5. Subarchitectures: Methods & Patterns

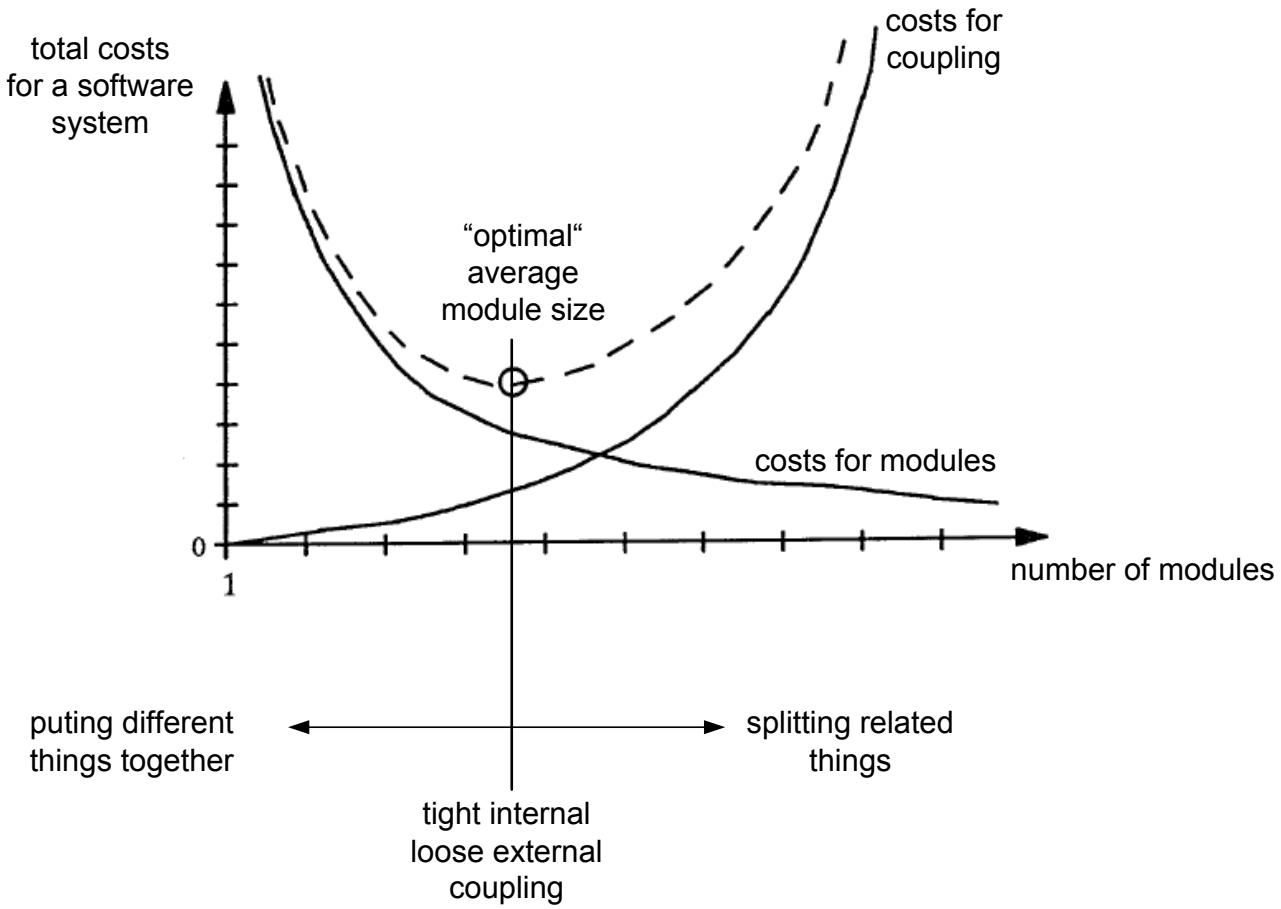
Aims

- Discussion on level of subarchitectures:
  coarser than last chapter, below of complete architectures
- Application chapter for preceding chapter
- Methodological questions:
  Functional abstraction vs. data abstraction
  How to deal with OO
  etc.
- Discuss patterns to be applied in many architectures:
  How to build standard situations within architectures
  How do modules of different types interact
- Transformation patterns:
  How to transform, e.g. “object situation“ into a “type situation“, so architectural situations
  A “wrong“ to a “right“ solution

Contents of Chapter 5

⇨ Application of chapter 4 (where which modules, relations)

Component Size
Size of Modules (Analogous for subsystems)



no precise definiton of "ideal size":

1 page sc

too small

(useful, if situation might get more complicated, e.g. opening dialog)

20 page sc

too big

⇨ from "optimal" size we get "optimal" no of components

Small Number of Connections

- “right“ size of components
- apply data abstractions: details within components
- every edge needs cooperation, coordination, and communication:

  different developers: design – implementation

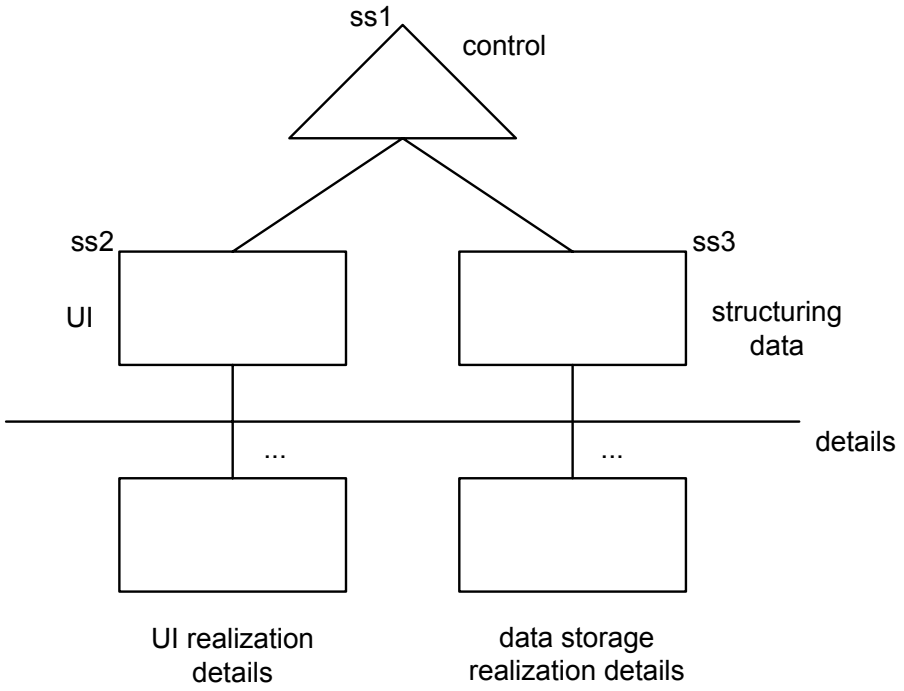
  implementation – integration

  implementation – quality assurance

  design, implementation - documentation

Independent Subarchitectures and, Consequently, Subsystems

- need not understand internals of a ss to understand the whole picture
- allows two – level design process
- only some designers look into a ss
- ss are big reusable chunks
- can look on a system regarding levels 1 – i



⇨ see motivation for subsystems

Summary: Component and Relation Numbers

- optimal size / no of components
- minimum no of edges
- independent subarchitectures / subsystems

} **vague** rules !

Is there a metric for architectures, which is practically useful ?

Evident Architecture Language Application / Method Rules

(l1) Locality structures should not be too deep. Are there parts of general nature? Can you find parts which are independent, i.e. subarchitectures / subsystems.

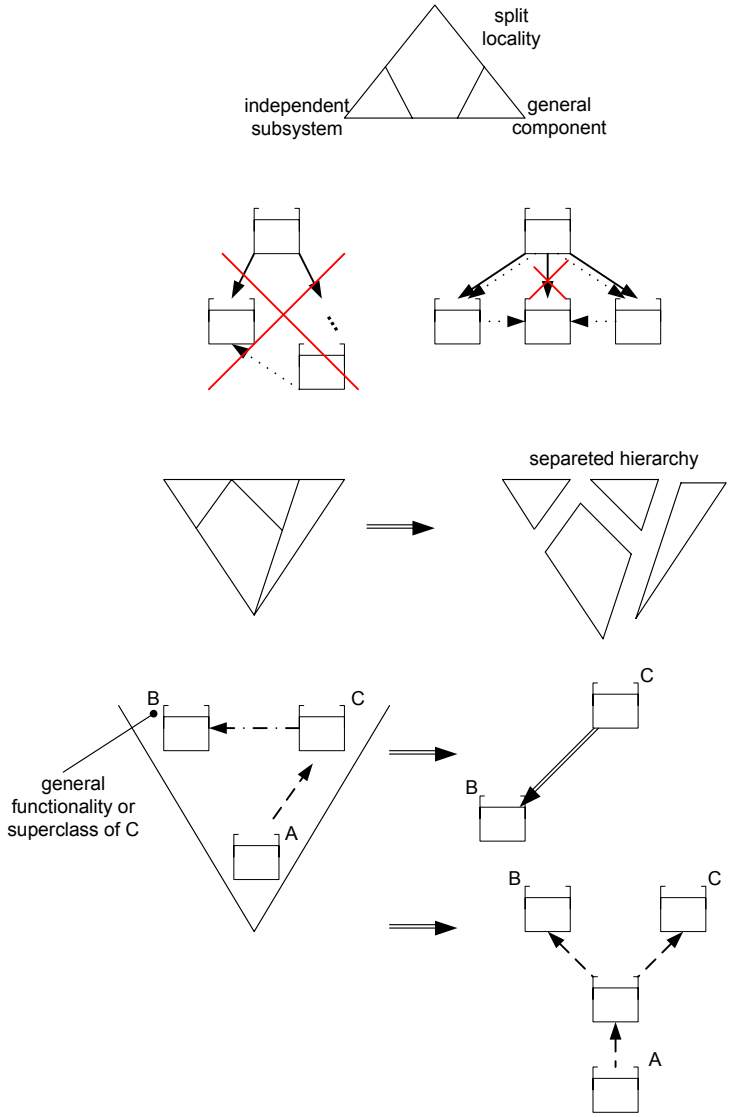
(l2) A contains relation has a parallel local usability relation. If not the case, you have modelled a general component by a local situation. A local usability relation need not have a parallel contains relation (see recursive situation).

(i1) Inheritance structures should not unite different things. As with locality, think about different parts to be separated.

(i2) If there are close relations between different parts of an inheritance structure there might be two design errors:

(a) There is general functionality which should be a component outside the inheritance structure.

(b) B also represents commonalities which should be put in a superclass of B and C.

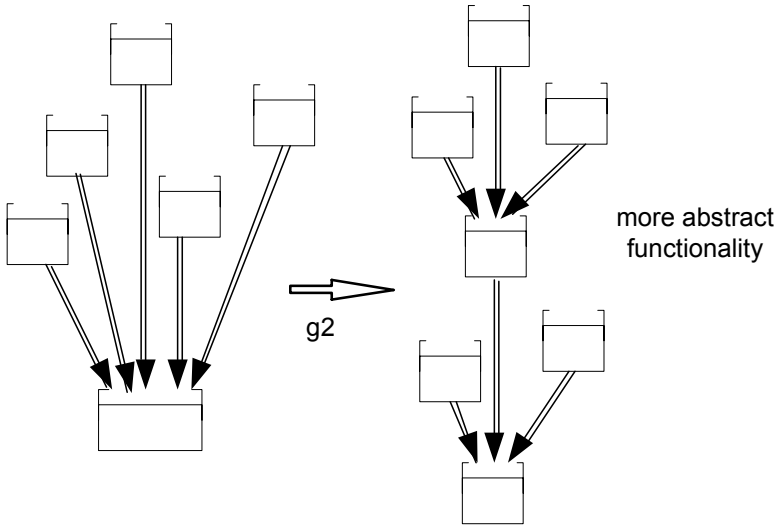


Rules Condt.

(g1) If a general component H (module or subsystem) has only one targeting general usability, please check whether it is really a general component.

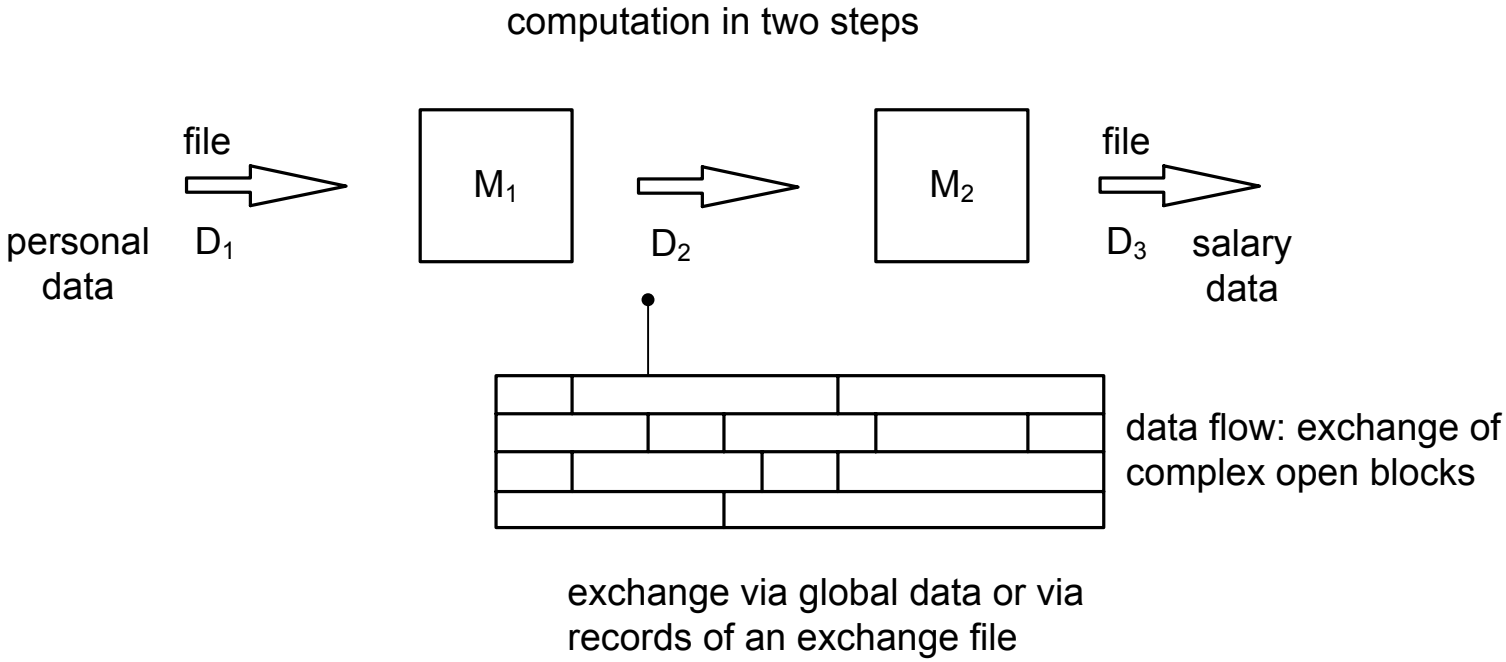
(g2) If a component has many targeting general usability edges, especially from differnt layers, it was forgotten to define more abstract general functionality.

(int 1) If the interface of a module gets two broad, check whether it belongs to one module or different modules, or to different interfaces belonging to one subsystem. Every component should have one design decision.

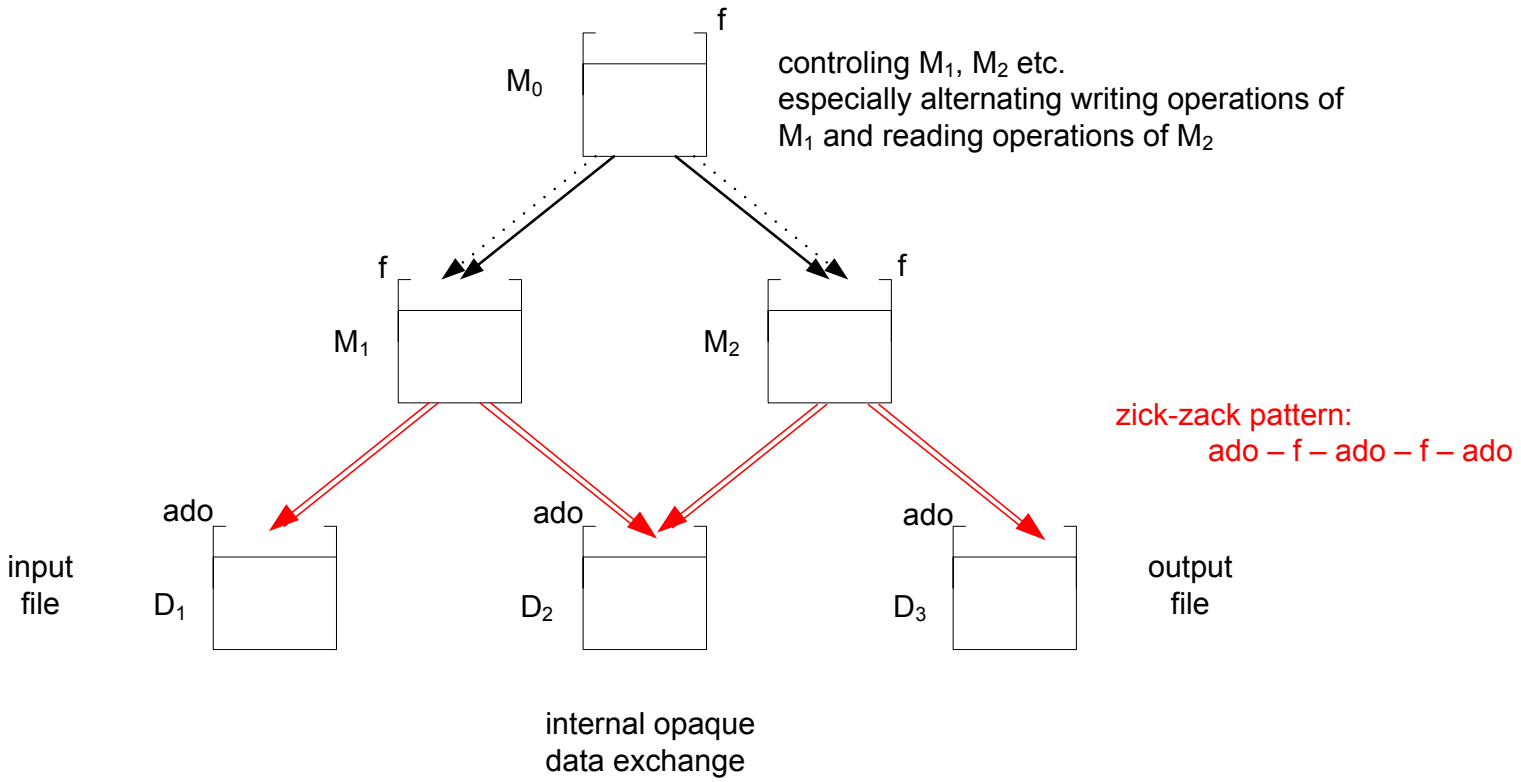


Exchange of Open Data (No Data Abstraction)

traditional software development of programs and data,
data with all their representation details



Exchange of Functional Modules via Ados



the same situation for a compiler: M_1, M_2 phases (lex scan, syntax analysis)

D_1 text file, D_3 syntax tree, D_2 token list now a file (sequence of tokens),

M_1 and M_2 decoupled

Data Exchange of Functional Modules via Adts

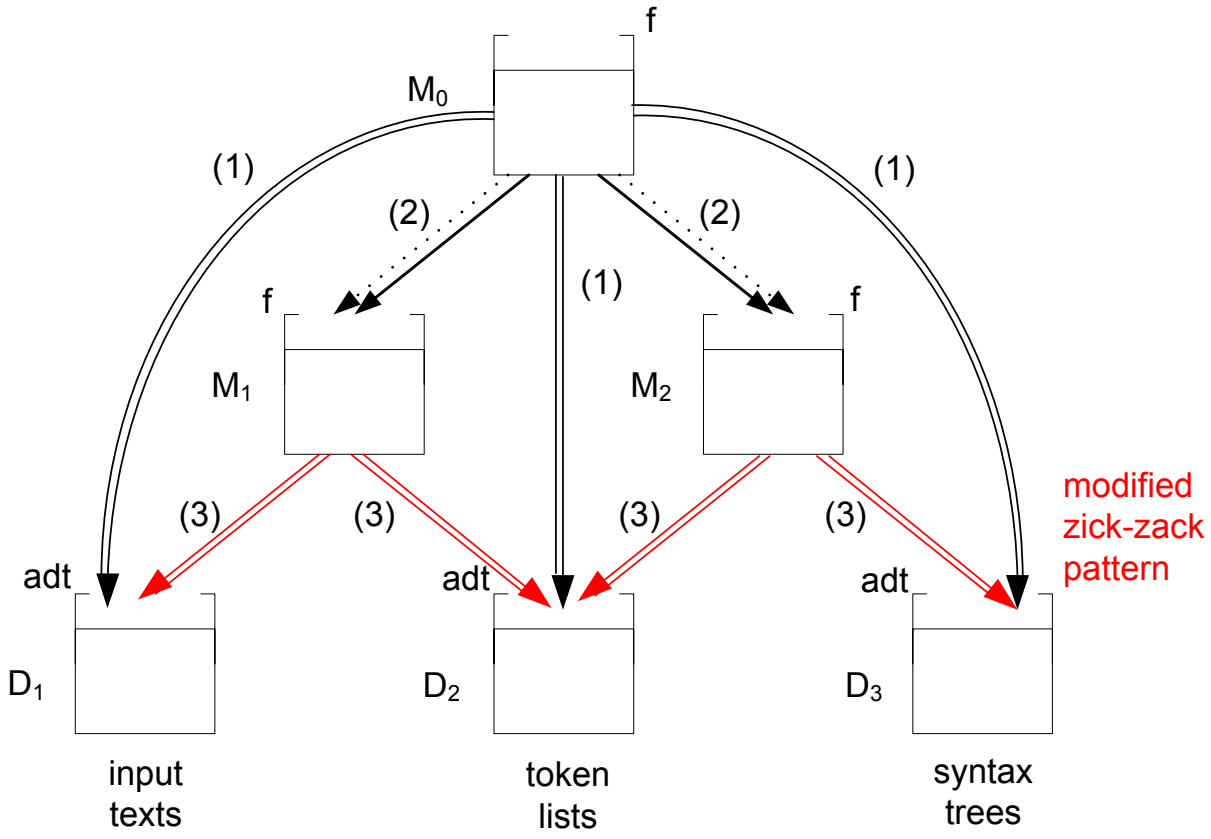
again compiler example, now language with seperate compilation:
translation of more than one program text

(1) ados created in the body of M_0

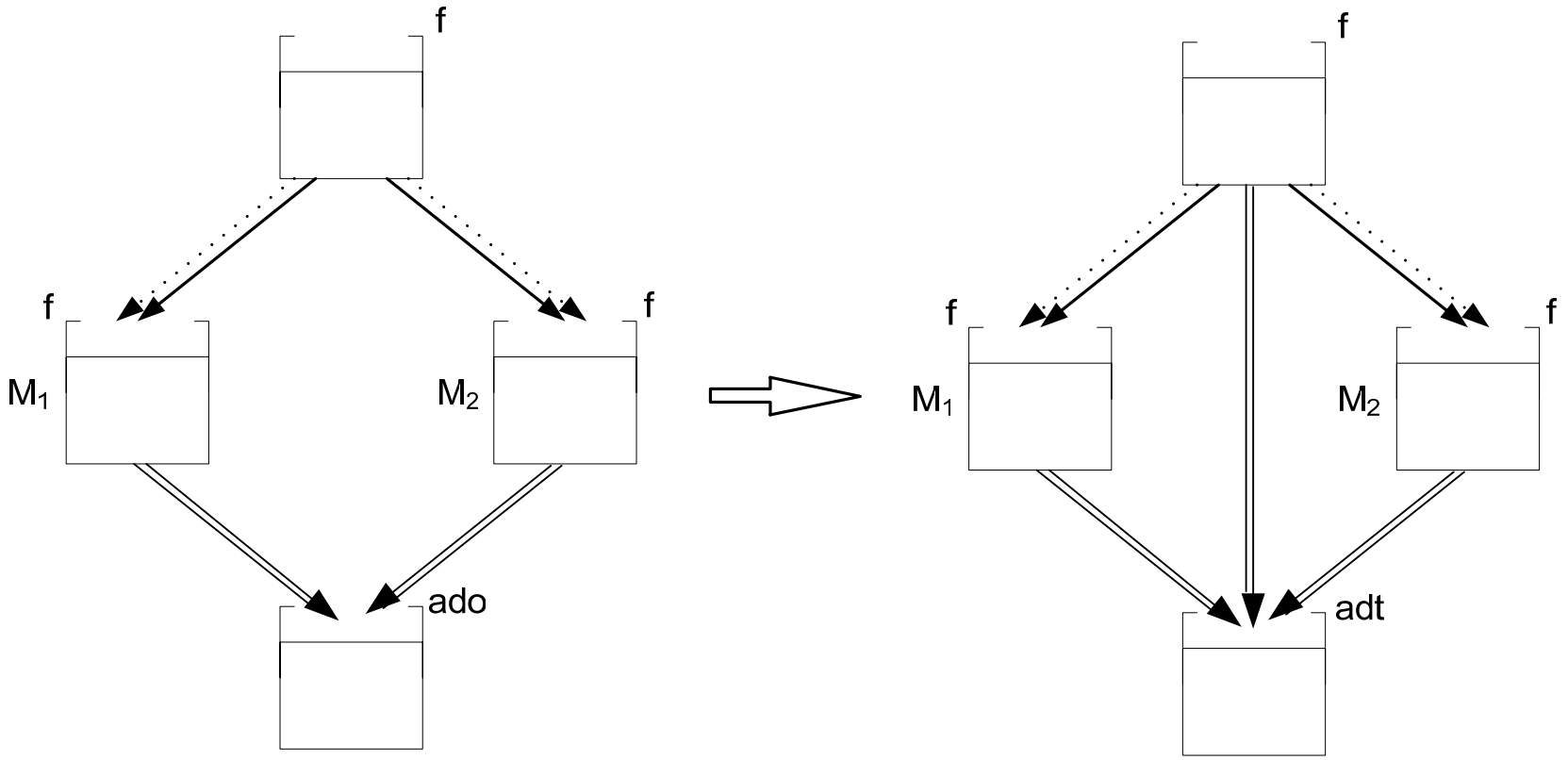
(2) passed as parameters to M_1, M_2

(3) M_i need access operations of adt

adts on architecture are templeates

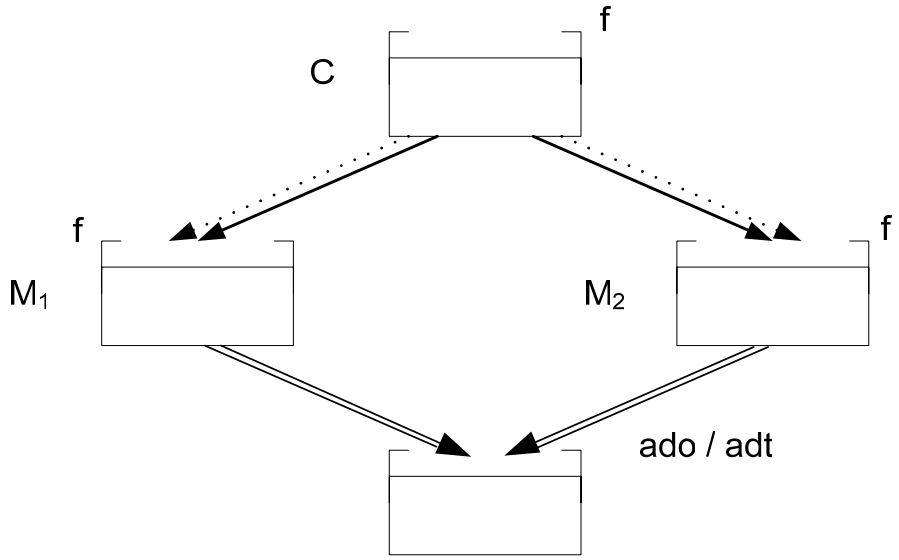


Transformation Pattern



substituting ados by adts

Different Forms of Data Exchange



one write one read on the ado : entry

so M_1, M_2 strictly one after the other

write and read, on a collection, e.g. : collections

queue, or in same order

or arbitrary, if M_1 and M_2 are processes

→ later

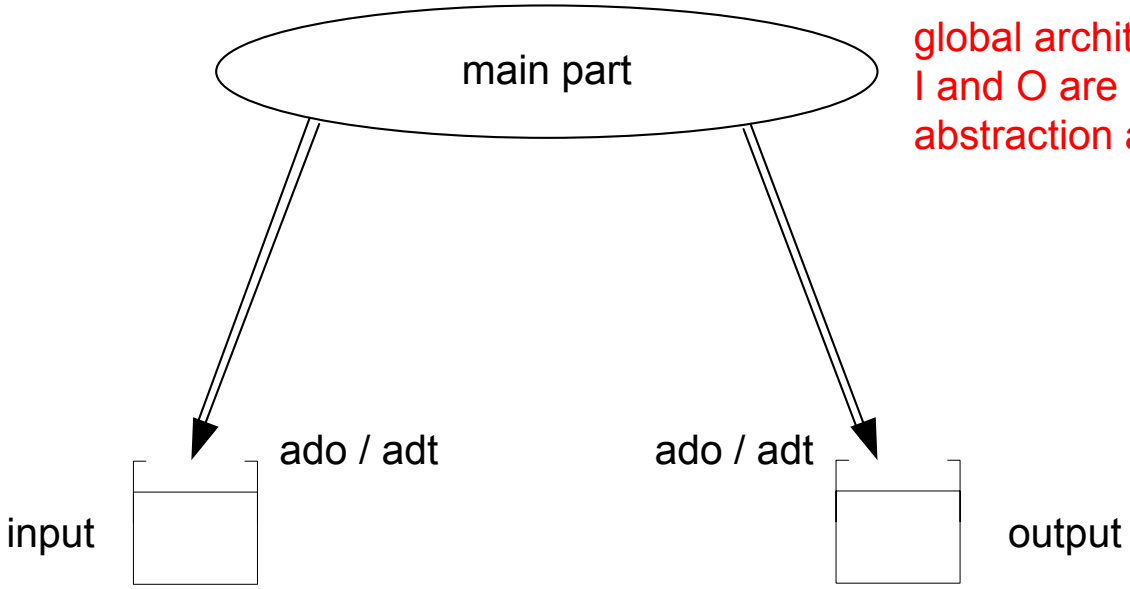
all writes, then all reads, loose coupling : file

of M_1 and M_2, M_1 and M_2 possibly different systems

controlled by C

4 exchange patterns

I and O Are DA Applications



global architecture pattern:
I and O are (multiple) data abstraction applications

hide details:

- virtual files
- virtual devices

in interactive systems also:

- layout
- UI style
- window system

disscussion:

I/O what do we regard

device, data structure, i.e. containers from / to which we get / put data

or activity to organize / carry out I or O “process“

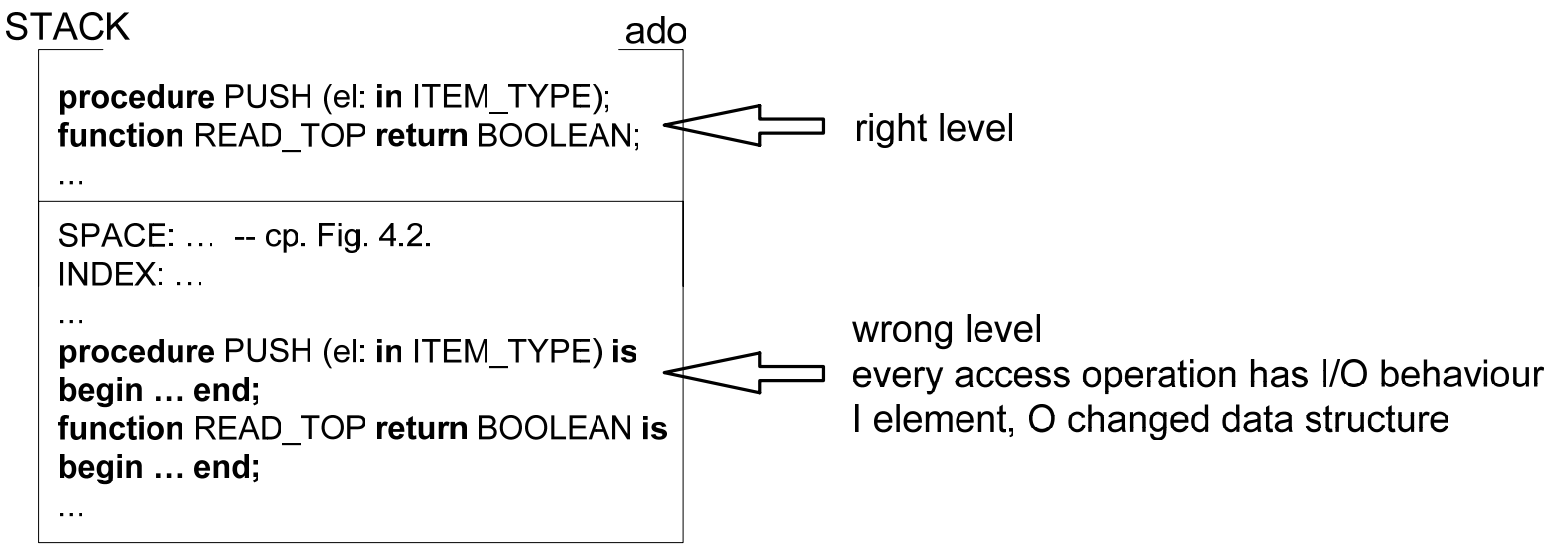
Functional vs. Data Abstraction: Remarks

- of which type is a module:
  - » state / memory → ado
  - » template for states → adt
  - » transformation, computation, I/O behaviour → f
- not “a“ module and determination afterwards yields “fruit basket“ modules
- essential question:
  independent f modules or access operations of a ado / adt module
- f modules action-oriented:
  does not mean that they get active from themselves, are invoked from above

  ado / adt passive:
  may invoke other modules below, even functional ones
- iterators in the interface of a da module: no clear distinction between passive (da module) and active (iterator)

Functional vs. Data Abstraction: Rules

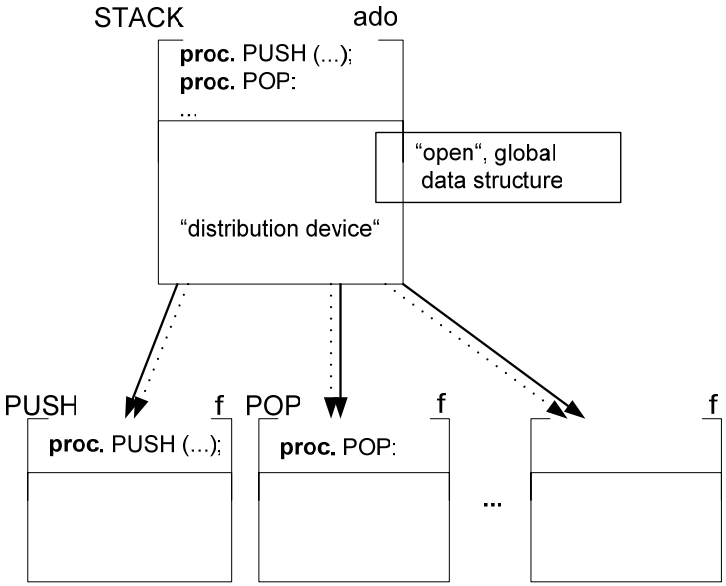
- rule 1:
  look at interface and not on body
- rule 2:
  regard module and not system where module appears
  example: control of a dialog within interactive system, see below
- rule 3:
  whenever f module A_1 acts on a data structure, check whether A_1 together with A_i belong together as operations of a da module



Functional vs. DA: Wrong Situations

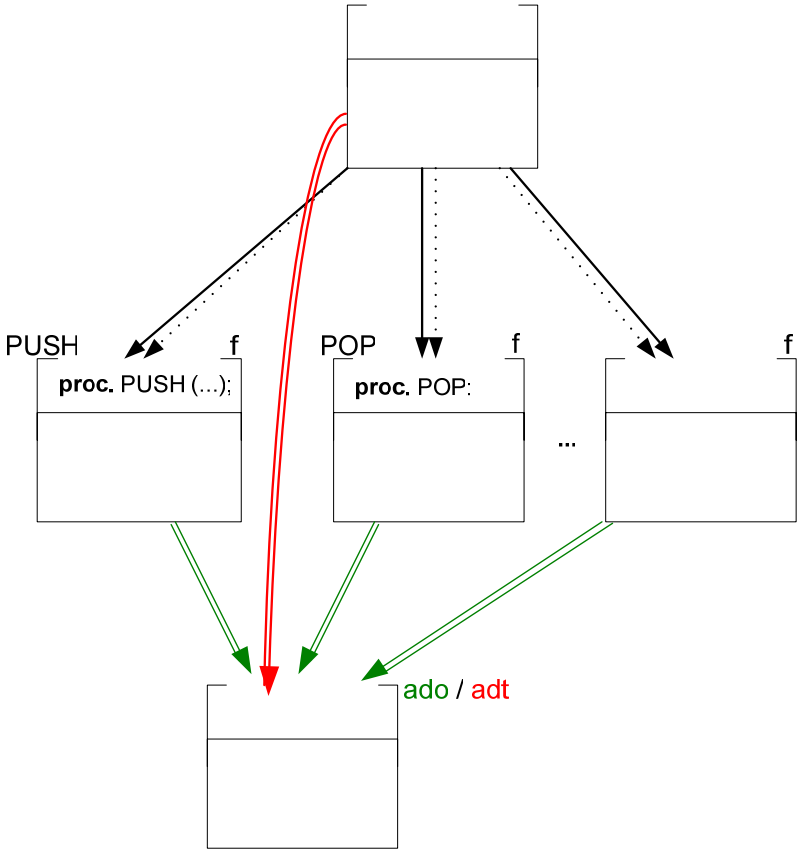
situation a) is impossible

a)

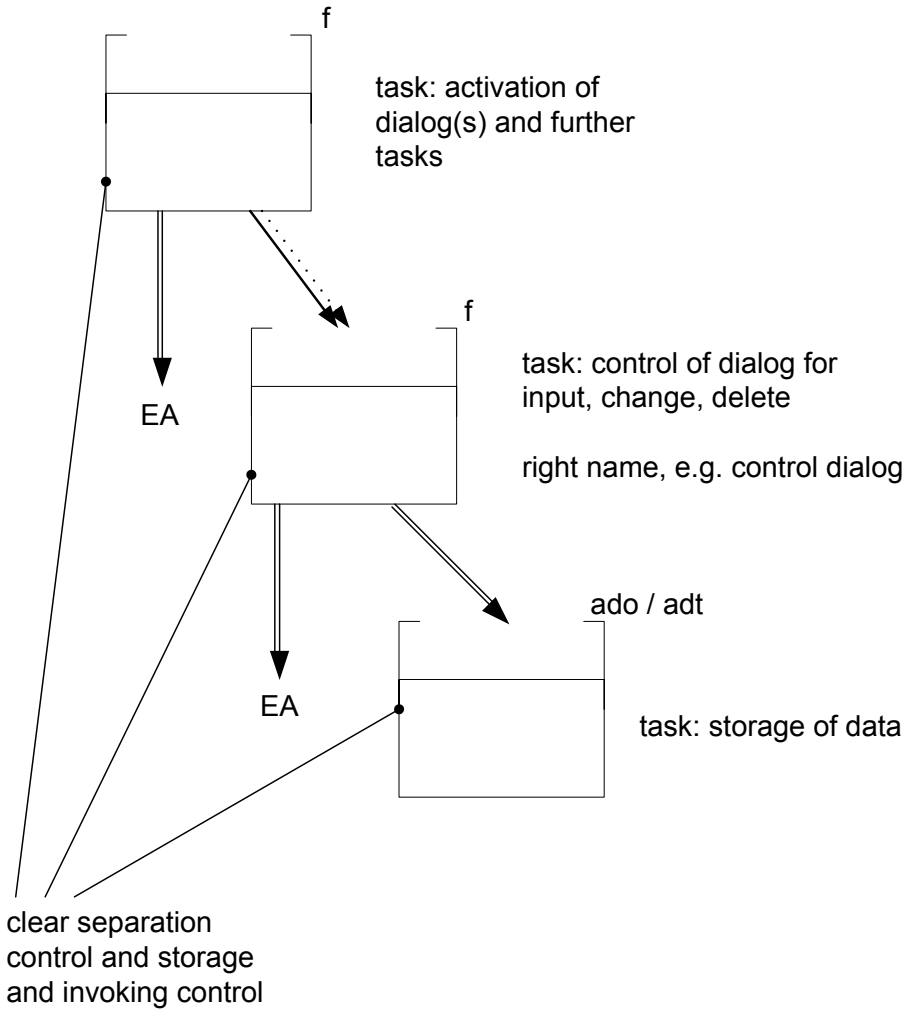


situation b) is possible

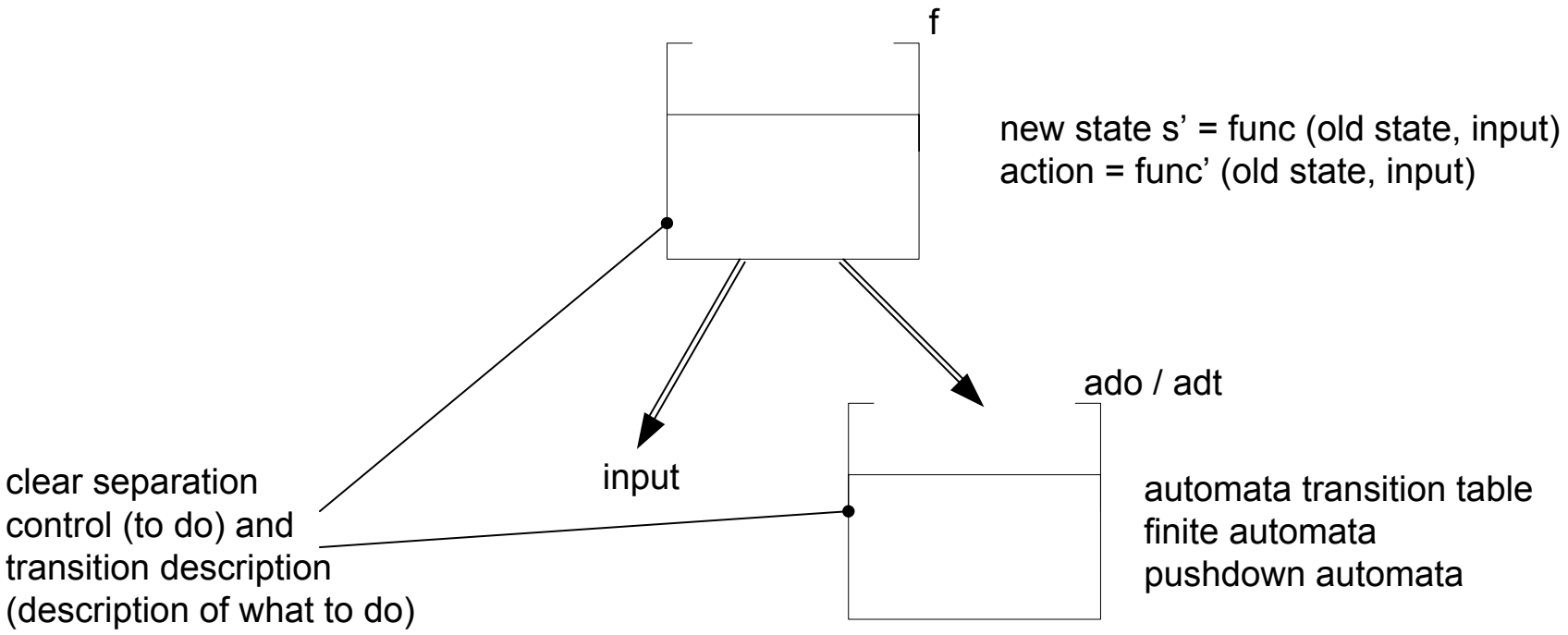
b)



Functional vs. DA: Example Dialog Control

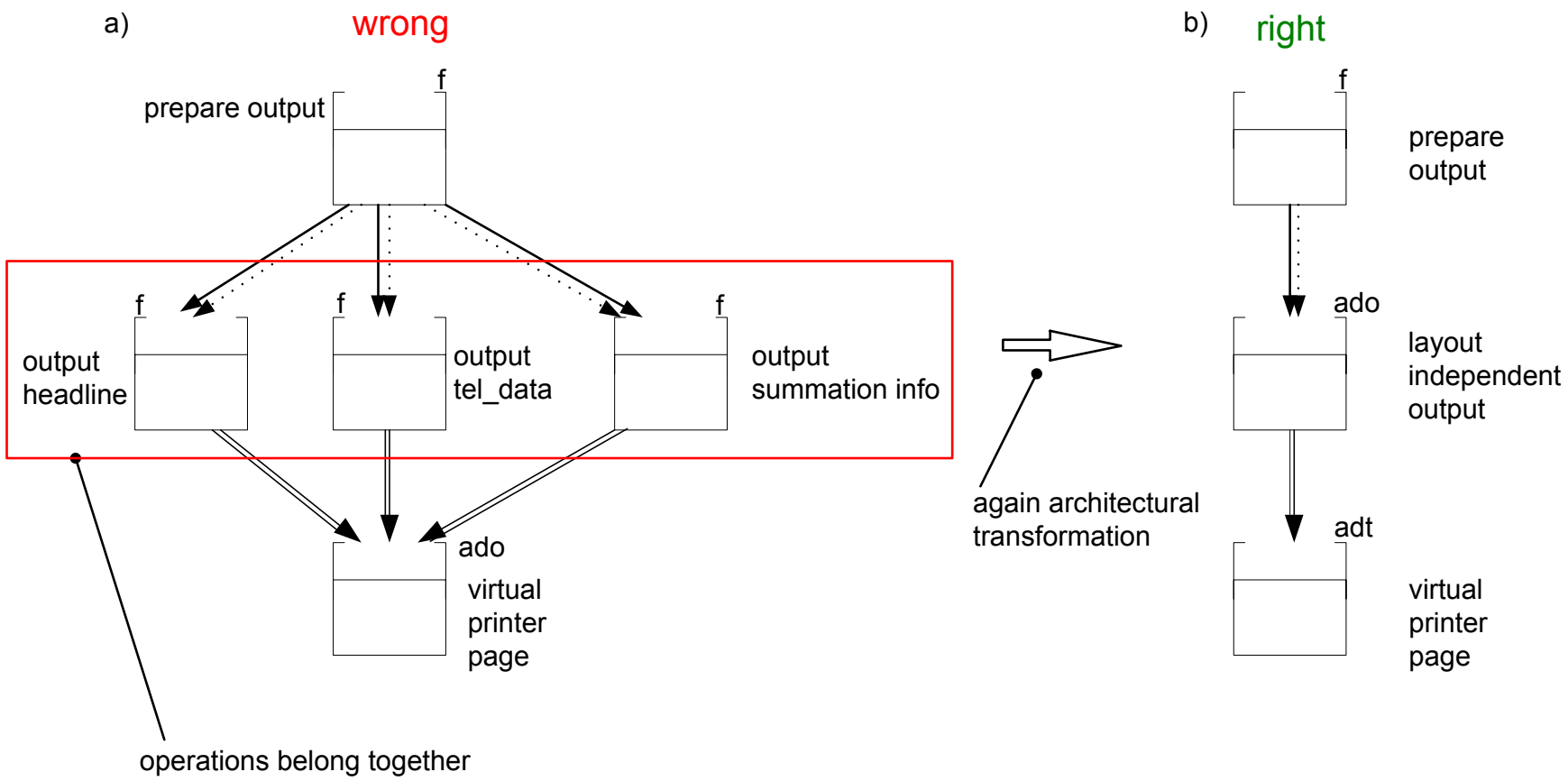


Functional vs. DA: Example Machine / Automata Control

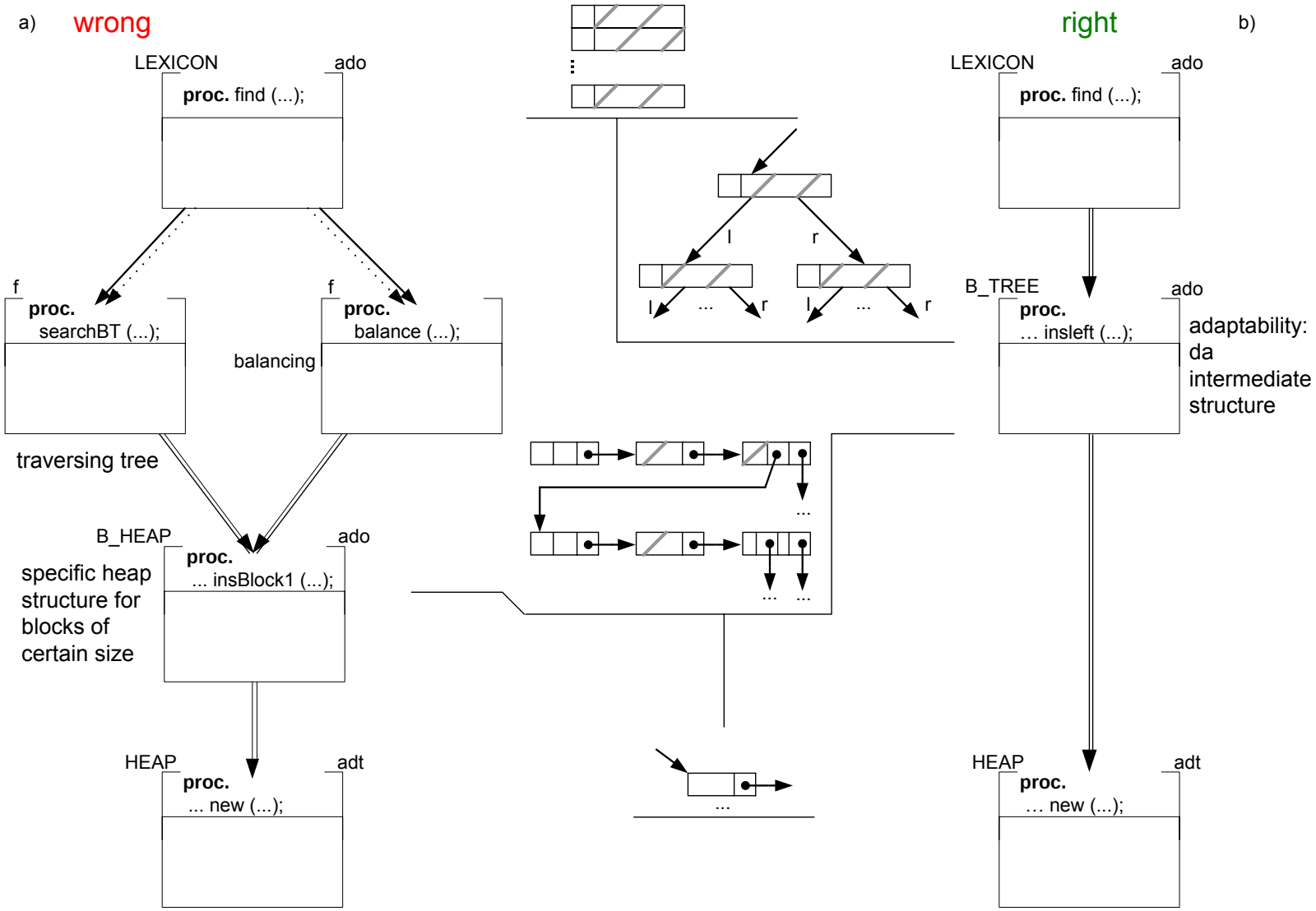


- discussion:
  - » functional module has no state
  - » exchange of table and behaviour

Example: Handling Output

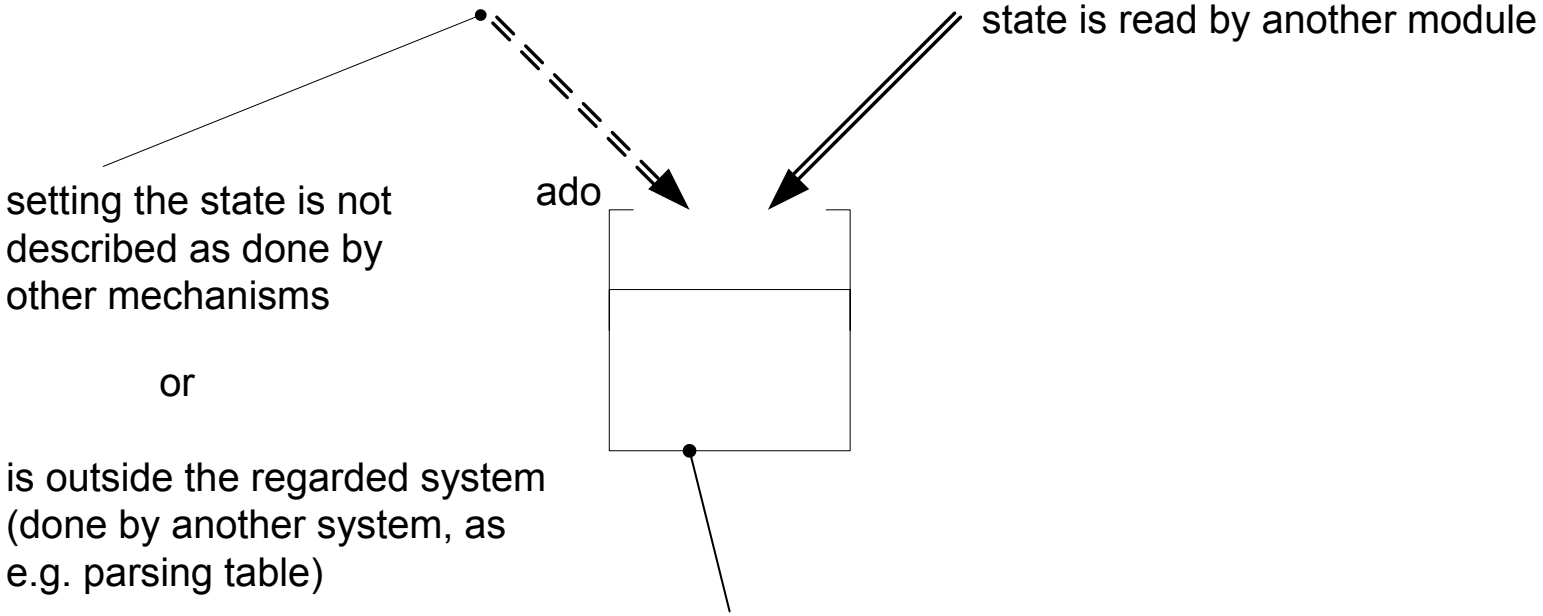


Example: Design Below a DA Module

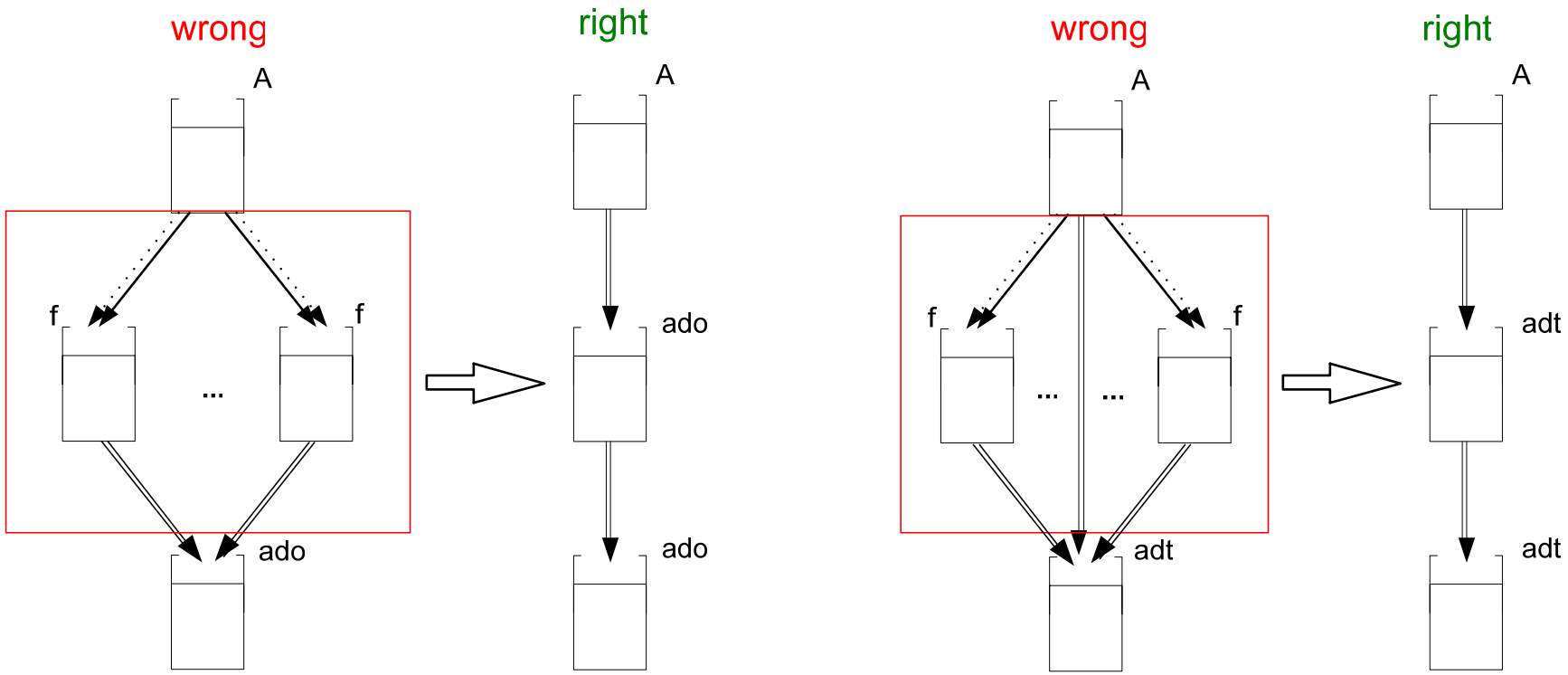


Intermediate Forms of Modules ?

- What is a
  - » Random number generator
  - » Keyboard
  - » Mouse ?



Functional vs. DA: Summary



A may be a da module under which there was a wrong design (stack example)

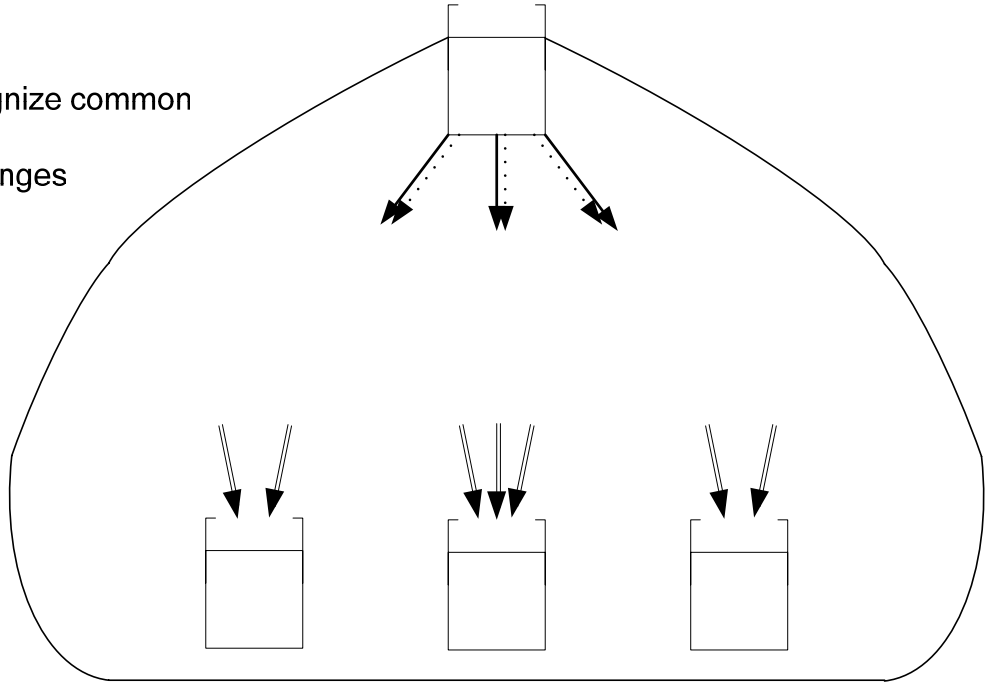
or a f module under which da was overseen

⇨ cut horizontally, not vertically

Where Does Functional and Data Abstraction Occur ?

Top down

danger:
not to recognize common modules
severel changes afterwards



functional modules and
local usability:
specific to a certain system

da modules and
general usability:
general building blocks

Bottum – up

onion model

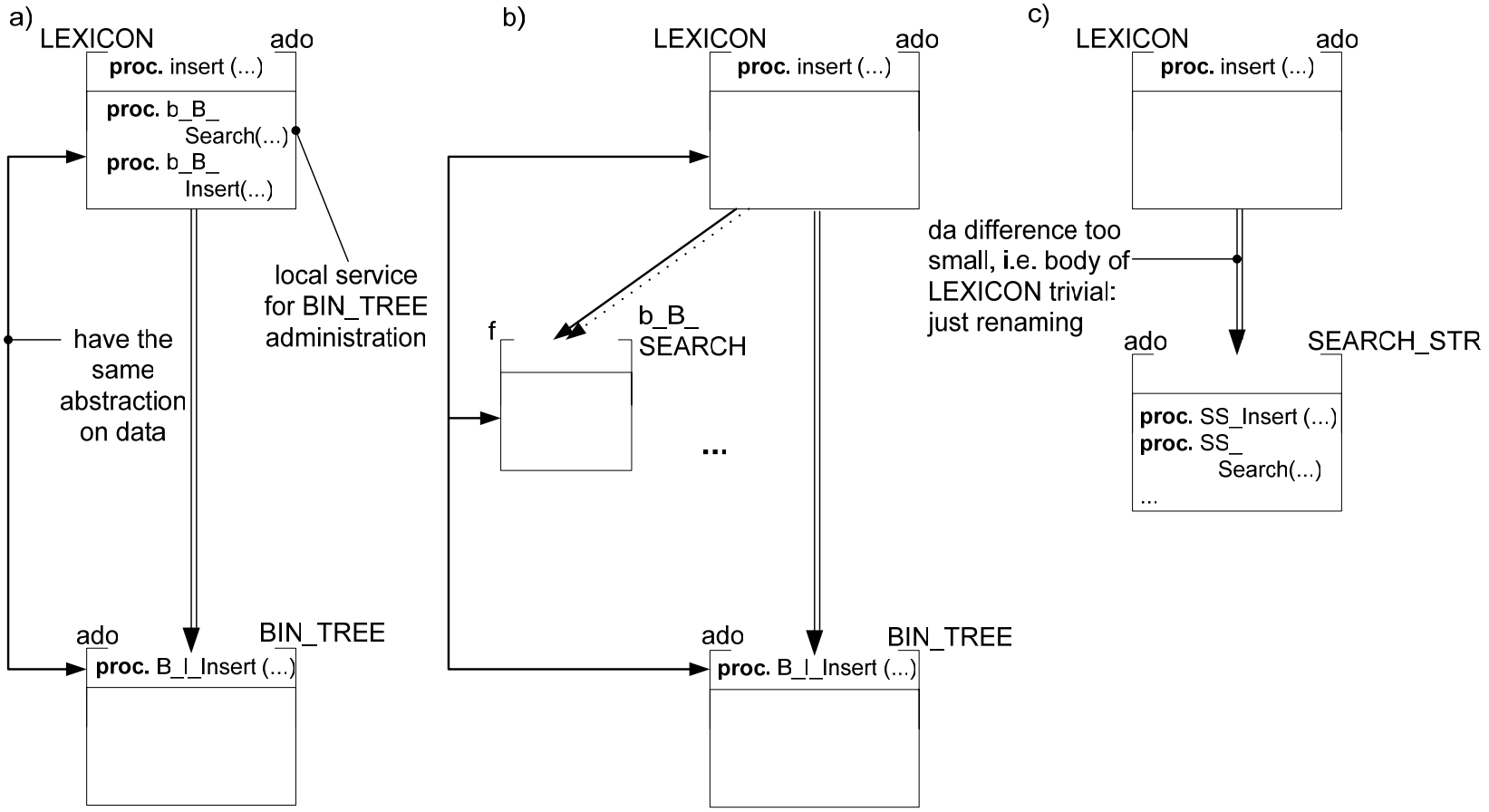
Layering Below a DA Module

How looks subarchitecture below a da module A (see discription above):

- Realization in the body of A alone
- Only one further level below : stack – list
- 3 or more levels: lexicon – binary tree – heap
- Functional layer in between (was it a da layer, we did not recognize ?)

⇨ Layering below da modules: topic of this section

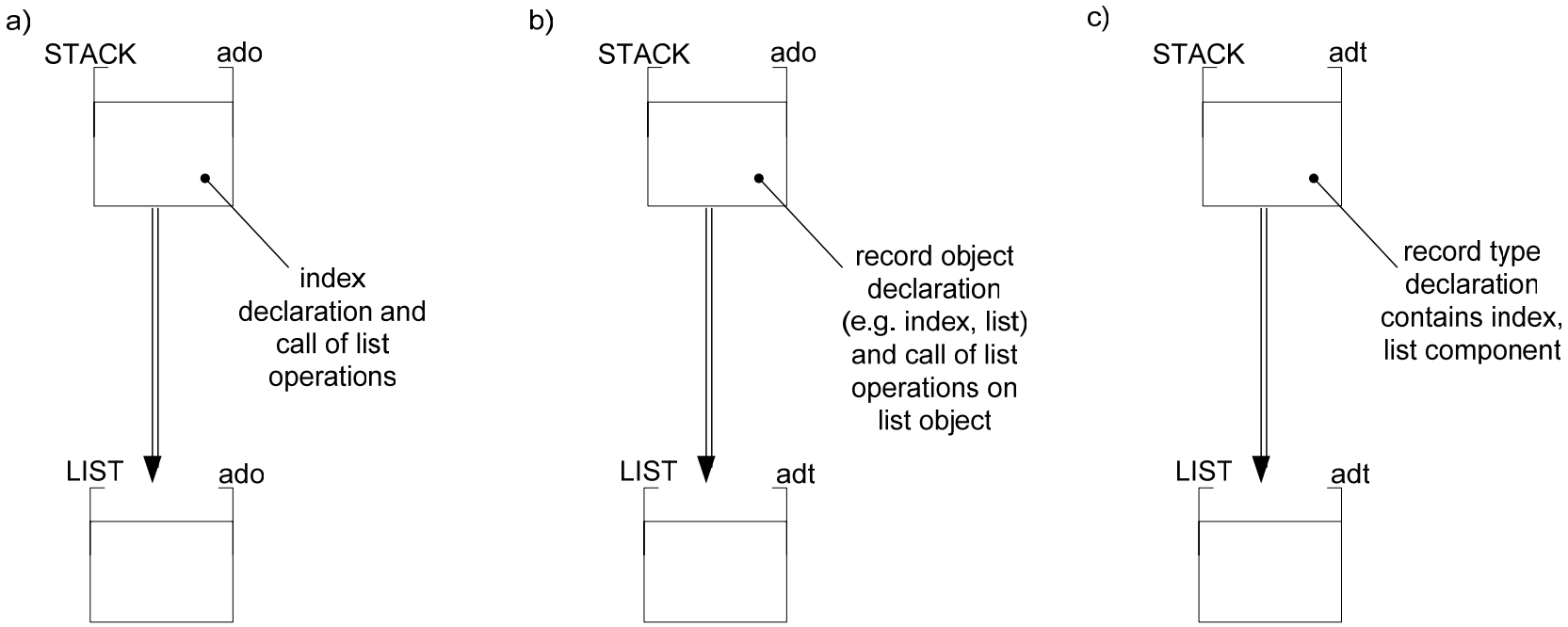
Interaction of Two DA Layers



discussion:

- » where to put common services: in the body, own layer ?
- » abstraction difference: not trivial, not too big

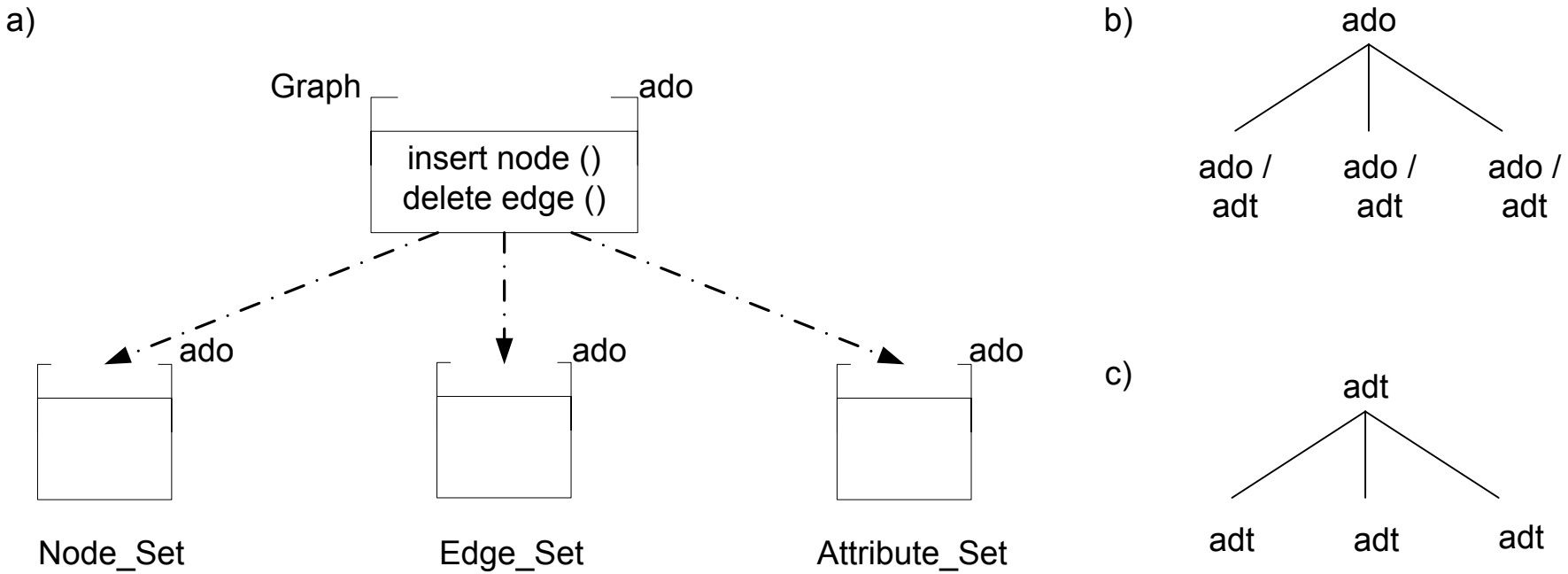
Neighboring DA Layers and Module Types



discussion:

- » also for more than two levels
- » adt – ado:
  - not allowed if both are collections, entries (same DA application)
  - allowed for changing DA applications: entry adt and corresponding heap (collection)

Using More Than One DA Module



Where Is The "Memory" of an Ado Module ?

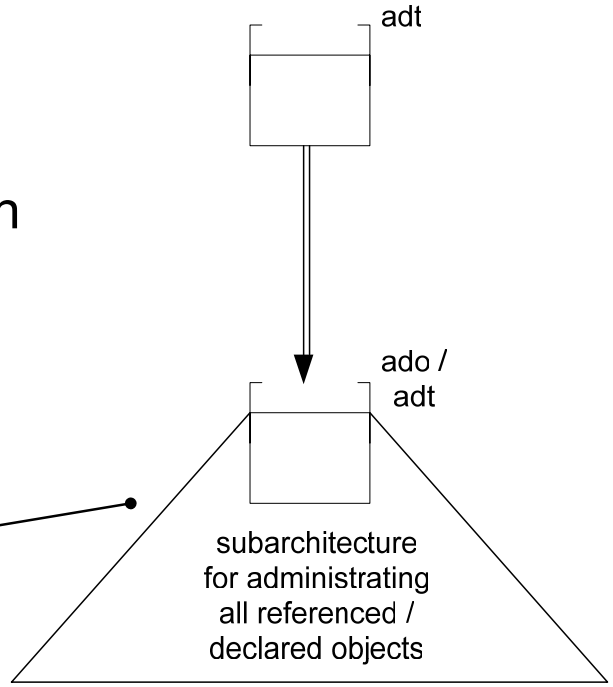
- realization complete in the body of the ado STACK
  memory data container, bookkeeping info, altogether there using programming language constructs (array, index)
- ado STACK is based on an ado LIST:
  bookkeeping in STACK, data container in LIST
- ado STACK is based on an adt LIST:
  bookkeeping in STACK, LIST object in STACK
- adt STACK
  memory (container + bookkeeping info) in the client module which uses STACK (more precisely next slide)

Discussion above for objects (variable semantics).

How are the answers for references (reference semantics) ?

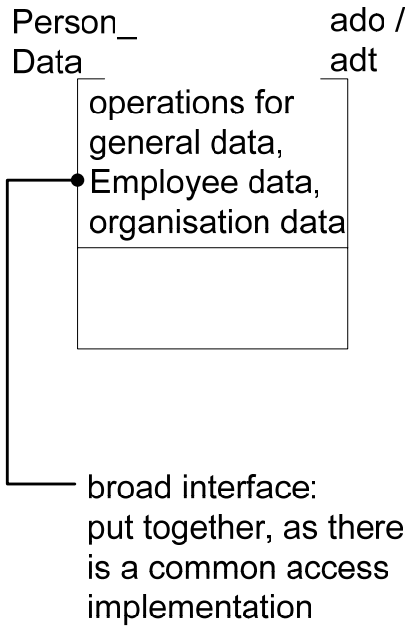
Where Are The Objects of an Adt Module ?

- Variable semantics adt:
  - » objects are created via declaration in the body of client module
  - » runtime stack of the underlying PL implementation administrates objects (creation, deletion)
- Reference semantics adt:
  - » objects are created via creation operation and deleted via deletion operation
  - » below the adt there is a subarchitecture, eventually heap of the underlying programming language
- Summary: below an adt there is a subarchitecture, often given by PL implementation

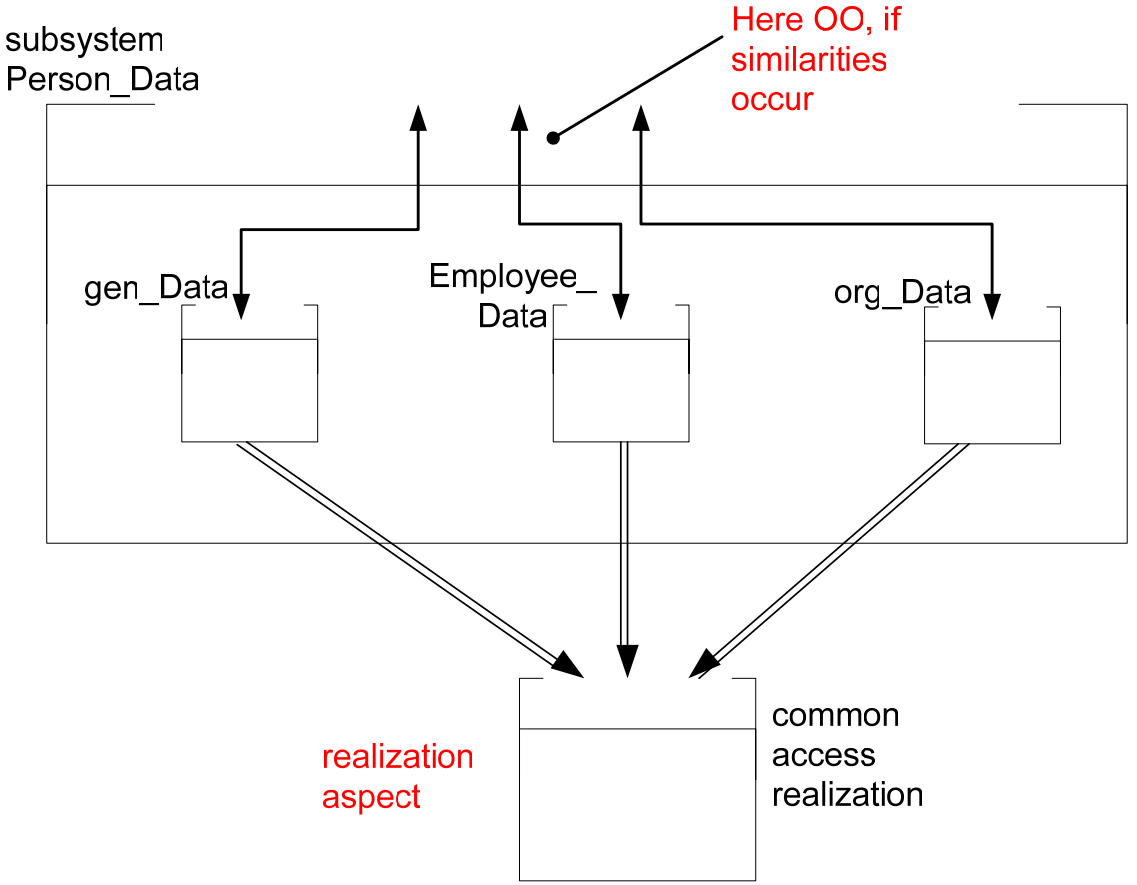


Access Modules & Broad Interfaces

a) access module



b) different interfaces aggregated



Overview

- Entries in a collection
- Two – fold data abstraction: ...
- How do both DA modules interact with each other and a client ?
- Application of subsystem concepts

Simplified Solution: Entries Implicit

```
abstract data object module Card_Boxes is                                    --
    procedure Card_store (...);          -- all components of an entry      --
    procedure Card_delete (...);         -- occur as parameters of the      --
    procedure Card_change (...);         -- collection operations           --
    function does_Card_exist (...) return BOOLEAN;                          --
    function Space_available return BOOLEAN;                                --
    ...                                                                     --
    Card_not_existing, no_Space_available: exception;                       --
    -- the semantics of the operations are the following: ...               --
end Card_Boxes;                                                             --
```

Prerequisites:

- client uses only one entry at the same time
- client uses entry only as part of the collection

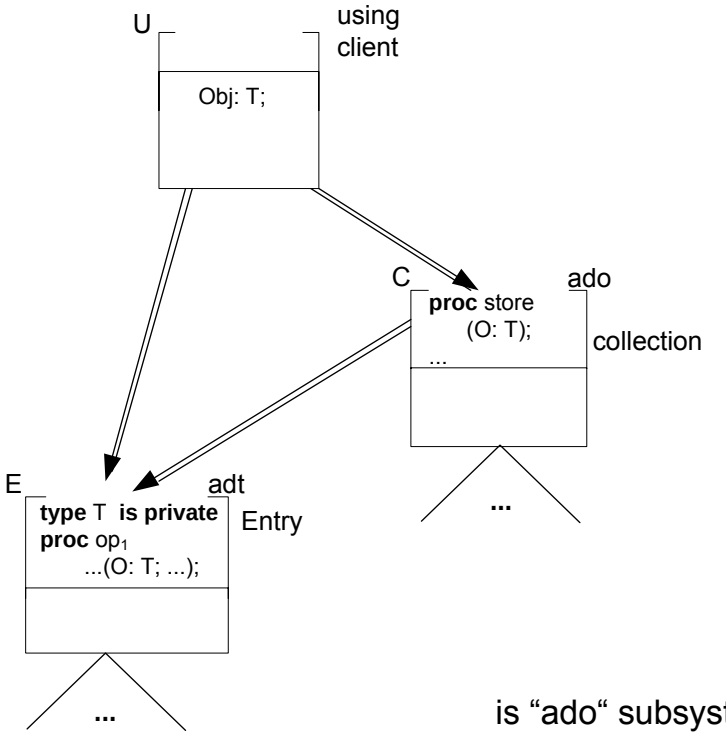
Dirty Solution: Open Entries

```
"undetermined" module Card_Boxes is                                    --
    type Card is                                                       --
        record                          -- all components of the       --
            KEY: KEY_T                  -- record are open: whenever   --
            NAME: NAME_T                -- the entry is changed,       --
            ...                         -- many modificatons           --
        end;                                                           --
    procedure Card_store (K: in Card);                                 --
    procedure Card_delete (K: in Card);                                --
    ...                                                                --
end Card_Boxes;                                                        --
```

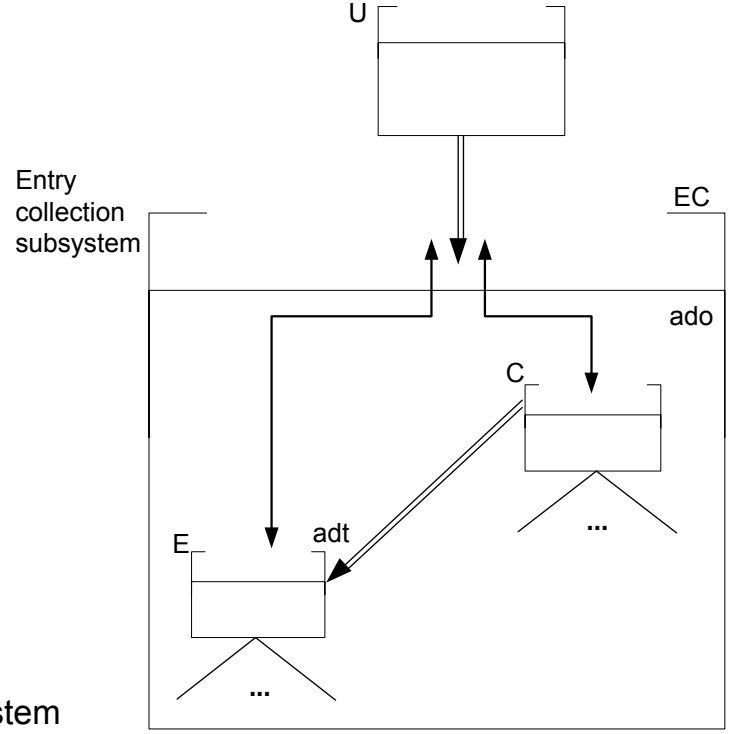
no data abstraction for entries, entry data by client

module of mixed character: adt and ado

Clean Solution as Subarchitecture or Subsystem



is “ado“ subsystem



discussion:

- we can deal with several entries
- entries have same importance
- usability C – E necessary

using E in client

- variable semantics
  - objects declared, manipulated, stored
- reference semantics
  - reference declared, objects created, manipulated, and stored

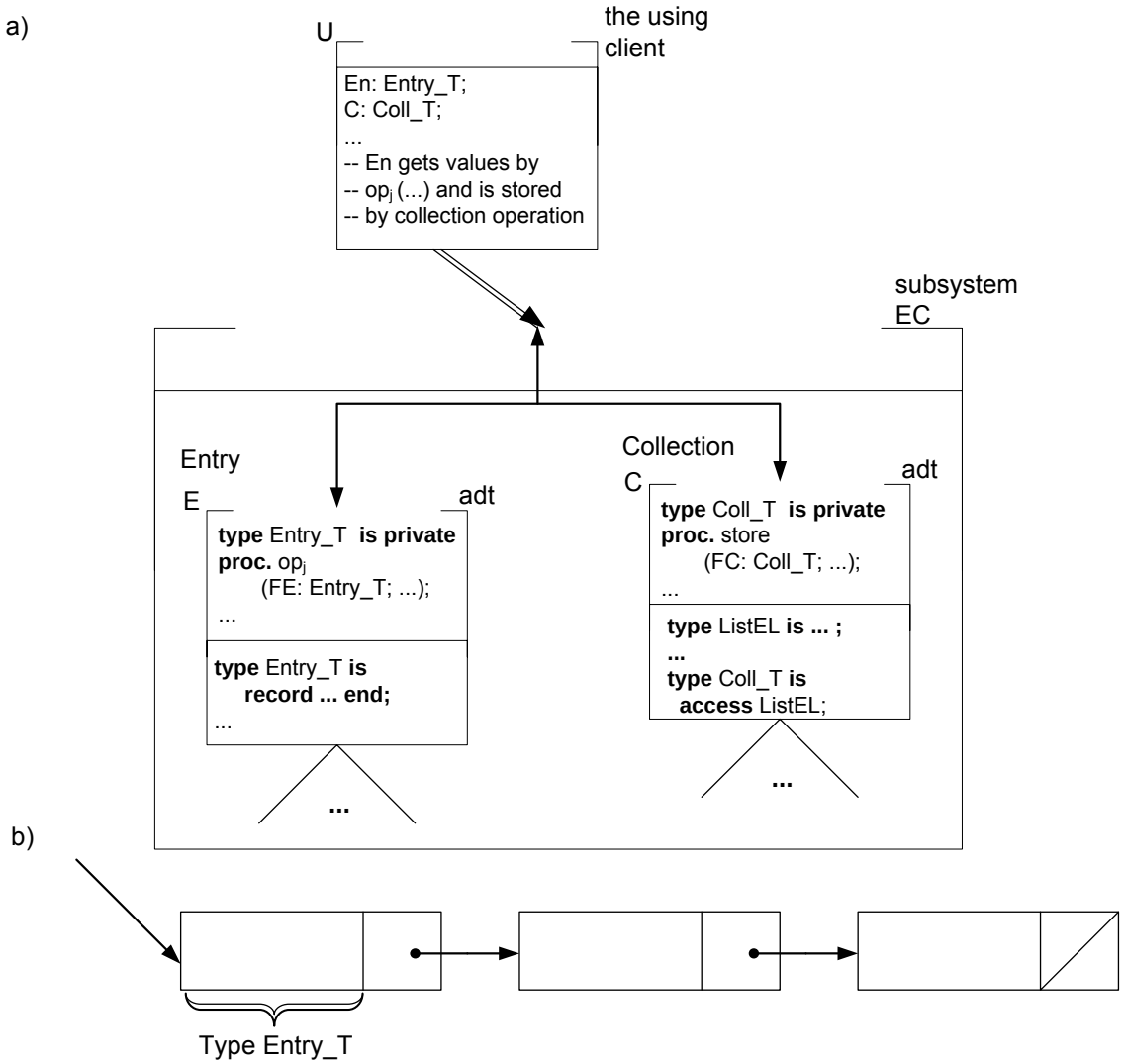
Application to Card_Boxes Problem: Interface of Subsystem

```
-- data type                                                                    --
subsystem Card_Boxes is                                                         --
    general import from type collection module Component_Types using all;       --
-- The subsystem deals with entry collection situations. We chose an adt for the collection to allow for multiple  --
-- collections. The parameter types for entry operations are imported.          --

    export from abstract data type module Card is                               --
        type Card_T is private;                                                 --
        procedure initialize_Card (aCard: out Card_T);                          --
        procedure delete_Card_components (aCard: in out Card_T);                --
        ...                              -- The semantics if entry operations are the following: ...  --
    end Card;                                                                   --

    export from abstract data type module Boxes is                              --
        type Box_T is private:                                                  --
        procedure store_Card (Card: in Card_T; Box: in out Boxes_T);            --
        procedure delete_Card (Card: in Card_T; Box: in out Boxes_T);           --
        ...                              -- The semantics of collection operations is the following: ...  --
    end Boxes:                                                                  --

end Card_Boxes;                                                                 --
```

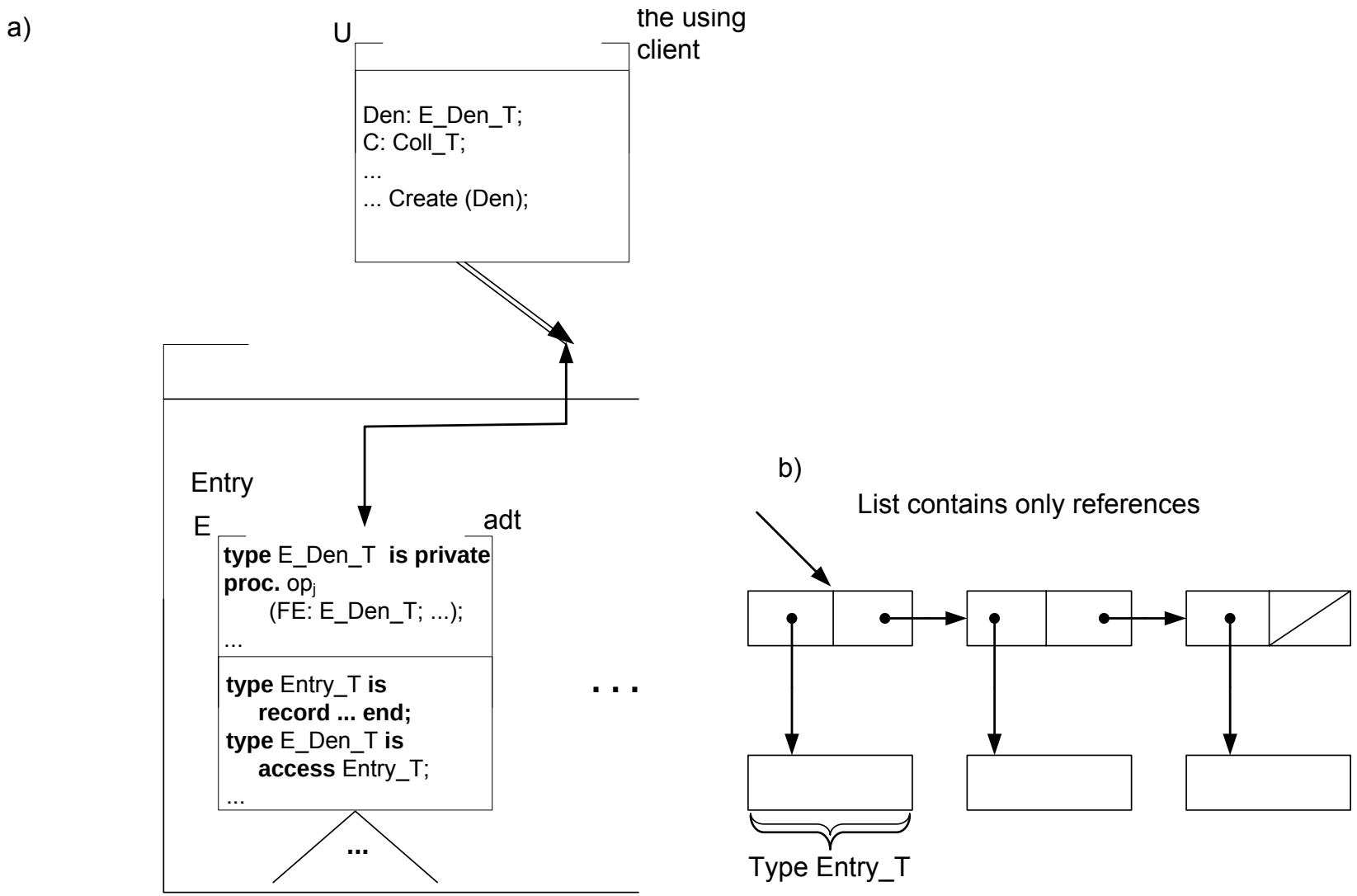
Entry with Variable or Reference Semantics

- discuss interaction of E, C, U
- entry may have variable or reference semantics
- is the difference important on architecture level ? Yes
- (also important for realization, e.g. body of U)
- study situation
  - » E has variable or reference semantics
  - » collection C is adt, realized as linked chain

Variable Semantics for Entries



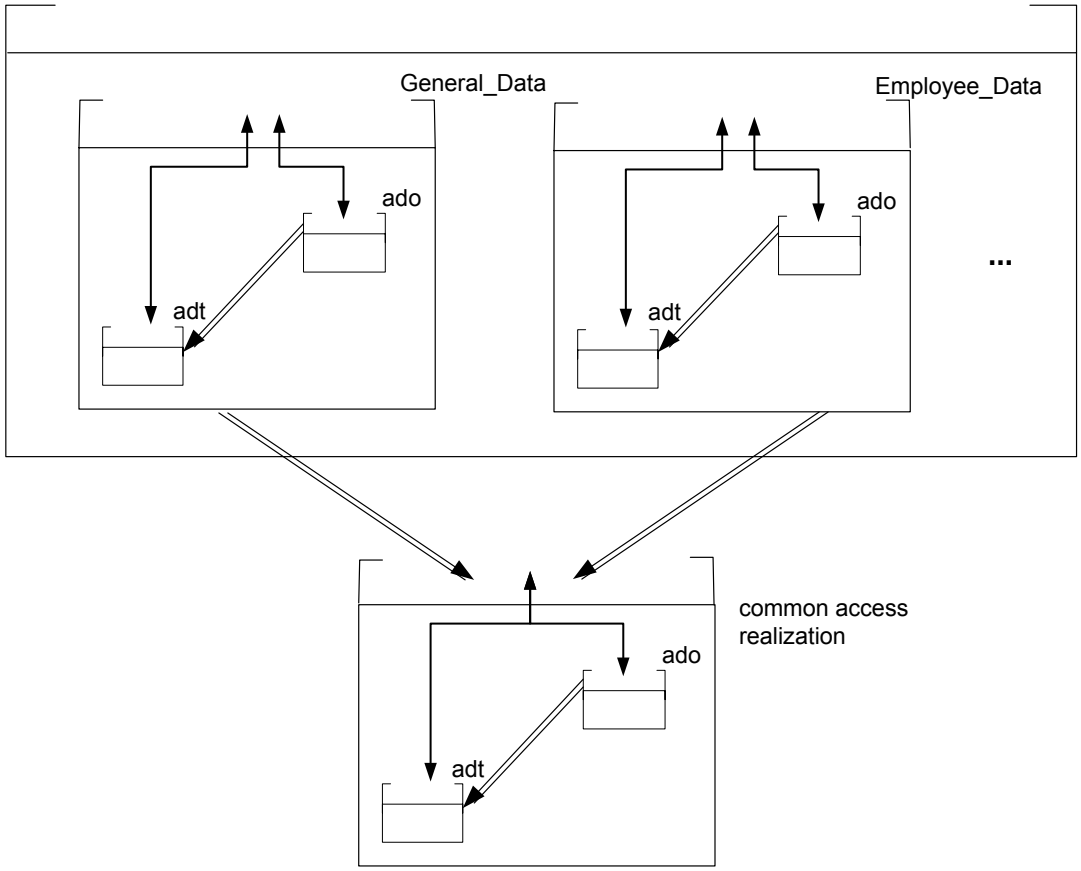
Reference Semantics for Entries



Difference Variable and Reference Semantics

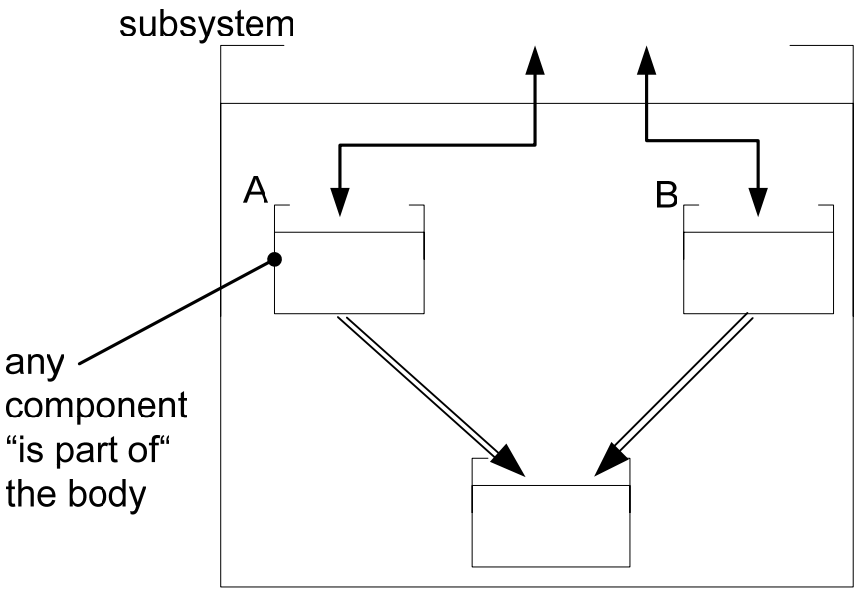
- Difference
  - » variable semantics: copying values
    reference semantics: different denotations to the same “heap“ object
  - » changing entry:
    changing one copy
    changing all “entries“ in possibly different collections (aliasing)
  - » comparing entries:
    comparing values
    comparing references
- Difference to be seen on architecture level
  - » interface entry module E
    opaque type – denoter type, creation / deletion operations
  - » eventually heap structure below entry module

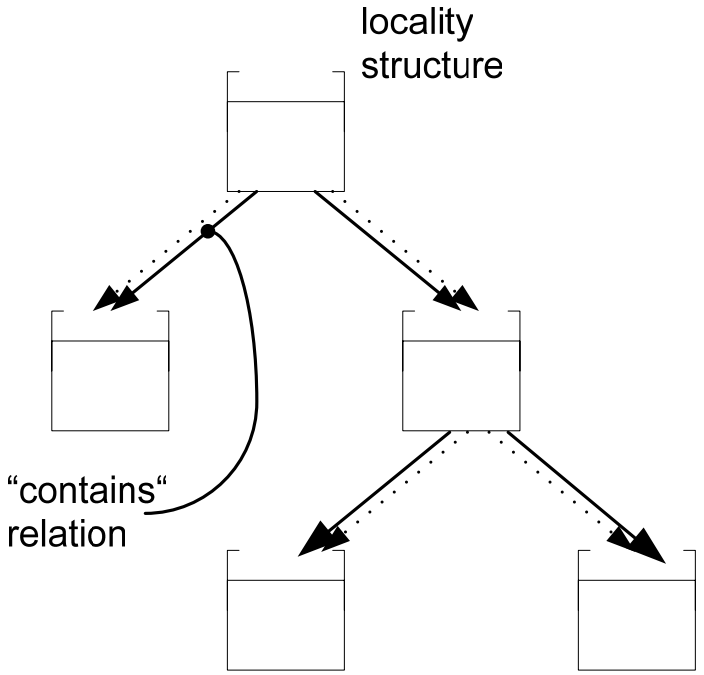
Person Data Example Revisited



- Discussion:
  - » Have aggregated entry collection situations
  - » Example shows aggregation (subsystems) of aggregation (subsystems)
  - » One layer down outside the subsystems: common realization

Hiding Subsystem´s Internals ≠ Locality





- “arbitrary“ body architecture
- body architecture

  hidden (modularity concept)
- interface composed
- no scope / visibility

- “flat“ architecture built up by locality
- interface of top module
- modules to be “seen“ but no use of inner modules
- information hiding by scope / visibility

Aggregation of Subsystems ≠ Org. Structure of Libraries

ss aggregation concept	org. structure
component from library used with realization structure	regard “domain mathematics, lists, ...
if components have tight coupling ⇨ within subsystem	big components (subsystems) together with realization in library or bookkeeping of what library units´ realization consists of
if needed for information hiding: within subsystem, otherwise subarchitecture	org. structure also for aggregation of interfaces (Ada 95)

so different

but used for similar purposes

"Kind" of a Subsystem

- modules f, (ft), ado, adt: kind of subsystems ?
- all interface modules of a subsystem of same kind
  ⟹ subsystem of this kind
  all f (math. functions): f subsystem
  all (ado different parts of a composed state): ado subsystem
  all (adt entry and collection): adt subsystem
- if interfaces are of different kind:
  is there a dominating module ?
  entry adt, collection ado: ado subsystem
- module: determined

  subsystem: often determined

  complete system: mixed
  batch system: functional
  interactive system: data layer: ado
  functionality on top: f

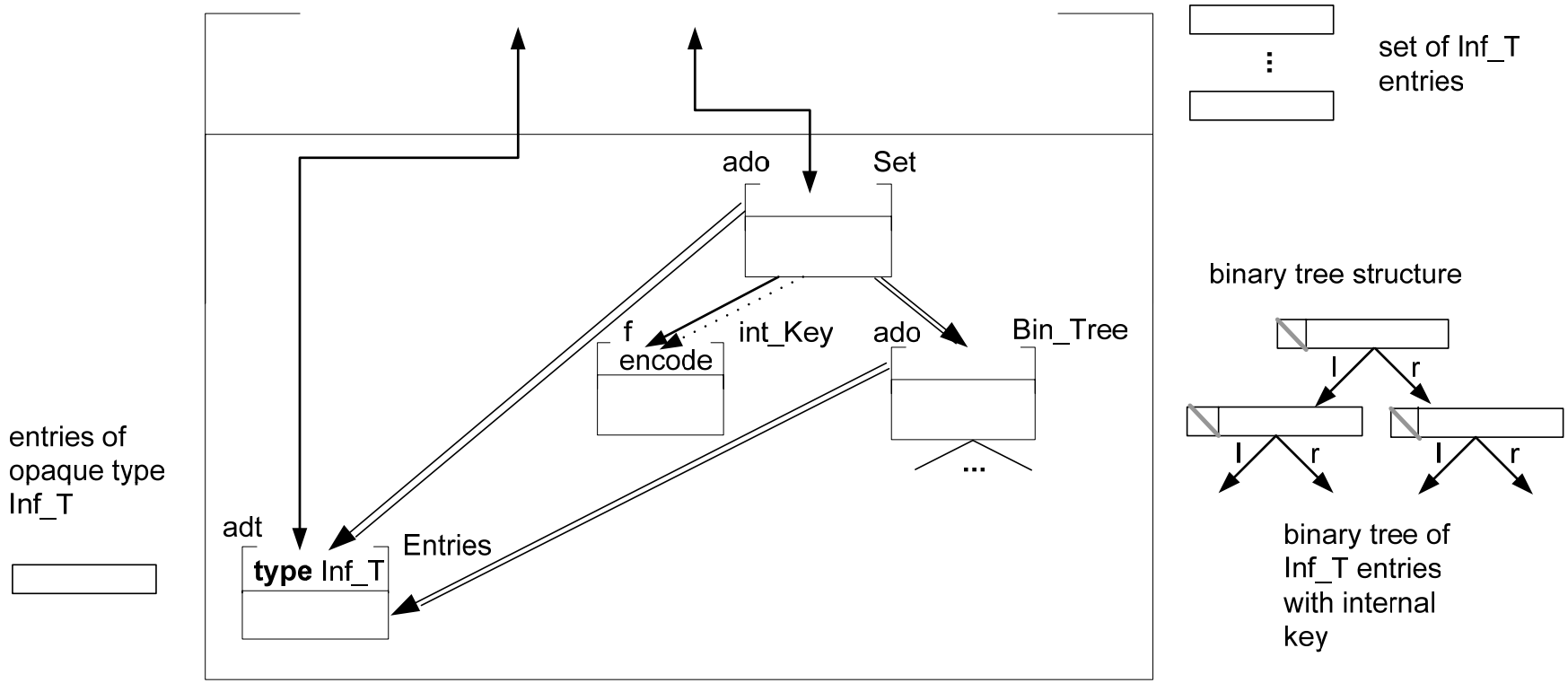
How Are Subsystems Created / Used

- Development:
  As subsystem designed and implemented
- Reuse:
  Ready-to-use subsystems become part of a solution
- Composed by Reuse:
  Reusable components composed to a subsystem
- Hiding a subarchitecture:
  Subarchitecture is put into a subsystem
- Flattening a subarchitecture:
  A subsystem´s architecture is made to a part of solution
  (a kind of reuse: copy / paste)

Motivation

- More complex entry collection situations:
  collection or entry realization across different levels
- More than one collection in the interface
- Storages for graph-like data structures

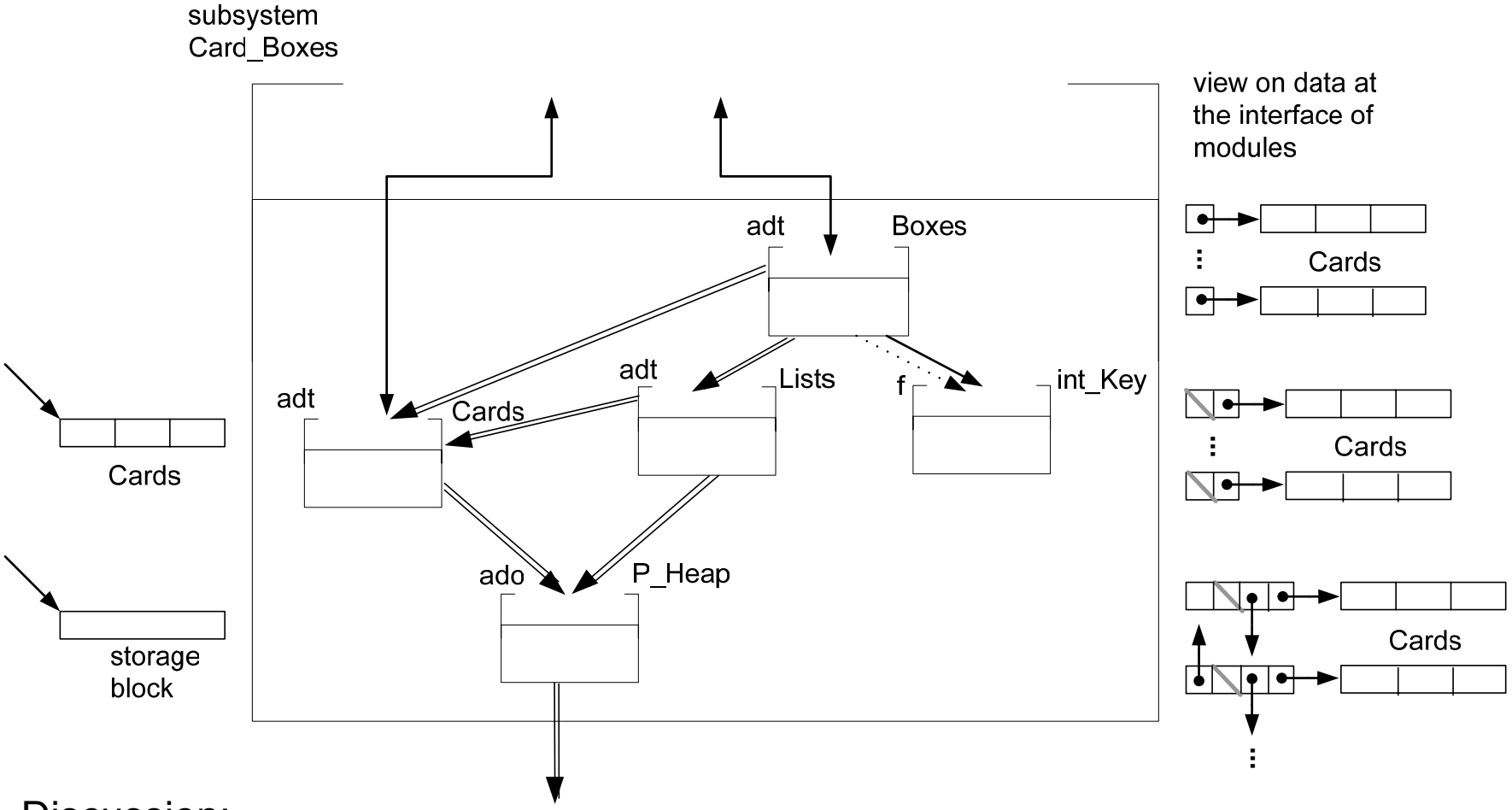
Sets, Binary Trees, ... of Entries



Discussion:

- variable semantics of entry adt
- more than two collection levels: set, tree, file
  l, r by indices of index
  index-sequential file

Card Boxes Example with Reference Semantics



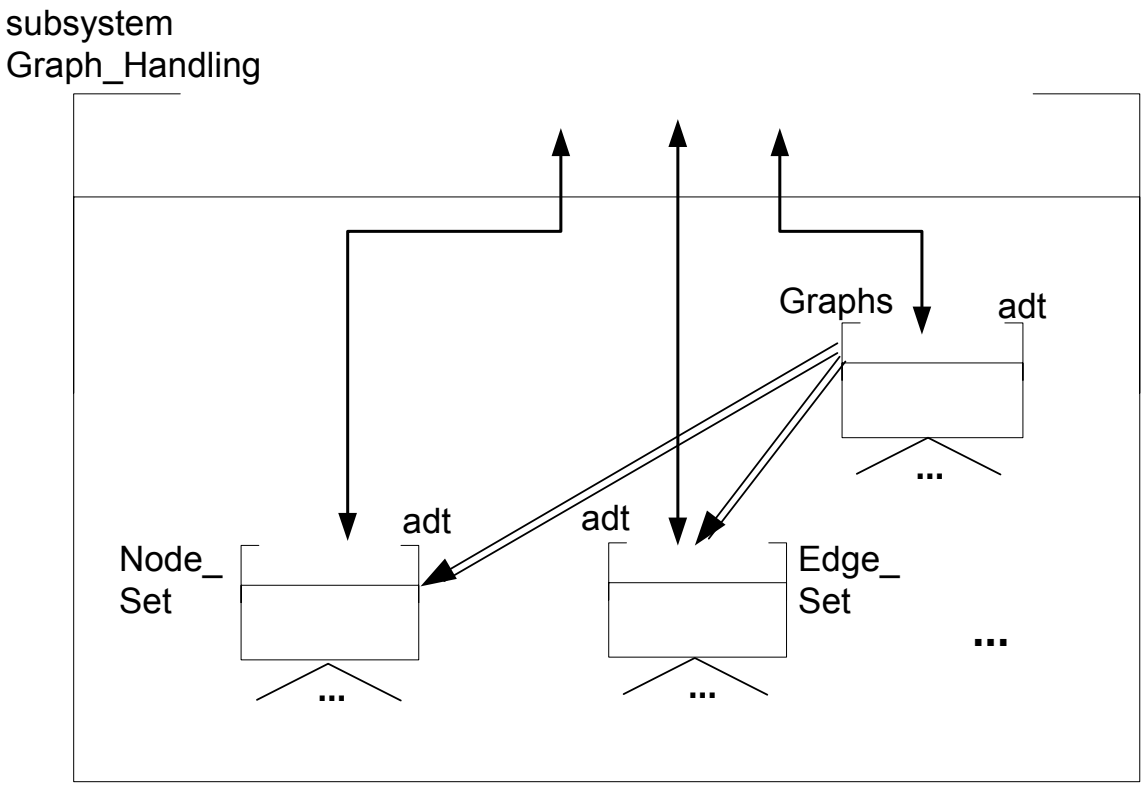
Discussion:

- collection and entry realization over different levels
- collections with simple access mechanisms

More Complicated Access Mechanisms

- associative access to entries of a collection:
  associative access, key access, path access
- entry and collection realization are intertwined
- different complexity:
  determined access (e.g. FiFo, LiFo)
  context-based via determined components (e.g. name)
  full associative access (via arbitrary component values,
  delivering a set of entries)

Graph Structures:



Discussion:

- Node_ and Edge_Set for realization
- Node_ and Edge_Set for the interface
- Client reads Node_ and Edge_Sets, can neither write nor create them

More Realistic Graph Storages

from GRAS nonstandard database system

(see e.g. Dissertation Baumann, Böklen): thousands of code pages

OO Advantages: Summary

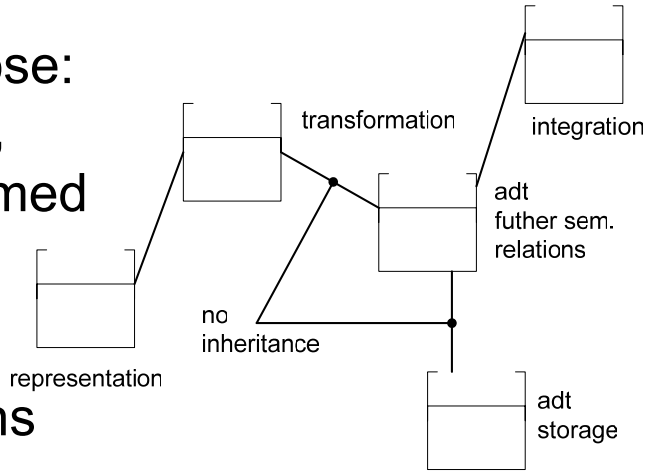
- Model similarities and thereby commonalties:
  inheritance hierarchies = classification + similarities
- Reuse approach and elaborated class libraries but ...
- Think more in system families than single systems but ...
- Think in extensibility (hierarchy extension) but ...
  (extensible languages, Smalltalk “moving target“)

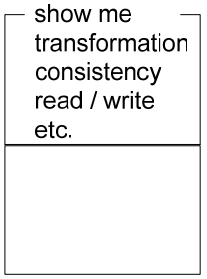
What Is Behind OO: Summary

- OO structuring is classification:
  not easy to build, extend, or restructure OO hierarchies
  (sometimes multicriterial classification needed, not expressible)
- Specific calculations, general calculations
  corresponding mechanisms (see Smalltalk, Ada 95)
  not completely statically checkable
- Need type → object generation mechanism at runtime
- Essentials: extending interface
  variant programming for bodies
  dispatching due to polymorphism

Use Rules: Not “Class Is Class“

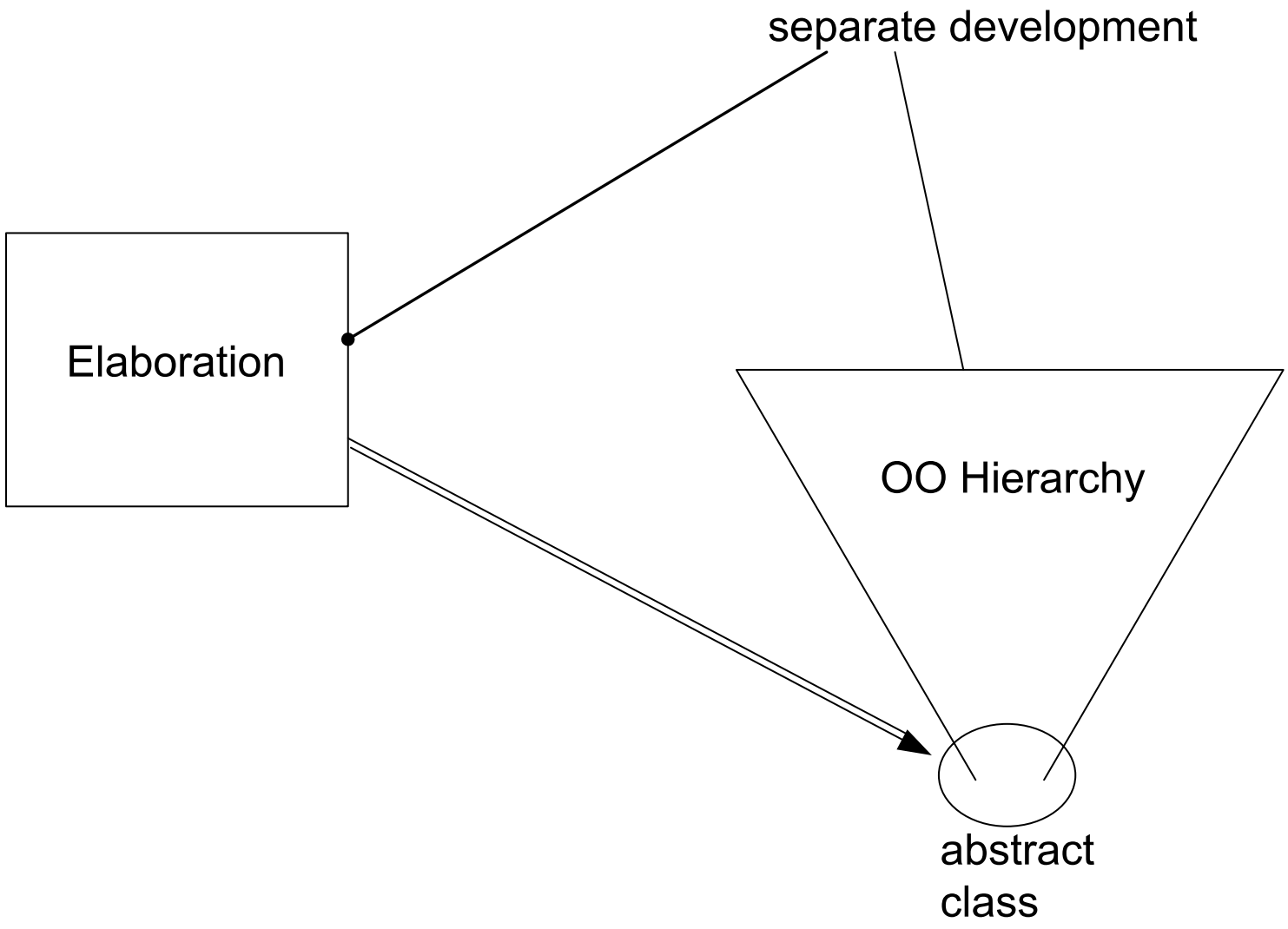
- Distinguish classes, corresponding to purpose:
  storing data, semantical relations on top, transforming data, representing transformed data, integrating with other data etc.
- So, no classes with multiple design decisions
- Instead: “class“ for coupling, transformation etc.





multiple
descision class

Separate OO Hierarchy from Elaborating Objects



How to Use Inheritance Hierarchies

How to use:

(1) leaf: most specific properties

(2) root: common properties + dispatching

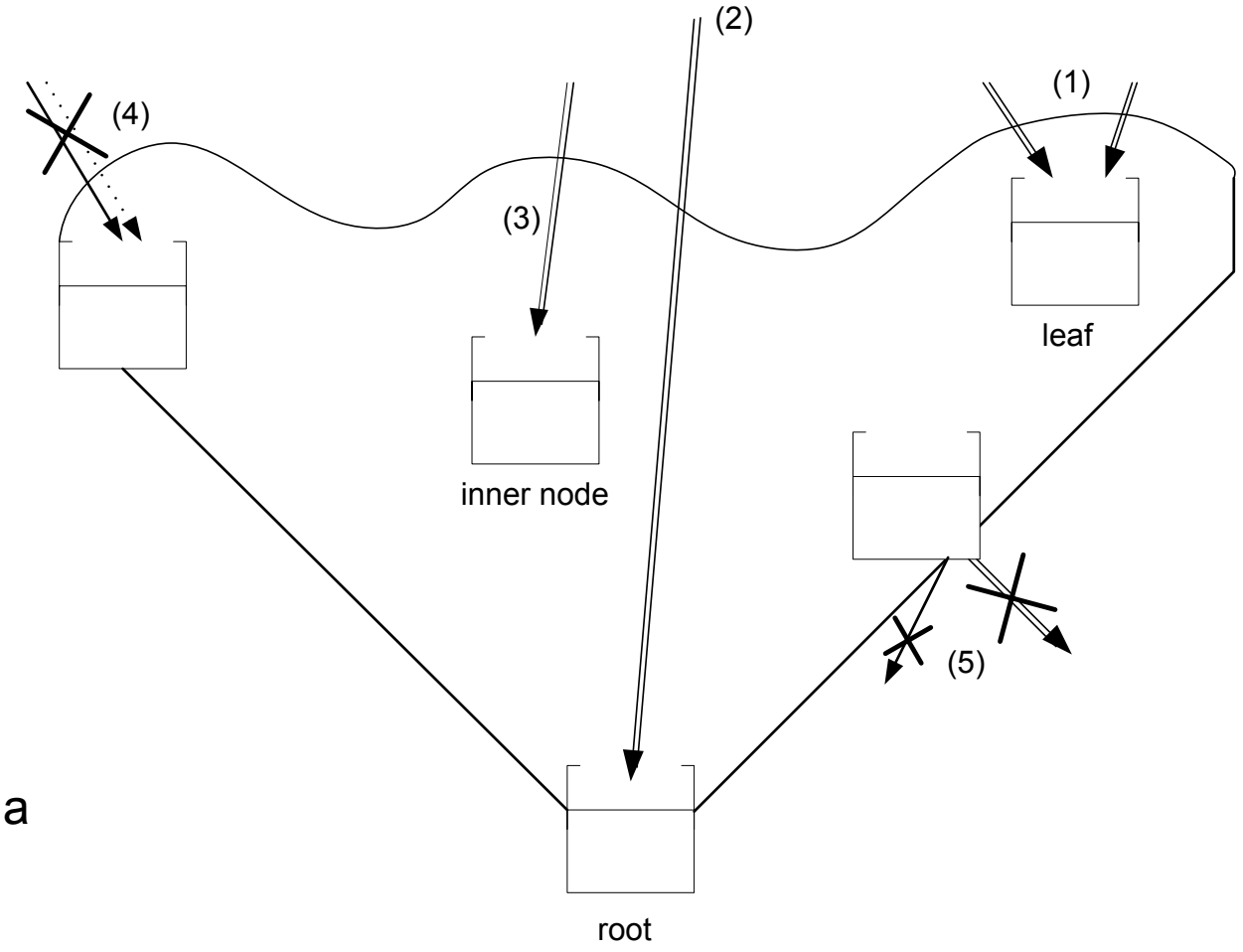
(3) inner node: intermediate case

Not possible:

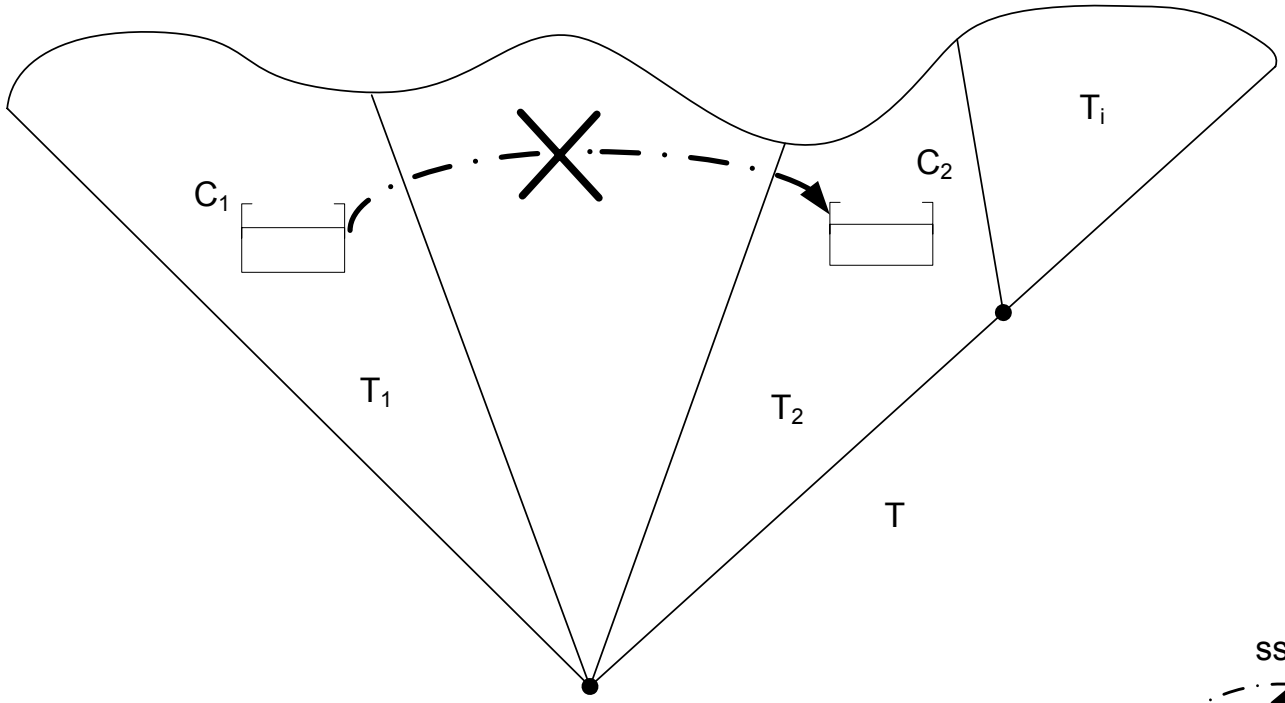
(4) classes by locality

Not recommended:

(5) implementation of a class outside hierarchy



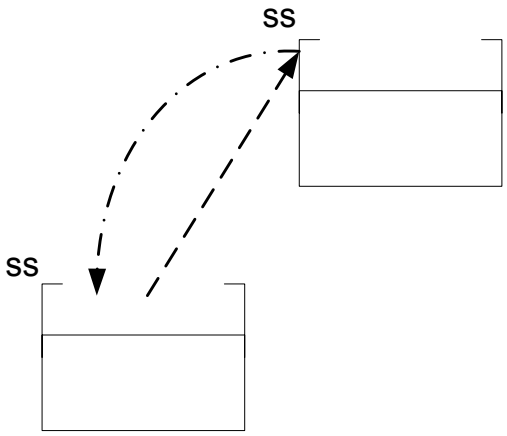
New Candidates for Subsystems



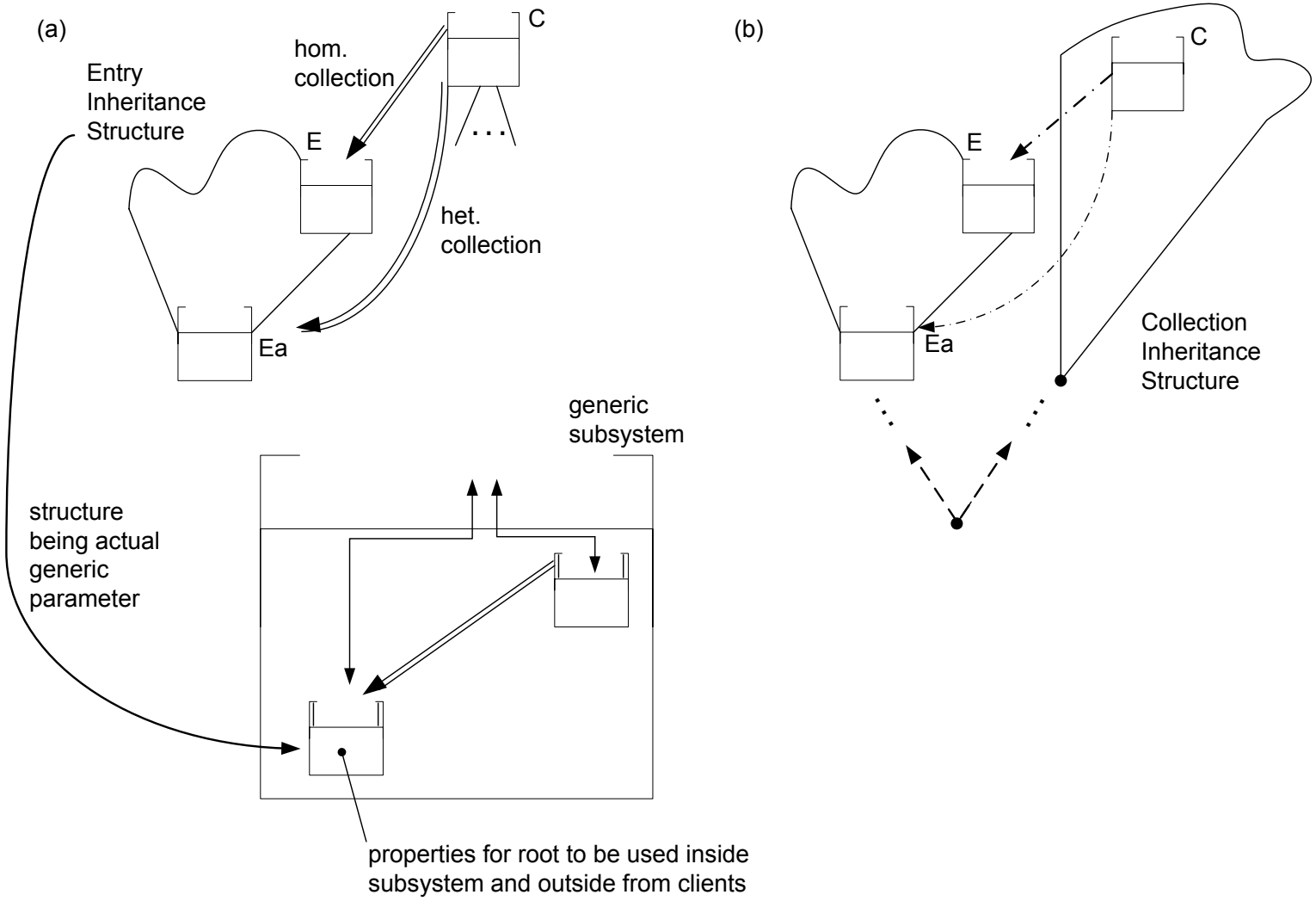
Inheritance subhierarchies T_1, T_2, T_3

Whole hierarchy T

Yields inheritance between subsystems



New Form for Entry – Collection Situation or Genericity and OO



Where to Use OO

OO = Data Abstraction + Handling Similarities

Applications

- Entries
- Collections
- Devices
- Layout
- Style

} I / O

- Geometry
- Communication Layers (ISO / OSI)
- Database Schema

OO and Component “Types“

f module = abstract class without objects

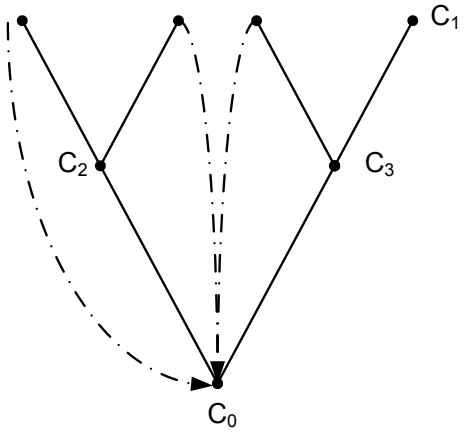
ado module = abstract class without objects

adt module (reference semantics) → class with creation for objects

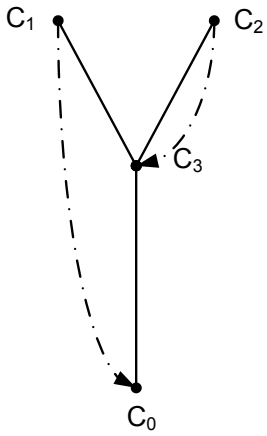
(variable semantics) → class with declaration for objects

general usability	usually available
inheritance	available
inheritance import	available or by “simulation“
locality	by general usability and discipline
subsystems	if not available by “simulation“

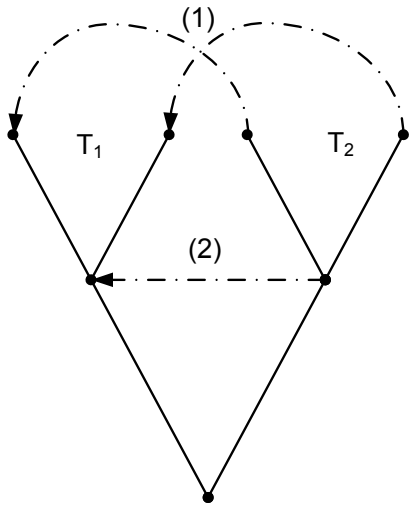
Design Errors in Inheritance Structures ?



Why not (C_1, C_0) ?
Why not (C_2, C_0), (C_3, C_0) ?



Why not (C_1, C_3),
(C_3, C_0), (C_2, C_0) ?



(1):
Inheritance structures T_1, T_2 not really related to each other: better general usability.

(2):
Simulation of genericity via inheritance ?
Similarities in inheritance not modeled ?

Problems

- Class is class: no distinction of components according to purpose
- Component has one design decision – often violated:
  data representation, consistency of data, transformation of data, presentation of transformed data, integration with other data
- Only “types“ on architecture level, no “objects“:
  objects only on “heap“, no memory / state components, big abstraction from reality
- Interaction of objects fixed in classes´ code:
  in process control: objects of classes are related to each other at design time
- No strict information hiding:
  see inheritance structure below an interface, as interfaces of this structure are also applicable

Problems Continued

- Genericity not available: bad simulation
- OO is classification:

  no good means for integration
  - coupling logical data, representation data, presentation data
  - integration between logical data (integration documents)
  - graph data structures different from nodes or edges
- Class hierarchies and their usability:

  mirror complexity, show design errors, allow to estimate changes and to manage changes
- Reorganisation of inheritance structures is difficult

  reengineering, extension, melting
- OO needs subsystems: mostly missing

OO and Architectures

- Architecture is statical, essential structure:
  OO structure only on “type“ level
- Class hierarchy does not contain much information
  (see batch or dialog example)
- Predefined classes and inheritance structure:
  how to compose to a solution ?
- UML / UP: Many diagrams – where is the architecture ?
- Bigger units of architectures
  (see application class engineering, structure class engineering)

frameworks	are distributed
basic layers	over inheritance
big generic components	structures

OO and Architectures: Continued

- Detection of similarities:
  dialog system for (a) addresses, (b) bibliographic references etc
    looking on inheritance structures: who detects that (a), (b) are the same
  compiler construction:
    who detects that there is a complete and universal built plan such that only tables are to be exchanged
- Architecture is interaction of many components
  development of a conceptual frame, reusable layers,

  design of such structures
  their adaptation using genericity
  automation by code generation
  table – driven architectures

  } these ideas are not supported by OO

OO and Reuse: Claim and Reality

- Worldwide class library: a realistic scenario ?
- Solution by standardization: not only components (classes)
  but also their superclasses have to be standardized
- Redundancy in inheritance structures:
  reengineering or code multiplication ?
  need some classes which come with inheritance structures
    reuse more than needed
    or difficulty to melt inheritance structures

OO and Real Life of Software Engineering

- no reengineering projects (about 85% of practical life):
  - there is an old system to be reverse engineered
  - this system is reengineered
  - further extensions are made
  - the system is integrated with others
  - the old system is written in Cobol, C, FORTRAN, so not OO: what to do?
- How to deal with embedded systems (see automotive):
  - hardware means thinking in objects, not types
  - there is no object creation at runtime for a type (class) for hardware
  - there is no universal heap in a distributed system
  - structuring in parameterized object structures is missing
  - distribution is easier for objects than for types

Resumée OO

- OO contains valuable ideas to be applied at the right places
- OO does not support architectures of many components
- Practical life development
  embedded systems: not easy
- Reuse is more a claim than reality

⇨ OO is a part within a comprehensive instrumentarium

Architectural Language´s Applications to Subarchitectures

- Distinguishing functional and data abstraction
- Interaction of f, ado, adt modules
- Data abstraction layers
- Multiple data abstraction
- Subsystems use
- OO Rules of Good Use
- OO: Critical Discussion

Method Extension

- In addition to language methodology we now have
  - Use methodology / rules
    size of modules
    bad connectivity situations
    etc.
  - Knowledge of good patterns: how to use
    f – ado – f
    f – f – f adts in between
    da layers etc.
  - Transformation patterns
    ado situation → adt situation,
    code → tables
    wrong → right situation
- Application methodology
  - Data abstraction applications: entry, collection, …
  - OO is data abstraction + similarity