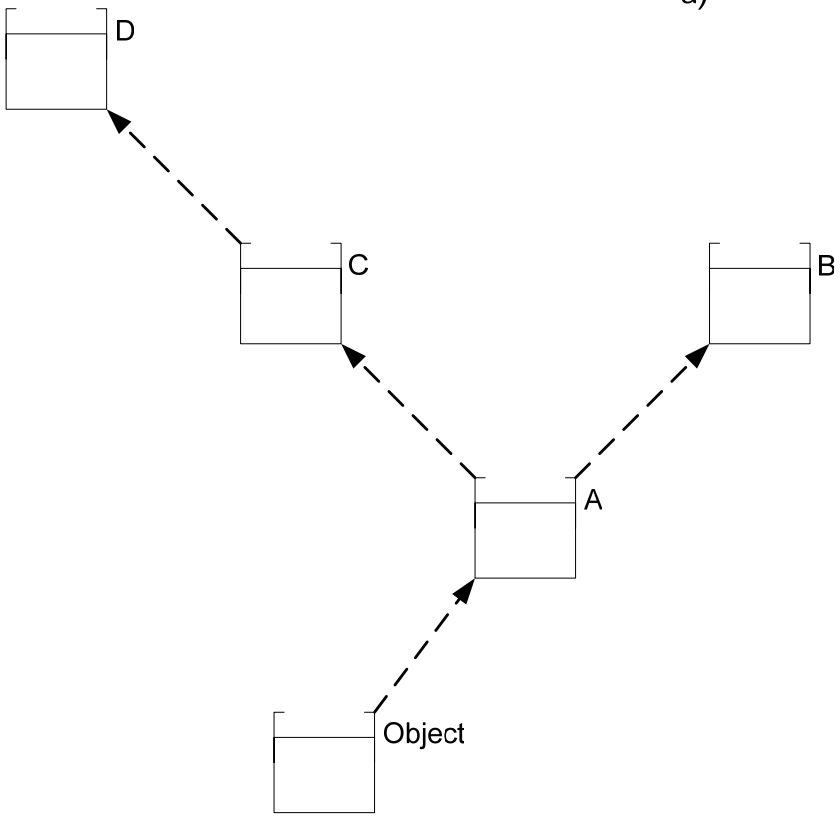
We paint trees downside – up:

- a subclass is more abstract, i.e. far away from basic machine
- inheritance is one concept for layering besides others

Single vs. Multiple Inheritance

single inheritance: trees

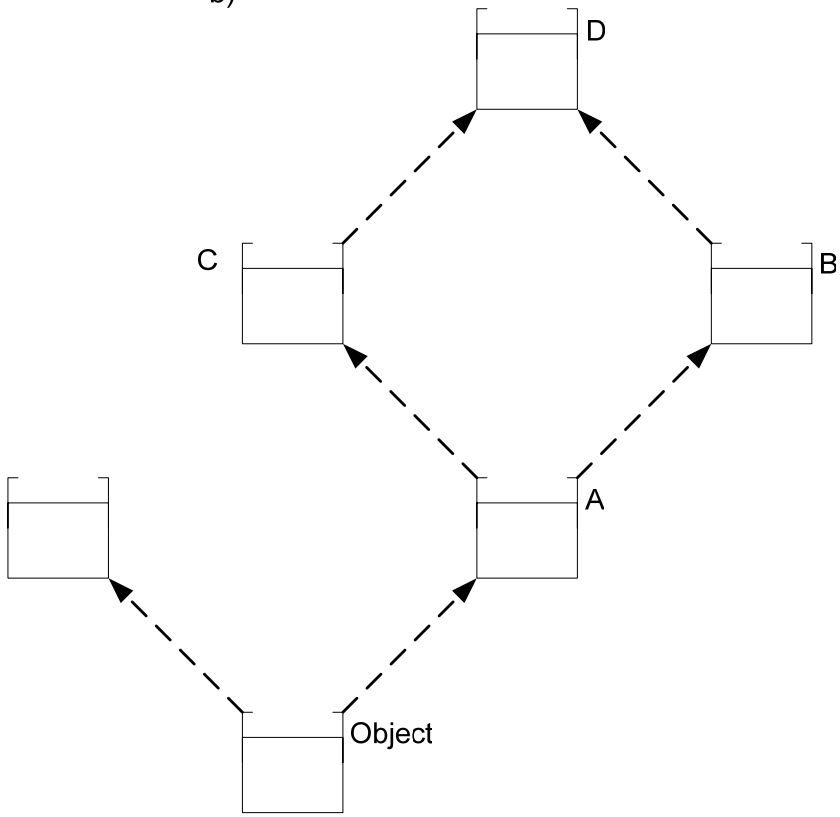
a)



regarded further

multiple inheritance: dags

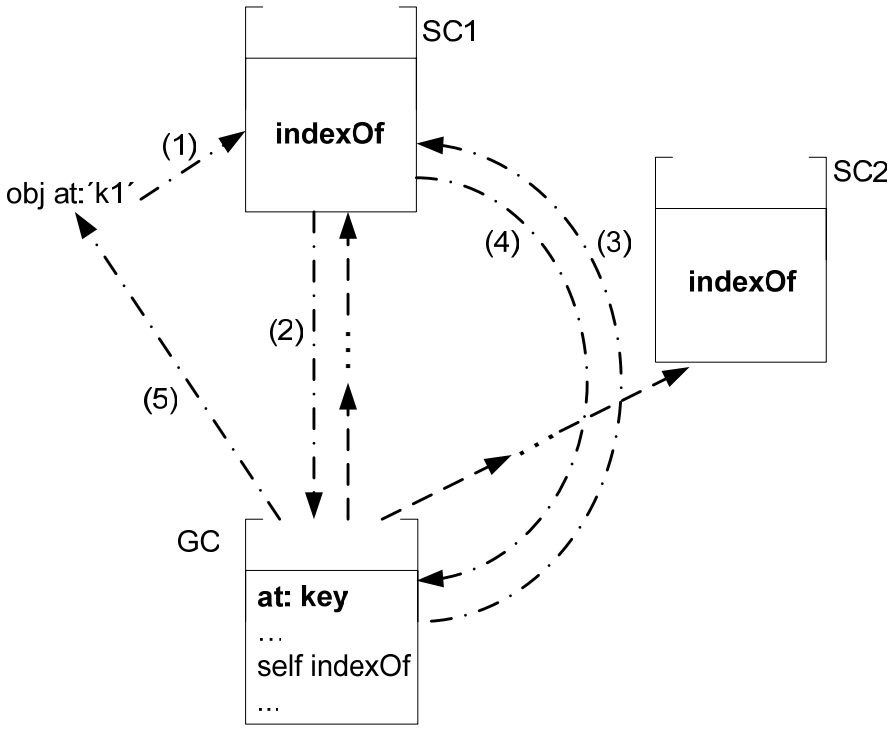
b)



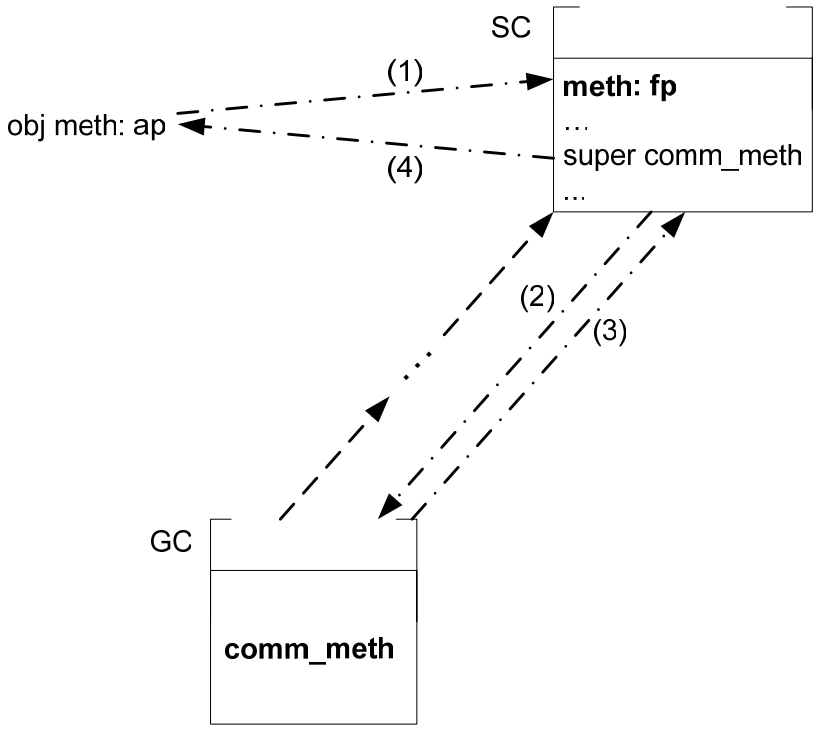
conflicts: resolution complicated

Calculation Mechanisms (Smalltalk)

specific method
by self from common method



common method
by super from specific ones



is_a - relation

method search / activation/
jump back to callee

similar mechanism also in other OO languages

Calculation Mechanisms (Ada 95)

specific method by dispatching

common method by variant programming

```
procedure Process_Person_Objects (O: Personnel' Class) is
    S := Currency_T;
begin
    Calc_Salary (O, S);
    …
end;
```

dispatching e.g. to Tariff_Employee

```
procedure Calc_Salary (O: in out Tariff_Employee_T; Sal: out Currency_T) is
    S1 := Currency_T;
begin
    Calc_Salary (Employee_T (O), S2);
    …
end;
```

general "methods" by variant programming

```
procedure Calc_Salary (O: in out Employee_T; Sal: out Currency_T) is
    S2 := Currency_T;
begin
    Calc_Salary (Personnel (O), S2);
    …
end;
```

```
procedure Calc_Salary (O: in out Person_T; Sal: out Currency_T) is
begin
    …
end;
```

OO is Bottom – up Development

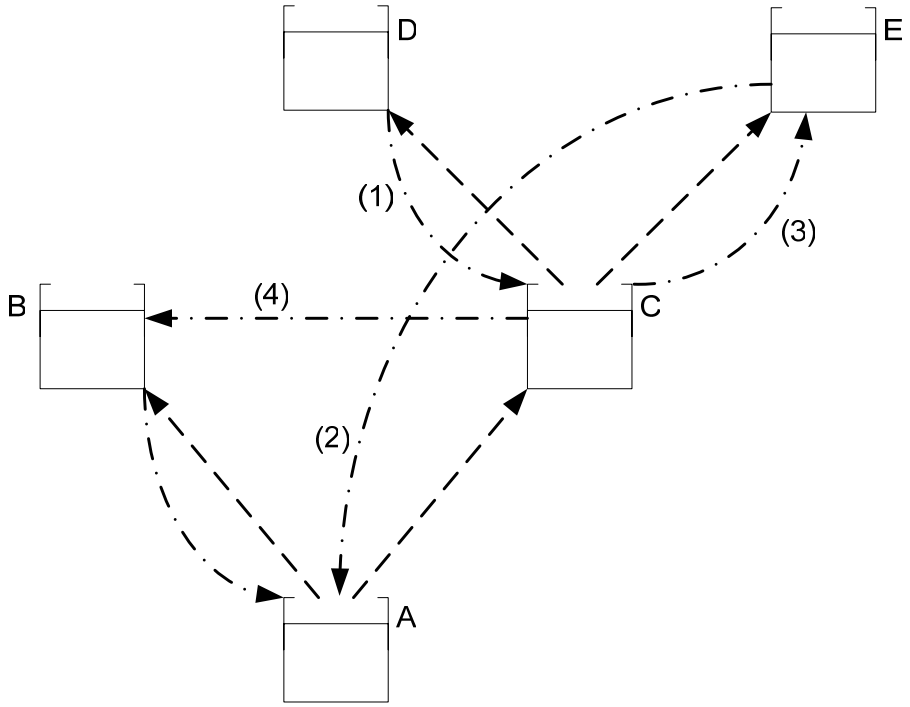
- have (a) given inheritance structure(s)
- system development means extending inheritance structure(s)
- thereby extending a reusable class hierarchy
- OO languages come with rich class hierarchy
  e.g. Smalltalk
    numbers
    data and file structure
    process administration
    graphics
    even control structures
    etc.

OO and Reuse: Difficulties

- make clear which reusable classes exist
- extend inheritance hierarchy
  (thereby existing classes may be changed)
- all hierarchies are open for extension, irrespective of systems under development belongs to
  - specific application domain
  - specific context within developing company
  - specific class of systems
- extensions may not be interesting for everybody:
  - environment pollution
- reuse demands for “worldwide” hierarchy administration:
  - unrealistic, so reuse is restricted to specific context, libraries with multiple development unavoidable
- if only specific classes are needed: how to get the specific subhierarchies
- melting different inheritance hierarchies: difficult

OO in Our Architecture Language

architecture diagram



structure relation
is a (specializes)

inheritance import
explicit
for invoking specific
methods (self) (3)
for variant programming
using (1), (2)

MIL

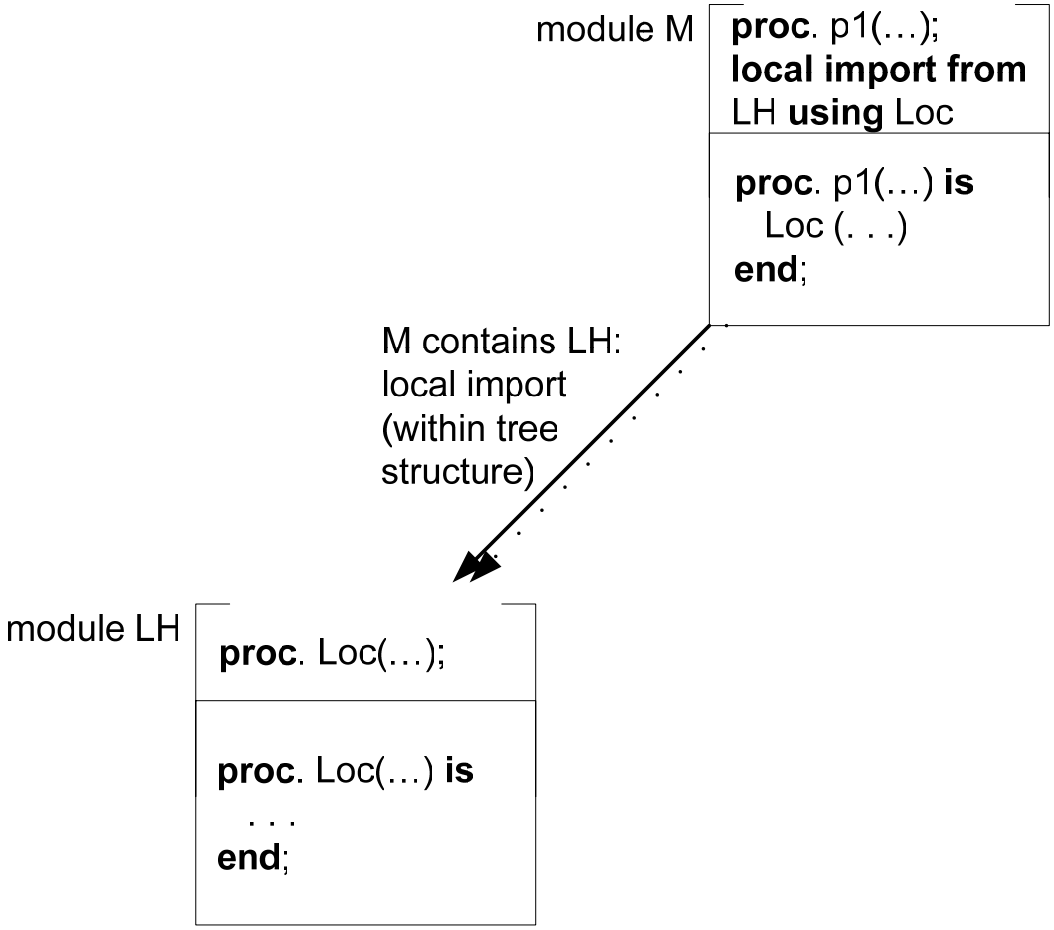
```
abstract data type E is                    -- E is a specialization of C
  is a C;
  type TE extends TC with private …
inheritance import from A using …          -- E needs resources from A
```

Many Usabilities within Class Hierarchies

- mirrors existing complexity
- makes design decisions explicit
- we see misuse of concepts, potential for improving a hierarchy
- delete a class: which problems arise
- insert a class: which protocol part to fulfil
  e.g. for self-like mechanism
- allows static analyses on complete hierarchy: variant programming, dispatching, self-like mechanism
- a tool should define abbreviations

OO vs. Locality

- OO: only adt modules
  only **is a** – relation: modelling similarities
- OO is the opposite of locality (local importance, local use)

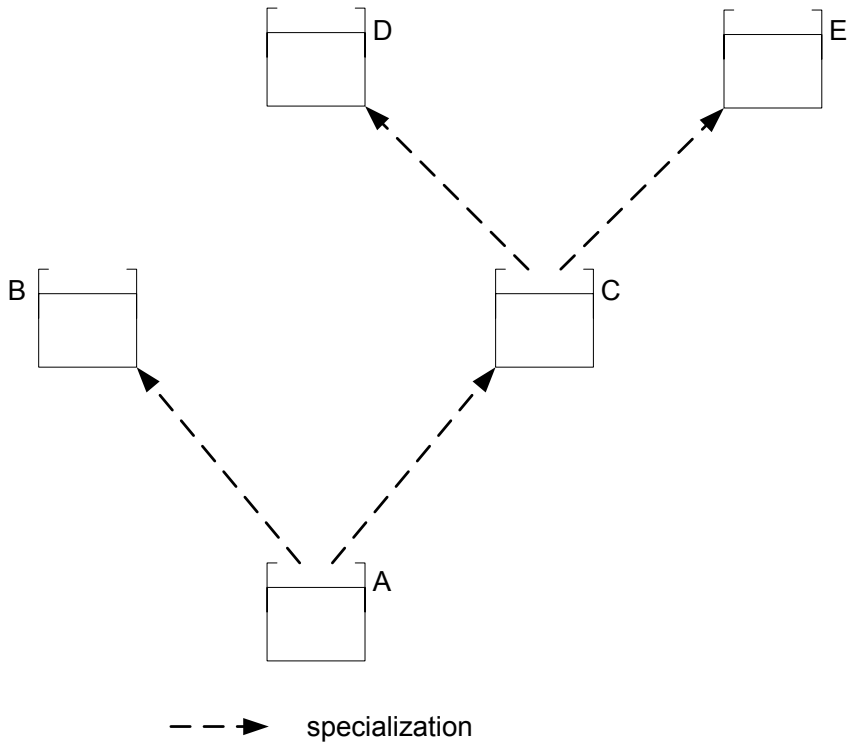


although we also have a tree – like structure relation (contains)

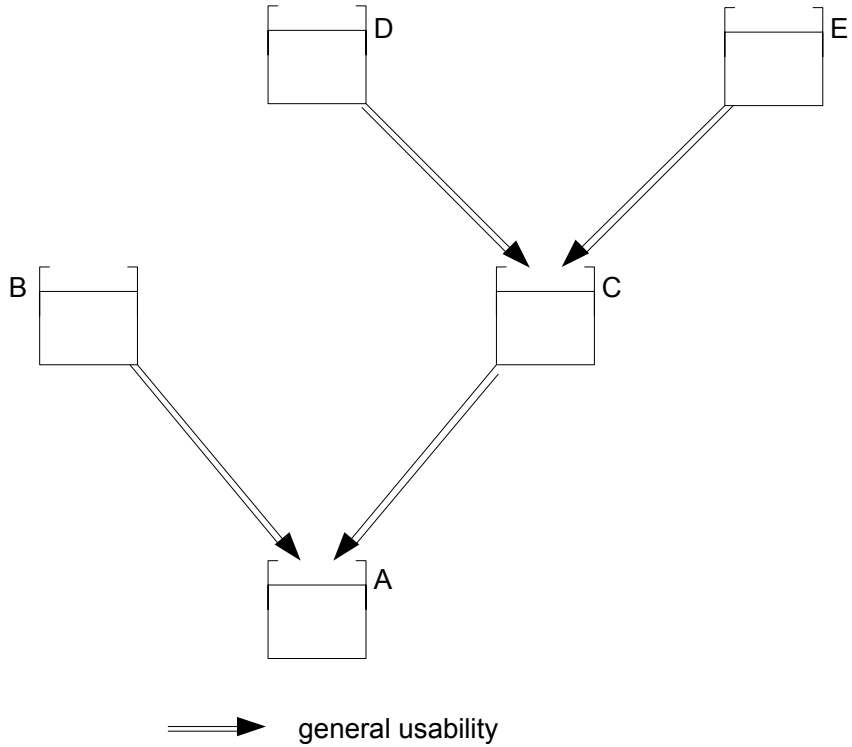
and imports only which are also „consistent“ to tree structure (local usability)

OO vs. General Usability: Looks Similar

inheritance structure



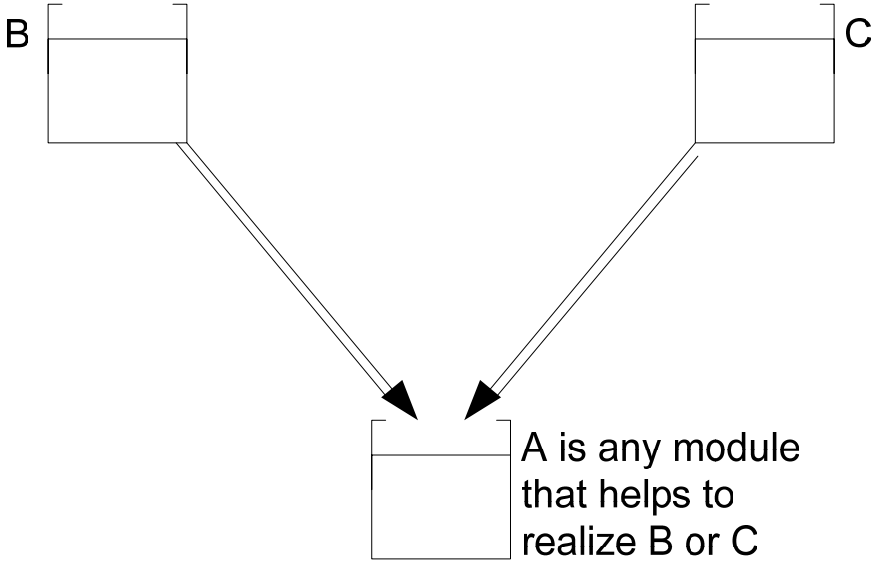
general usability layers



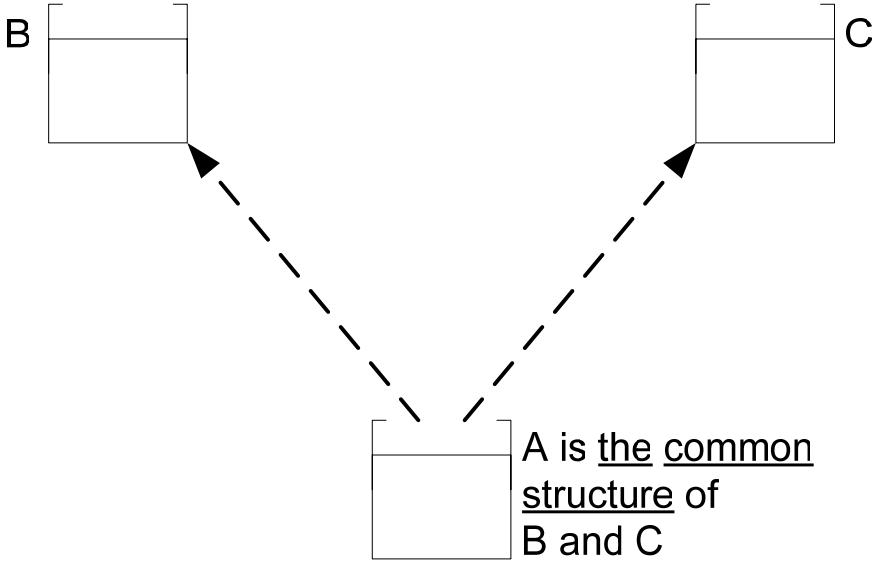
even “works“ for multiple inheritance

OO vs. General Usability: but Structure Differently

general usability layering

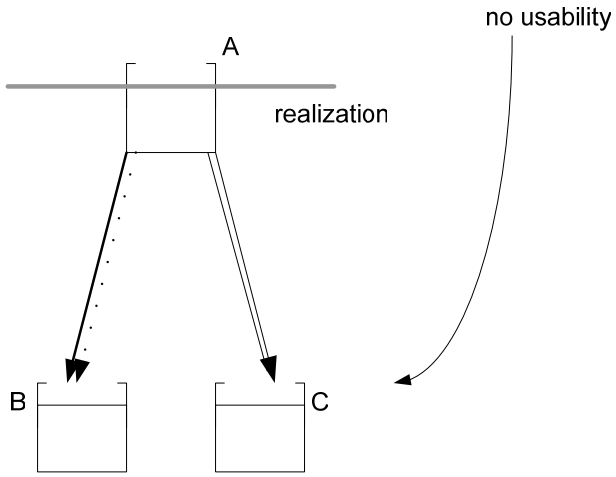


OO classification hierarchy

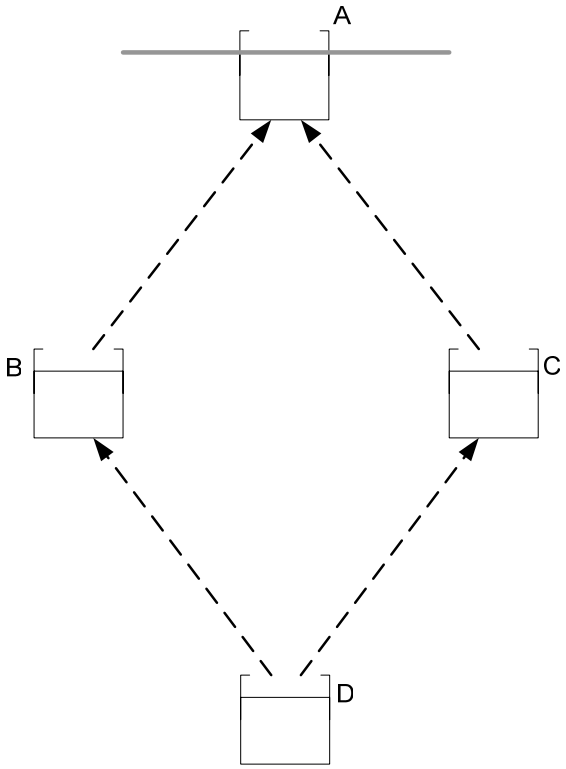


OO and Information Hiding

traditional architectures
can hide realization



OO architectures
let you see some part of realization



we are interested in using the interfaces
of B, C, D without copying them

we want to generally process all objects

Motivation

- Use a group of modules together with their realization
- Information hiding for a subarchitecture
- Allowing coarse design (producing larger design units)
- Big units for reuse
- Allowing subprojects (realizing a subsystem)
  - ⇨ architectural unit larger than a module for better
    - understanding an architecture
    - developing and maintaining a system
    - reuse
    - organizing a project

Requirements and Variety

Requirements

- modules of a subsystem are related to each other:
  internal tight coupling
- subsystems is independence from others (w.r.t. what is offers)
  external loose coupling

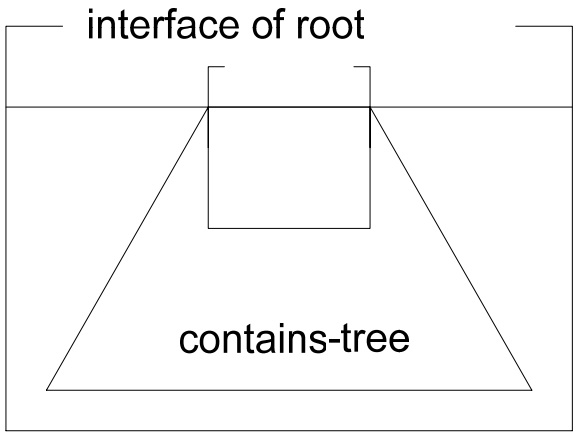
Variety

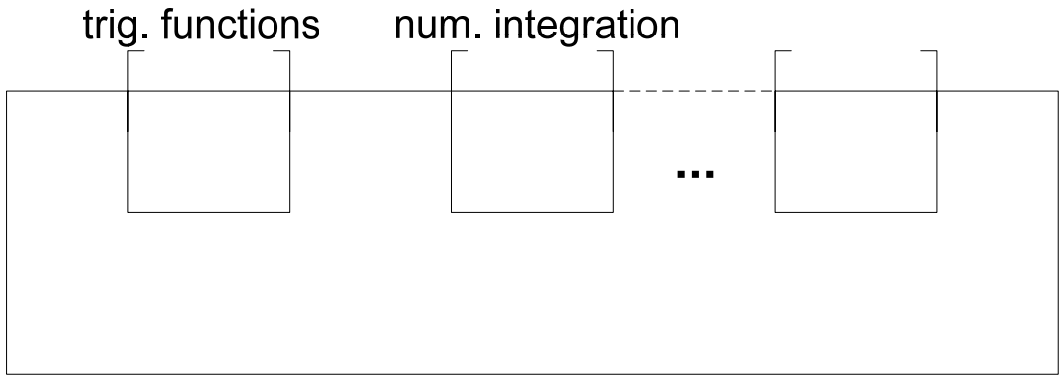
- what subsystem offers: how interface is aggregated
- how it is realized: “any” architecture inside

Two first and simple Examples

a) contains-tree
made invisible

b) aggregated needed
interfaces from library





Rough Definition

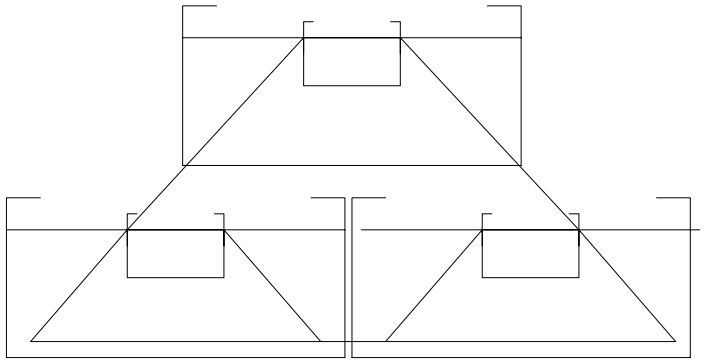
"A subsystem is a collection of logically related modules put into a new coarser unit. (1) It is determined which (resources of which) modules build up the interface of the subsystem. (2) The internal subarchitecture is hidden."

(1) export interface of the subsystem

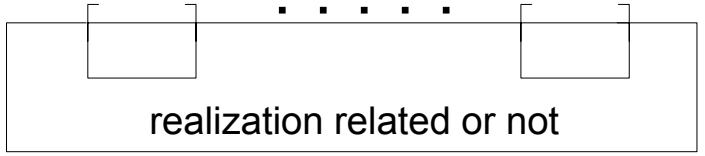
(2) body of the subsystem

Typical Examples & Classification

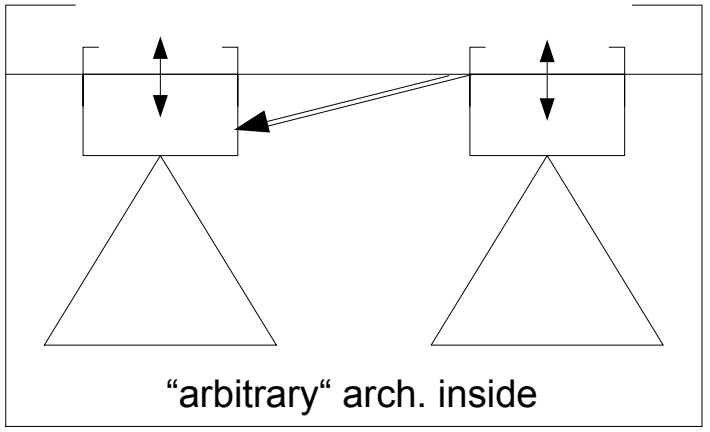
- subtrees of contains tree
- complete contains tree



- aggregation of related interfaces

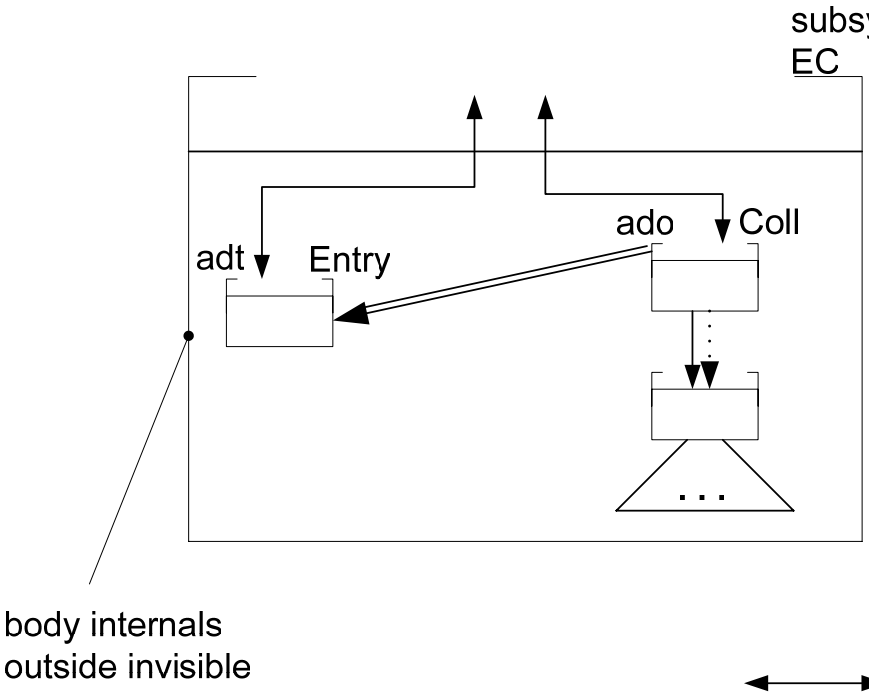


- arbitrary subarchitecture with distinction between outside and inside

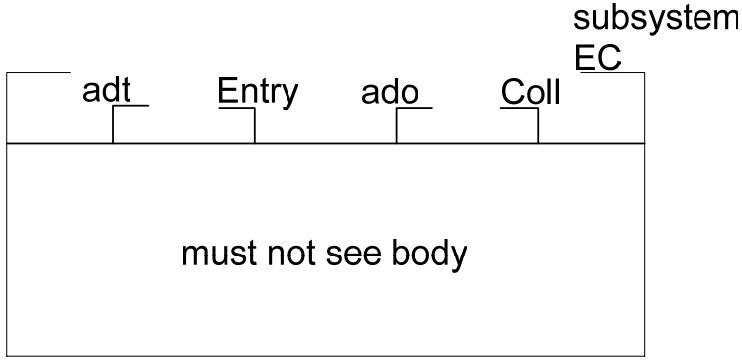


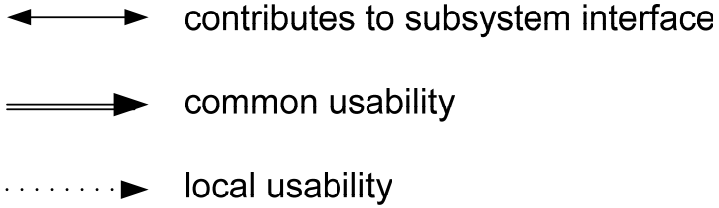
Graphical Notation

designers view



client´s view





MIL Notation for Subsystems

```
-- produced by designer for clients                                                                    --
subsystem EC is  --*************************************************************************************** --
    -- interface of subsystem EC built up by interfaces of E and C                                     --
    export from abstract data type module E is                                                          --
        type ET is private;                                                                             --
        procedure op_j (F0: ET, …);                                                                     --
        …                                                                                               --
        -- the semantics of interface operations is …                                                   --
        -- … .                                                                                          --
    end E; ----------------------------------------------------------------------------------------------- --
    export from abstract data object module C is  -- collection                                         --
        procedure op_j (…);                                                                             --
        …                                                                                               --
        -- the semantics of interface operations is …                                                   --
        -- … .                                                                                          --
    end E;                                                                                              --
end EC; ---------------------------------------------------------------------------------------------------- --

-- realized by designers and implementors                                                              --
subsystem body EC is  --************************************************************************** --
    general import from abstract data type module Sub1 using M1, M2, Sub2 using M3                      --
        -- all components of the body in usual notation                                                 --
end EC; ---------------------------------------------------------------------------------------------- --
```

Subsystems and Design Process

architecture contains subsystems and modules

subsystems contain subsystems and modules

. . .

Mostly, two-level design suffices:

(1) overview architecture

the whole picture

overview design

(2) designing (and implementing subsystems)

big component design

thereby also defining modules

Module vs. Subsystem: Analogy and Difference

Analogy:

- export interface
- export interface is part of the body
- body hidden
- body allows different structuring principles
  - global and local resources
- import interface introduces resources of other components

Difference:

- subsystems have aggregated interfaces
- subsystems realization includes design and later implementation
  - modules are immediately implemented
- subsystems introduce hierarchical design, modules flat design
  - however, modules appear on any design level

We Need a Construct for Subsystems (Different from Modules)

- understandability:

  (1) differences of layers

  (2) understand imports of a subsystem

- designers expresses:

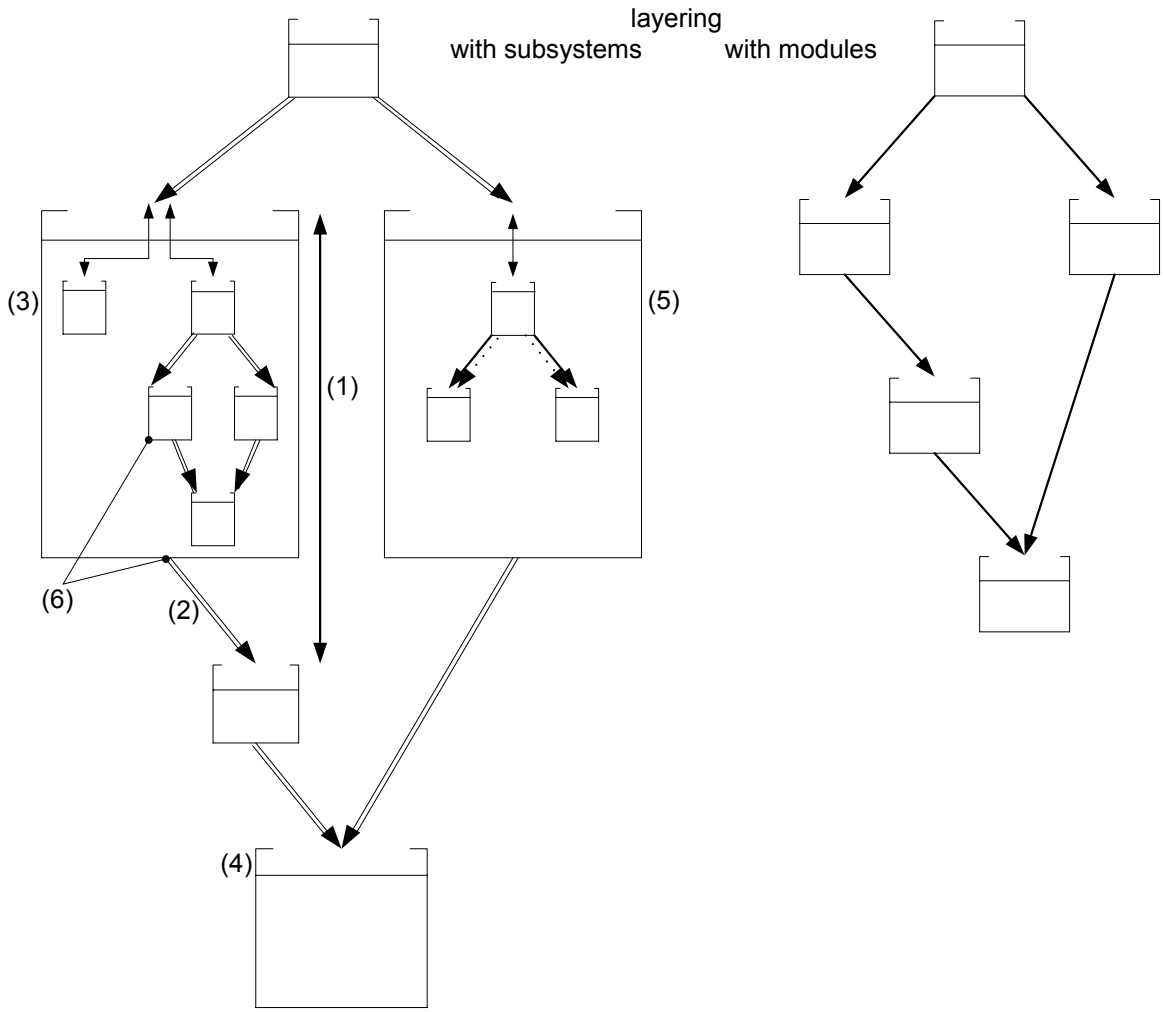
  (3) subarchitecture which was made to a subsystem

  (4) complex basic component

  (5) a new complex component has to be designed

- consistency:

  (6) consistency export / import can only be checked when regarding architectures of subsystem bodies



Reuse and Adaptability of Components

- Reuse demands adaptability
  - » of component to be reused
  - » of context into which component is to place

- Up to now for reuse:
  - » distinction interface and body of modules
  - » different realizations for one interface
  - » functional / data abstraction as semantical information hiding
  - » locality, layering
  - » subsystems

Idea of Genericity (Ada 83)

```
generic                                                          --  generic part                    --
      SIZE: NATURAL;                                             -- generic parameters to            --
      type TYPE_ITEM is private;                                 -- be fixed later                   --
package G_ITEM_STACK is -- ********************INTERFACE*************************************************  --
            procedure PUSH (X: in TYPE_ITEM);                    -- interface                        --
            procedure POP;                                       -- as usual: …                      --
                                                                 -- using generic parameters          --
            function READ_TOP return TYPE_ITEM;                                                      --
            function IS_EMPTY return BOOLEAN;                                                        --
            function IS_FULL return BOOLEAN;                                                         --
            ST_UNDERFLOW, ST_OVERFLOW: exception;                                                    --
end G_ITEM_STACK;-------------------------------------------------------------------------------------------  --

package body G_ITEM_STACK is -- ********************BODY*****************************************************  --
            SPACE: array (1..SIZE) of TYPE_ITEM;                                                     --
            INDEX: INTEGER range 0..SIZE;                                                            --
            procedure PUSH (X: in TYPE_ITEM) is begin … end;                                         --
            …                                                                                        --
            function IS_FULL return BOOLEAN is begin … end;                                          --
            ST_UNDERFLOW, ST_OVERFLOW: exception;                                                    --
end G_ITEM_STACK;-------------------------------------------------------------------------------------------  --

-- generation of a generic instance                                                                  --
package INTEGER_STACK is                                                                             --
      new G_ITEM_STACK (SIZE => 200, TYPE_ITEM => INTEGER);                                          --
-- can be modified by INTEGER_STACK.PUSH (…)                                                         --
-- INTEGER_STACK is an ado, G_INTEGER_STACK is a generic ado                                         --
```

Genericity and Parameterization

- Parameterization
  generic unit is a template
  generic parameters are “screws”
  are fixed for every generic instance
  generic unit represents a set of similar units
- Generic parameters are used for
  a type from a “class” of types
  a procedure from a “class” of procedure (par. type profile)
  dimensioning
- Generic units are reusable in a new sense
  generic unit only designed and implemented once
  is a template for many different components
  simple way to get a specific component (instantiation)
  reuse²: generic unit → specific unit → bind into system

instantiation reuse

Genericity is a compile-time mechanism

Genericity (as we understand) has a new quality:

- generic unit is no architectural unit (a template)
- by instantiation we get a unit
- genericity corresponds to design process rather than design results

Genericity and concepts of programming languages:

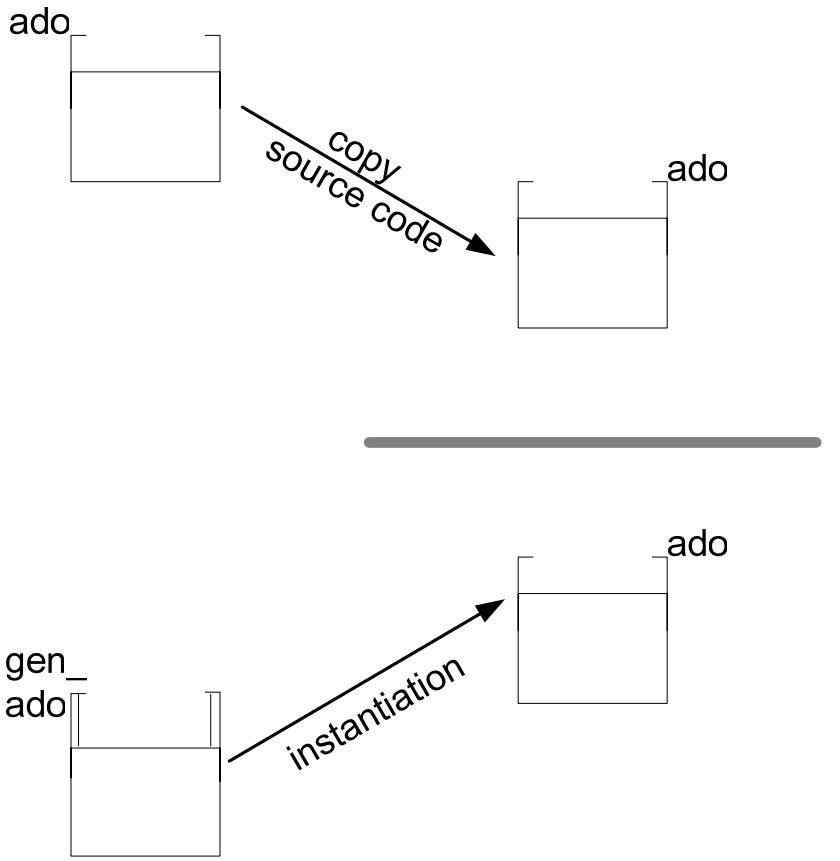
- macro mechanism: only instances are later checked (C)
- generic units and instantiations are checked (Ada):
  compile-time mechanism, strong typing
- run-time mechanism: formal type and formal procedure parameters
  (PL / I, Algol 68)

Different mechanism in one language, e.g. Ada 95:

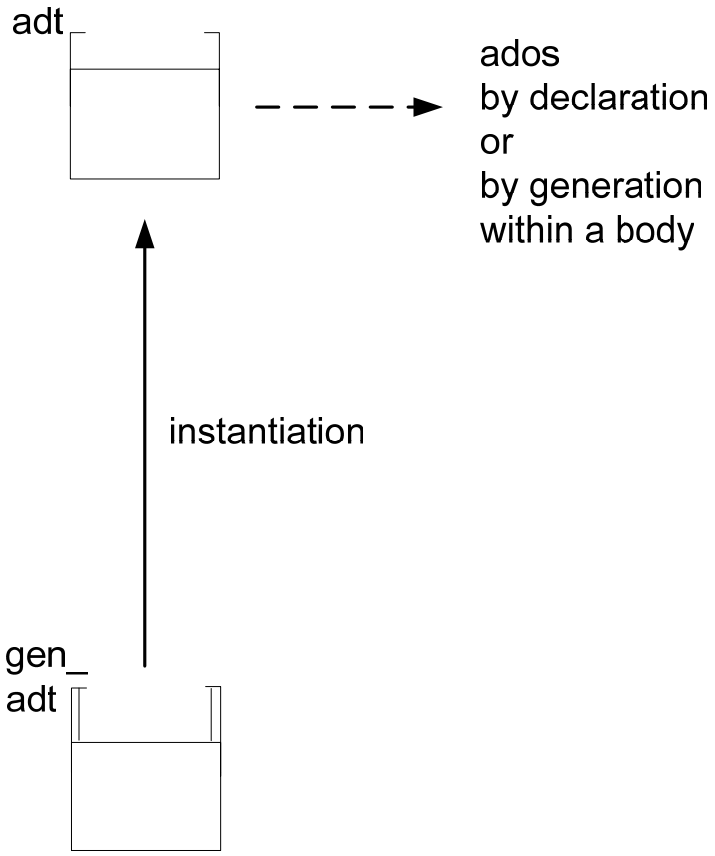
dimensioning	→	private types with discriminants	runtime mechanisms
procedures	→	pointers to procedures	
types	→	genericity	compile-time mechanism

Ados in a System

ados in the architecture

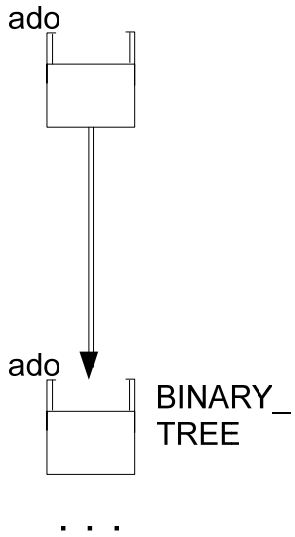


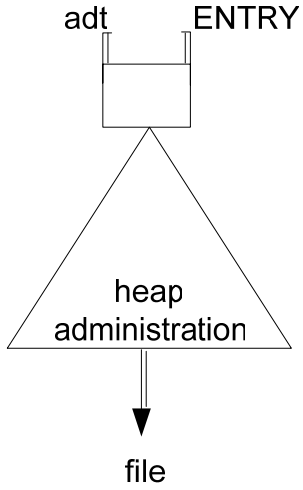
adts in the architecture
(ados generated in the bodies of clients)



Generic Subarchitectures and Subsystems

subarchitectures

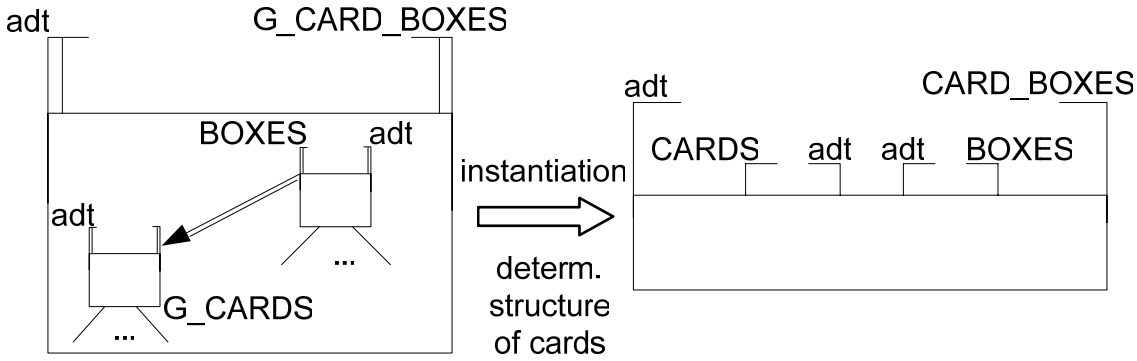




generic Parameters:
key type
info type

generic Parameters:
entry type
and its
components

subsystems



Textual Notation for a Generic Subsystem

```
generic     -- ********************************************************************************************************** --
      type Key_T, Info_T is private;                    --  operations for these opaque types                          --
-- data type                                                                                                            --
subsystem G_Card_Boxes is -- ************************************************************************************** --
      …                                                                                                               --
      export from abstract data type module Cards is                                                                  --
            type Card_T is private                                                                                    --
            procedure initialize_Card (Card: out Card_T);                                                             --
            procedure delete_Card_Info (Card: in out Card_T);                                                         --
            procedure set_Key (Card: in out Card_T; Key: in Key_T);                                                   --
            function read_Key (Card: in Card_T) return Key_T;                                                         --
            procedure set_Info (Card: in out Card_T; Info: in Info_T);                                                --
            function read_Info (Card: in Card_T) return Info_T;                                                       --
            …                                                                                                         --
      end Cards;-- ------------------------------------------------------------------------------------------------- --
      export from abstract data type module Boxes is                                                                  --
            type Box_T is private                                                                                     --
            procedure deposit_Card (Card: in Card_T; Box: in out Box_T);                                              --
            procedure delete_Card (Card: in Card_T; Box: in out Box_T);                                               --
            …                                                                                                         --
      end Boxes;-- ------------------------------------------------------------------------------------------------- --
end G_Card_Boxes; -- ------------------------------------------------------------------------------------------------ --


subsystem body G_Card_Boxes is -- ********************************************************************************* --
      ...                                                                                                             --
end G_Card_Boxes; -- ------------------------------------------------------------------------------------------------ --
```

Architectures: Representation and Understanding

- Necessary for understanding a design:
  architecture diagrams: for overview
  textual description of modules and subsystems: for details
  design rationale for reasoning and explanation
  "spec" of module bodies
  … (semantics, distribution, concurrency, system control: later)
- Architecture diagrams:
  overview to understand a complete system or its subsystems
  to discuss their structure (here locality structure, there subsystem or inheritance hierarchy
  to detect reusable components, local or global reuse "patterns"
  experienced designer makes only diagrams in the first steps
  big system: overview diagram, subsystem diagrams

  ⇨ contain essentials of the system to construct / maintain

Architecture Diagram Language

- Components
  modules: f, ado, adt (later ft)
  subsystems

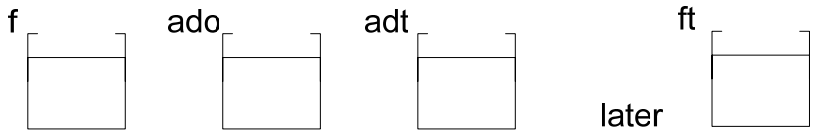
  generic components and their instantiations

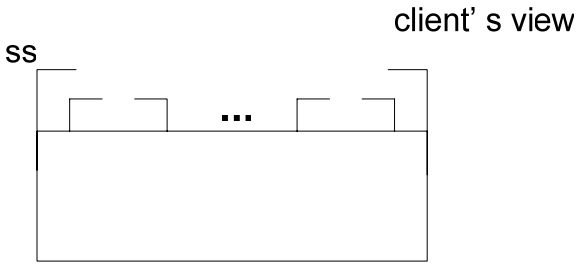
- Relations
  structure relations: contains, is – a
  import relations: local import, general import, inheritance import

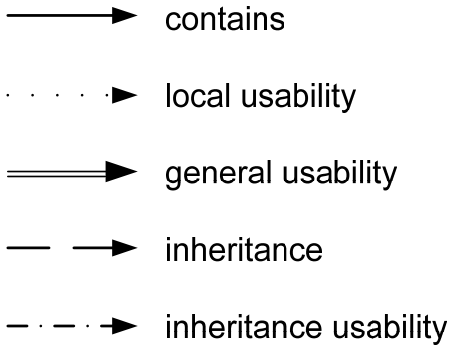
- Consistency conditions (syntactical restriction): next section

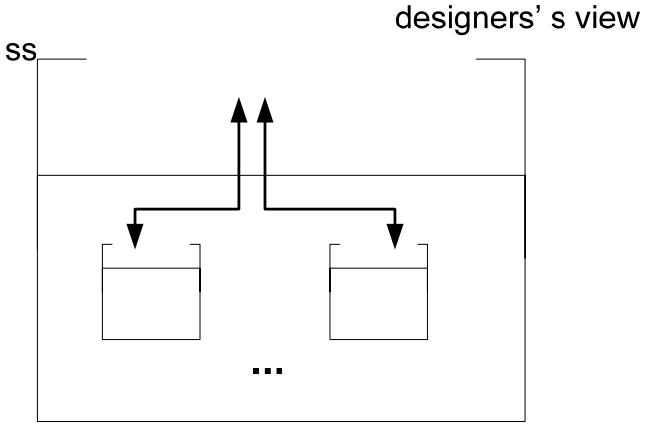
  ⇨ class of directed and labeled graphs in graphical representation

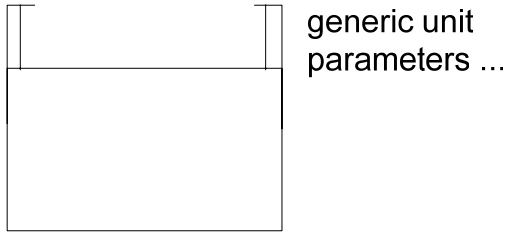
Diagram Language Elements











a formal definition of context-free as well as context-sensitive syntax can be given by graph grammars

Detailed context-free Syntax Description (MIL: Module Interconnection Language)

component_description ::=	module_description | subsystem_description
module_description ::=	interface_description body_description
interface_description ::=	module_kind **module** *module*_name **is** [structure_clause] [*module_global*_imports] export_interface [module_semantics] **end** *module*_name;
body_description ::=	**module body** *module*_name **is** [*module_body*_imports] programming_in_the_small_part **end** *module*_name;
module_kind ::=	**functional** | **abstract data object** | **abstract data type** | **function type**
structure_clause ::=	**is contained in** *module_or_subsystem*_name; | **is a** *module*_name;
imports ::=	{ import_kind **import from** *module_or_subsystem*_name **using** *interface_resources*_name_list; | **using all**; }
import_kind ::=	**local** | **general** | **inheritance**
export_interface ::=	proc_or_func_or_limited_private_type_or_tagged_private_type_or_extends_type
module_semantics ::=	*semantic_description*_comment
progrmming_in _the_small_part ::=	*Ada*_source_text
comment ::=	*natural_language*_text
name_list ::=	Ada_identifier_list

MIL contd.

subsystem_description ::=	subystem_interface subsystem_body
subystem_interface ::=	**subsystem** *subystem*_name **is**
	[structure_clause]
	[*subsystem_global*_imports]
	subsystem_export_interface
	end *subsystem*_name;
subsystem_export_interface ::=	{interface_description}
subsystem_body ::=	**subsystem body** *subsystem*_name **is**
	[*subsystem_body*_imports]
	realization_part
	end *subsystem*_name;
realization_part ::=	[Ada_declarations] { *module*_body_description }
generic_components ::=	generic_clause component_description
generic_clause ::=	**generic** { generic_parameter }
generic_parameter ::=	Ada_like_parameter_description

Intro

- consistency constraints = context sensitive syntax of architectural (diagrammatic and module interconnection) languages and their relation to programming language (for bodies)
- no method rules → next chapter
- classification 1 of CC:
  allowed or unallowed architectures
  incomplete descriptions to be completed, otherwise unallowed
  warnings due to design errors
- ✓ classification 2 of CC:
  to be seen on graphical level (architecture diagram language)
  or on level of detailed specs (MIL)
  or if regarding realization of bodies
- for discussion we need:
  explanation of usability
  explanation of f, ado, adt modules
  explanation of locality, general usability layers, inheritance

Consistency Constraints for Architecture Diagrams

(1) A component (module, subsystem) is a “contained” in at most one component, i.e. contains relations (edges) build up a forest. Therefore, a component can only be the target of one contains relation.

(2) A local usability edge has to be consistent with scope / visibility. For example, an edge from a grand parent to grand child node is not allowed.

(3) Cyclic local usability is allowed and necessary for modelling recursive situations. On such a cycle no ado is allowed.

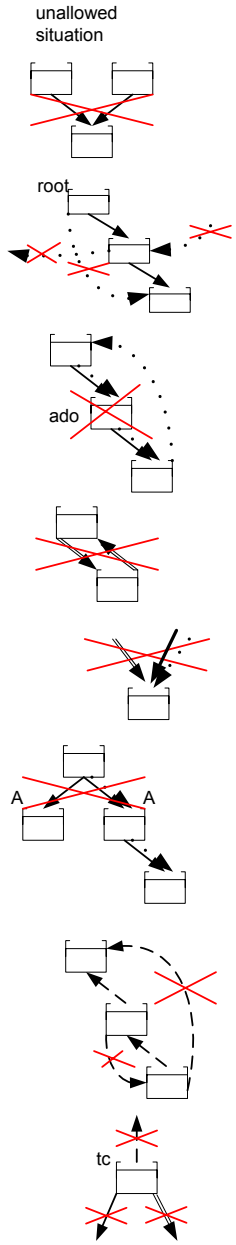
(4) General usability is acyclic, i.e. a component cannot import itself directly or indirectly.

(5) A general component cannot be a local component at the same time. So, it is the target of either a general or a local usability edge

(6) The names of all general usable components of a system are different. The same is true for components within a contains-tree. (This can easily be achieved: Version1_Bintree, Version2_Bintree; Bintree_quick, Bintree_efficient)

(7) The inheritance hierarchy is either a tree (single inheritance) or a dag (multiple inheritance). So, is – a is a strict hierarchy.

(8) A type collection module is not realized by using other components, i.e. it is atomic.



Export Consistencies (MIL Level)

(9) The export interface is consistent with its kind. A f module contains only proc or func specifications, an ado only access operations, an adt an opaque type and access operations or a creation / deletion together with access operations.

(10) Exported resources' names of one component are pairwise disjoint or have a different parameter profile (overloading). The formal parameter names within a single proc or func are pairwise disjoint.

(11) A f module has the same IO parameter profile for all resources. Alternatively, the I or O can be given by an import (see f module for drawing functions).

(12) Ado interface: if the ado is to be used more than once, there has to be an initialization operation. Any "dangerous" change operation has a security query and exception, or a corresponding return parameter.

(13) Adt interface: an adt with variable semantics has an opaque type T with at least one access operation. All access operations have one parameter of type T (implicit in OO notation). An adt with reference semantics has a creation / deletion operation. It may have a type in all access operations (if the parameter is not implicit). This is the type of references. Statements of (12) corresponding to security operations / return parameters hold analogously.

(14) Different interfaces of an inheritance hierarchy: operations' names of different public interfaces (and components' names of private interfaces) are usually different. Operations which are used for dynamic binding have the same name and parameter profile (otherwise case-statements for distinguishing different types are again necessary).

(15) A subsystem groups interfaces which are logically related to each other. So, between the interfaces of one subsystem there is a syntactical "similarity".

(16) The generic parameters of a generic component usually appear in the interface of the component (e.g. a generic type as type in interface operations).

Export – Import Consistencies (MIL Level)

(17) Imported resources of a component B within a component A have to be exported by B. So, import cannot be more comprehensive than export. It is allowed, however, to import only a nonempty part of B's interface.

(18) Any exported resource should be imported by another component. An importing module A imports only necessary resources of B.

(19) Importing a f module, means import of at least one function / procedure. Importing an ado means at least one query operation, or a write operation together with security query and exception / or write operation has return parameter. Importing an adt with variable semantics means import of the opaque type together with at least one query operation or, in the case of an adt with reference semantics, import of reference type with query. For creation / deletion import of the opaque type (variable semantics) or reference type with creation / deletion operations (reference semantics) are necessary. In case of change, import of write operations (with return parameter or security queries with exceptions) are necessary.

(20) The parameter types of imported resources have to be visible for an importing component as well (e.g. by an import).

(21) If different export interfaces of a subsystem are related to each other, they have to be imported correspondingly (e.g. entry together with collection).

(22) For variant programming within an inheritance hierarchy the corresponding imports of superclasses have to be available.

(23) For generic instantiation import of the generic component as well as availability of actual generic parameters (evtl. by import) is necessary. This is true, irrespective whether genericity is a compile time or runtime mechanism.

Interface - body Consistencies (MIL – PL Level)

(24) Export interface - body: What a component exports has to be realized within that component. For a f module there is a procedure on function in the body for every proc or func spec in the interface.The same is true for query, change, and security query operations in the interface of an ado. For any exception the corresponding raise situations have to be programmed in the bodies of the operations within the body of the ado. For an adt, in addition, we find a type definition for the opaque data type or the reference type (for technical reason in the private part of the interface). Trivially, creation and deletion operations have to be realized as well. For any component interface in the interface of a subsystem there has to be a component body in the body of the subsystem. Generic parameters are used in the export interface as well as in the body of a generic component.

(25) Export / import interface - body: Within the body there is no use of external resources which are not imported. The use is consistent with the export corresponding to types and numbers of parameters.

(26) Import - body: Every imported resource is used.

(27) Consistent use within body: If in the body of a component B a component A is used, then this use is consistent with A's export interface. If, for example, A is an ado with change operations to be used with security queries and exceptions, then the change operation is within an if-statement with the security query as condition. Furthermore, the body of B has an exception handler for all exceptions which could be raised.

(28) Export / import interfaces and bodies of an inheritance hierarchy: Every class of the hierarchy exports the corresponding extension and realizes it (programming by extension). It uses resources of imported superclasses (variant programming), mainly for creation operations and operations with the same name in the hierarchy (for dynamic binding). Those operations have to be realized for every class of the hierarchy.

Integrated Architecture Language Approach

- Type of components (not module is module):
  f, ado, adt (later ft)
- Subsystem as big components
- Different usability relations, bound to different structures
  locality: contains and local usability
  general usability for layering
  object-orientation: is – a and inheritance usability
- Genericity as additional concept (compile vs. run time)
- Consistency constraints (context-sensitive syntax)
  for architecture diagrams
  detailed spec language (MIL)
  even between architecture and realization (MIL – Programming Language)

All Concepts Come from Programming Languages

- f from procedural languages (Algol 60)

  ado, adt from modular languages of the 80's (Modula, CLU etc.)
- Subsystems (Habermann, Ada 83)
- Locality : Algol 60

  General usability: separate compilation languages

  OO: Simula 67, Smalltalk
- Genericity as compile time concept (Ada 83), poor man's version (C)

We Use them PL Independent

- Modules in FORTRAN 66, C
- Subsystems in FORTRAN, C
- Locality in FORTRAN, C

} simulation see chapter "Translations …"

- Genericity by macro expansion
- Consistency constraints

} by discipline

Further Kinds of Modules ?

- Collections of open types: Type Collection Module (tcm)
  - for different application domains e.g. physics
  - for connection to a programming language
  - also operations for type conversions
  - also constants
- Do not regard modules of that kind. Are necessary but
  - corresponding modules are simple
  - do not play a big role in architecture modelling
  - make an architecture unnecessarily complicated
- Due to methodology and as we do not regard tc modules
  - ⇨ There are no modules with "open objects" in the interface

Architectural Language(s) for Practical Problems
Not a "Brave New World"

- Reengineering (take classical part)
- Embedded Systems (concurrency extensions)
- Distributed Systems (signals, remote calls)
- Development means different stages:
  logical architecture, concrete architecture
  usual functionality, control functionality

  ⇨ see later chapters