Software Architecture – summary

Contents

Ch00 1
Ch01 1
Ch02 3
Ch03 6
Ch04 8
Ch05 18
Ch06 21
Ch07 22
Ch08 23
Ch09 25
Exercises 26

Ch00

- Nothing

Ch01

- Software engineering definitions:
  SE means to produce software economically, which runs efficiently and reliably on real computers;
  SE is the discipline of Computer Science which deals with supply and systematic use of methods and tools for the development or use of application software
- Software engineering characteristics:
  Development of large software is qualitatively different from small software
  Problems only if:
  1. Many developers cooperate
  2. Different versions of program system exist

  Cooperation in the project team:
  1. Planning and distribution of labor
  2. Tasks / solutions, interaction / integration
  3. Quality assurance

4. Documentation

- Life cycle models:
  Activities: phases
  Results: documents (result of subtask in development process, documents are mutually dependent)
- Reason for phases:
  1. Complexity reduction
  2. Quality assurance
- Phase model: time relation ("is result of ", "is necessary input for")
- Activity areas:
  1. Modelling on the same logical level
  2. Collecting similar activities occurring at different points of time scale
  3. Prerequisite for discussing the relations between similar and different tasks
  4. No distinction between construction and modification
- Working areas:
  1. Defining / modifying outside behavior / requirements: requirements engineering
  2. Defining / modifying essential structure of system according to requirements: architecture modelling (design, prog. in the large)
  3. Defining / modifying detailed realization: implementation (programming in the small)
- In any phase:
  1. Analysis for construction or verification
  2. Stepwise construction, internal analysis, analysis back and forward
  3. Final check back and forward
- Programming in the large/ design/ architecture modelling
  1. Analyze requirements specification under design aspects
  2. Stepwise design using modules and subsystems
  3. Stepwise verification of resulting design specification fragments: internal, against requirements specification, feasibility for implementation, integration, and maintenance
  4. Final backward verification against requirements specification
  5. Final forward evaluation against realizability
  6. Formulation of design specification in a programming language: Coding in the Large
  7. Integration and functional check of modules, subsystems, overall system corresponding to the software architecture
  8. Performance test of modules, subsystems, and overall system …
  9. Installation of overall system from components …
  10. Change of architecture (esp. during maintenance) by repeating the above steps
  11. …
- Working areas most important:
  Requirements engineering: build the right system
  Architecture: maintainability; adaptability; portability; reusability
  Management: costs and risks

Ch02

- Architecture paradigm:
  Ideal paradigm:
      Design before implementation
      All components of all layers: only interfaces
  Problems:
  1. Identification of all layers and all components
  2. Changes will happen
     -Components too complex
     -Trivial components
     -Layers appear/disappear

  Continuous paradigm:
  Problems:
  1. No division of labor
  2. No architecture:
     -No quality assurance before realization
     -No reuse of components
- Design strategies:
  1. Top-down
  2. Bottom-up
  3. Jo-Jo (a hybrid)
- <u>Error principal definition:</u>
  Wrong: statically wrong, principally wrong
  Error: anywhere, appearing anytime
  <u>Errors practical definition:</u>
  Time from error to error: acceptable
  Error correction effort < benefit of system runs
- Types of errors
  Requirements:
  1. Requirements specification not reflecting needs
  2. Requirements specification incomplete, contradicting

  Architecture:
  1. Does not meet requirements
  2. Does not reflect software engineering quality measures: adaptability, portability, etc.

  Implementation:
  1. Component realization not consistent to design specification
  2. Realization wrong or inefficient

  Documentation:
  1. Analysis and design decisions not documented
  2. Documentation not consistent with technical system

Project organization:

1. Bad estimation, qualified developers unavailable, no clear responsibilities, no precise monitoring, no risk analysis

- Conclusions

  More effort and competence:

  1. More effort for early phases
  2. Suitable notations / methods necessary
  3. Experience and tools

  Better quality assurance:

  1. Inspection of design results
  2. capability to evaluate design results *

  Modifiable architectures:

  1. Errors cause changes
  2. Changes are likely to happen
  3. Demand for suitable architectures
- Architecture characteristics
  1. Has to map requirements
  2. Determines long-term quality of system
  3. One architecture can answer different requirements
  4. Organization of development
  5. Organization of change
  6. Design errors have huge influence
- Architecture aims
  1. To estimate the effort of changes and to plan changes
  2. To carry out changes
  3. To study the structure of a system
  4. To identify reusable components, patterns, or standard solutions
  5. To define subprocesses in other working areas: implementation, quality assurance, integration, and documentation
- <u>Architecture parts:</u>

  Graphical notation: for overview

  Textual notation: for design details (i.e. interface description)
- Architecture vs project organization

	Architecture	Project organization
Tasks	Programming in the large (i.e. Design)	Programming in the many (Project organization)
	Control of versions, variants, configurations	Organization of versions / variants, configurations esp. important for architecture and units
Roles	Technical side: chief programmer / architect	Organizational side: project manager

  These two people maybe the same person

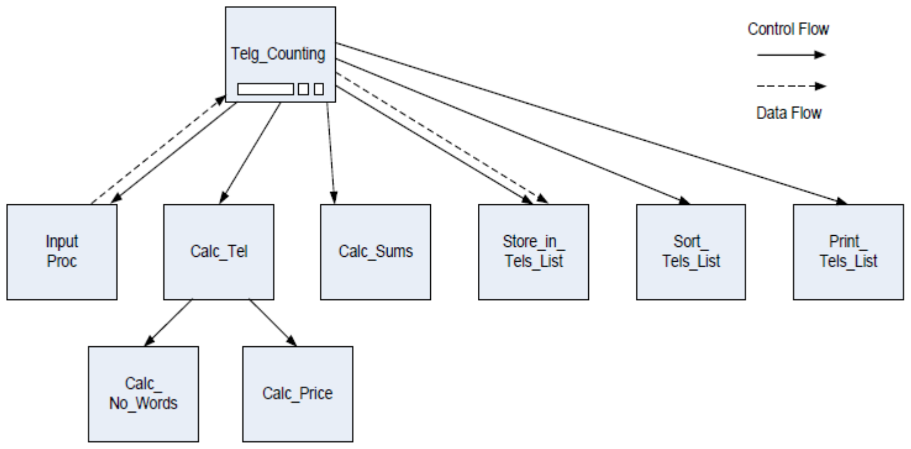
- There is no uniform built plan (i.e. there are many forms of architecture), no clear separation of requirements specification and architecture

- Architecture contains "type" of units necessary
  User is not an architecture unit, but the interface to the user is
  Data objects are not units but a component to control these units
- Architecture characteristics summary:
  1. Built plan as result of the design subprocess
  2. Complete plan: all components, relations, and layers
  3. Consistent to requirements specification, but different from that
  4. Plan contains all relevant design decisions
  5. Abstraction compared to realized system (architecture paradigm)
  6. Independent from programming language, but see the restrictions of that language (e.g. no OO design if system is to be realized in C)
  7. The plan contains the structure of the system: that part OO, another modular, that layer abstracts from, …
  8. The plan can be evaluated: portability, adaptability, …
  9. Architectures are different: application domain, classes of systems, quality of design subprocess, used target system, etc.
  10. A good architecture is the result of an intellectual process: not automatic
- Architecture is not necessarily only a static plan or an abstract plan
  But furthermore:
  1. Dynamic properties: denote typical execution paths
  2. Estimate future system properties: runtime behavior, deadlocks in concurrent systems, satisfiability of time constraints
  3. Describe properties of concurrent systems
  4. Describe distribution aspects
  5. Etc.
- Architecture is one big plan but with different abstractions

Part	Language
Graphical overview	design (architecture) diagrams
Textual detailed descriptions	textual MIL language
Different subplans for essential parts	graphical & textual
Different further aspects: 1. Concurrency 2. Runtime behavior 3. Semantics of components 4. Distribution	"annotation" language
Design rationale (道理) as part of technical documentation	Natural language explanation
"Specification" for use/purpose, of modules / subsystems	natural language, decision tables, finite automata

Ch03

- Small batch system: functional decomposition



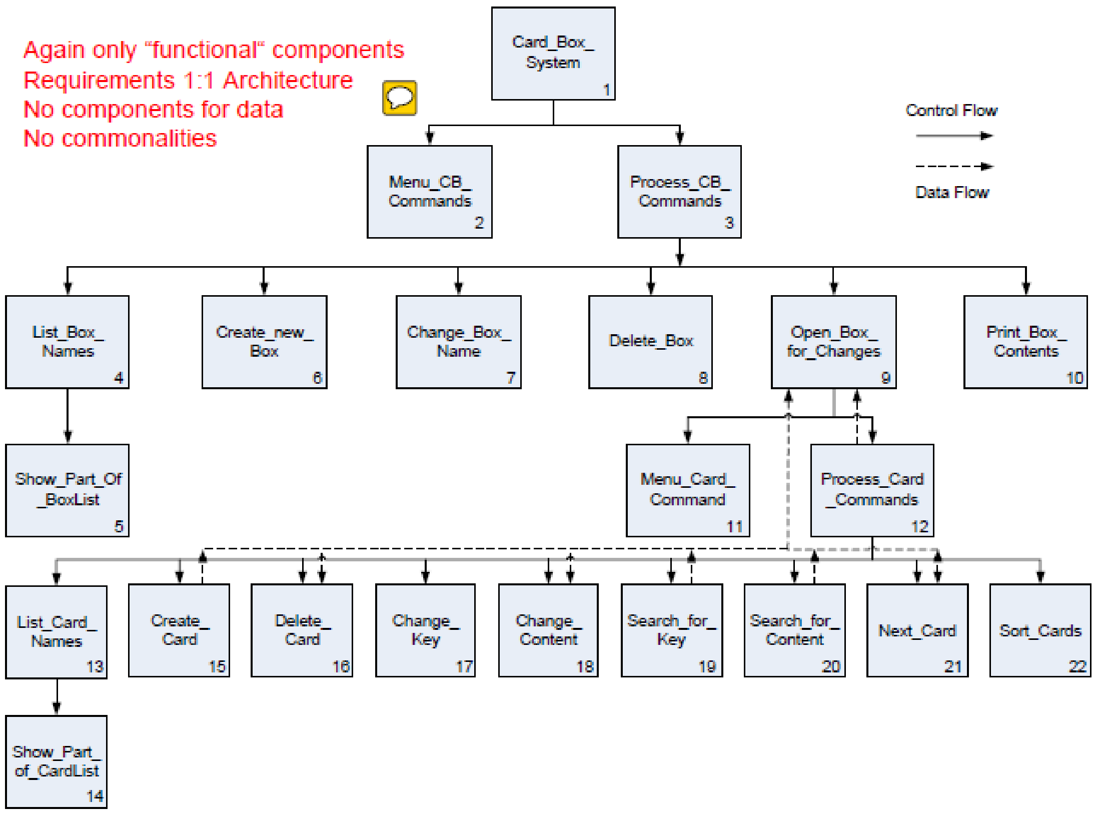
  Characteristics:

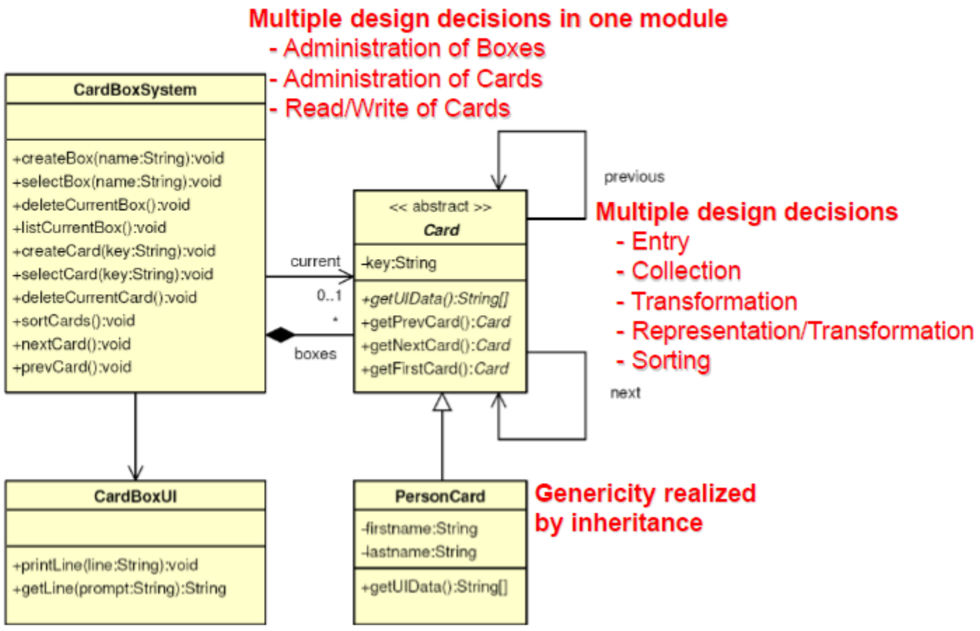
  1. Has only functional components, has only → subfunction relation
  2. Architecture is tree
     -Top-down development
     -Missing components
     -Thinking only in functional abstraction
  3. Tree is partially ordered
  4. Required tasks match architecture 1:1
  5. Global data, data flows

  Mistakes:

  1. Commonalities not recognized, e.g. telegram syntax analysis
  2. Components of very different complexity
     -Calc_Sums is a two lines program
     -Input_Processing more complex, includes syntax analysis
  3. No components for data

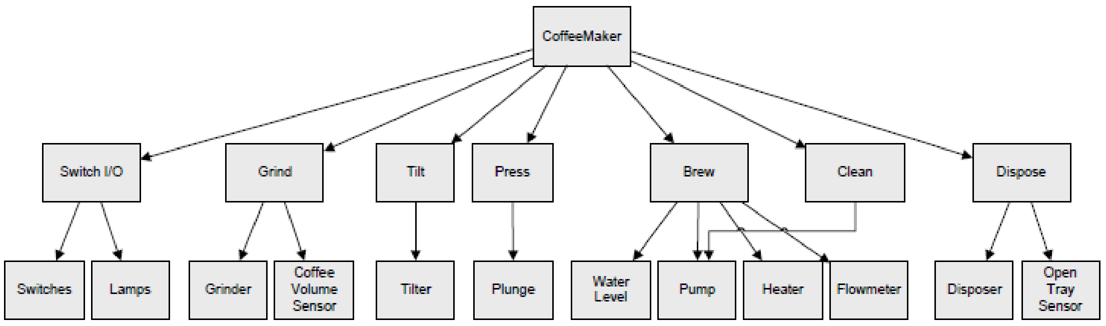
- Small interactive system: 2 solutions, functional decomposition and OO decomposition





Note: the red are the mistakes

- Small embedded system



Mistakes:

1. Use realization functionality and not abstract functionality
2. Direct usage of underlying sensors, actuators and components (e.g. switches, pump, water level sensor, etc.)
3. Mixed up standard and special functions (e.g. startup/shutdown, error and exception handling)

- Summary of above solutions

	Functional decomposition	OO decomposition
Characterization	• Tree structure, partially ordered • Global data flow • 1:1 mapping from functional requirement	• One inheritance structure, again a tree • 1:1 mapping from application entities
Mistakes	• Data components missing • Data changes cause severe system changes • Requirements changes also cause severe changes • Functionality as given: no abstraction • Devices / Sensors included: no abstractions	• Components with multiple decisions • Layering: classification, not use • Inheritance misunderstood • Requirements changes cause severe system changes

Ch04

- Module's characteristics(并不完全对，只是一些看法)
  a) Logical unit, one sentence for description
  b) Abstract machine or part of it
  c) Module has tight internal coupling and loose coupling to other modules
  d) Represents one design decision; architecture is sum of all decisions
  e) Unit consisting of data and operations
  f) Module has a certain complexity (length of spec, length of source code)

g) Offers resources to clients: interface components simple and orthogonal
h) Module realization (body) hidden
i) For realization import from other modules
j) Module is side-effect free: What it does can completely be seen from the resources of the interface (and imports)
k) Module can be exchanged by another with the same interface: no influence for semantics (but for pragmatics)
l) Correctness can be proven without knowledge of its use
m) Unit of independent development (project organization)
n) Unit of reuse
o) Unit of compilation

- Modules are <u>logical</u> units:
  a) Not implementation units
     as subprograms, macros, …(这俩是 implementation unit)
     are (may be) used for module bodies
  b) Not programming language units
     as interface module, implementation module of Modula-2
     both build up a module(这俩是语言 unit)
  c) Not realization details in the first step:
     sequential, concurrent, reentrant, distributed
     these details are regarded later
- Information hiding of details of the body
  1. Abstraction from body details: export interface
  2. Basis of the “Architecture Paradigm”
  3. Client does not want to see and must not see body details
  4. Basis of realization exchange and of reuse
- Module has exports interfaces, imports are relations
  Use textual description language (module interface connection language), syntactical and semantical descript of export interfaces and import clauses
- Modules have different types
  Functional abstraction: “type”: Functional module or shorter function module
  Data abstraction: “types”: (abstract) data object module, (abstract) data type module
- Functional Abstraction
  Characterization of functional modules:
  1. transform input to output data
  2. are “activity-oriented”
  3. I/O data appear in the export interface
  4. can have more than one operation in the interface
  5. no state (memory-less)

  Formalization: pre and post condition for every operation(用前后状态定义操作)
  Examples:
  1. compiler, also phases of a multi-pass compiler
  2. mathematical functions
  3. control of a user dialogue

4. main program

Typical applications for function modules:

1. transformation (e.g. lexical scan)
2. computation (e.g. an evaluation)
3. control (e.g. of a dialogue)
4. coordination (multi-pass compiler)
5. auxiliary service on data (e.g. navigation on a search tree)
6. coupling / integration (e.g. representation / logical data, see model view controller)

About functional abstraction:

1. Clear architectures: Demand for explicit units serving for integration or coupling
2. Understanding it is easy, right application difficult:
   -distinction between functional and data abstraction
   -right functional abstraction together with data abstraction
3. Special case: one operation in the interface: subprogram specification as "module interface"(this case, no specific formulation is introduced)

- Function module, example, in p23
  <u>Only one operation in the interface</u> of the function module:
  e.g. the numerical calculation of a mathematical function
  may nevertheless be complicated and use further modules
  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)
- Data abstraction
  Characterization of data modules:
  a) follow data abstraction (or data encapsulation) principle:
  See only operations on data and not the detailed representation of data or operation realization
  b) have a state (memory)
  c) are "passive"
  Formalization:
  -logical pre and post conditions
  State before, state after
  -or algebraic equations
  POP ○ PUSH = ID
  Principles:
  a) no direct access to data structures
  b) access only via logical operations: interface
  c) operations (updates, reads) and data representation are one unit
  d) data representation details hidden
  e) details are only used by realization of operations
  f) different realizations (data representation, corresponding operations

implementation) possible, realizations exchangeable

Origin: algebraic specification, abstract data types from Theoretical Computer Science
Use: makes any architecture adaptable changes of data, also other changes easier

- How abstract is an ado / adt module?
  a) interface more abstract than realization
  b) abstraction from a variety of realizations
  c) data abstraction modules often in **lower** parts of an architecture
- Abstract data object module, example, p24
  functional module 前面没有写 abstract，ado 就有写。
  有返回值的叫 function，没有的叫 procedure(functional module 也一样)
  Interface: Access operations FIND, STORE, CHANGE
  Body:
  realization of data structures (here on heap)
  realization of access operations(这两个 realization mutually dependent)
- Broad entry
  +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
- We should localize dangerous situations, make them within data abstraction module.(e.g. pointers, which can result in aliasing, inaccessible objects and dangling references)
- Data abstraction produces adaptability(自己内部改，别人不用改) and allow different realizations
  No longer distinction between programs and data: we have ado, adt
  Architecture contains the complete structure of the program system
- Ado security:
  1. Add checks and execute only when they are met
  2. Execute and raise exception in the end or return success = false, success parameters is used for efficiency or in concurrency context

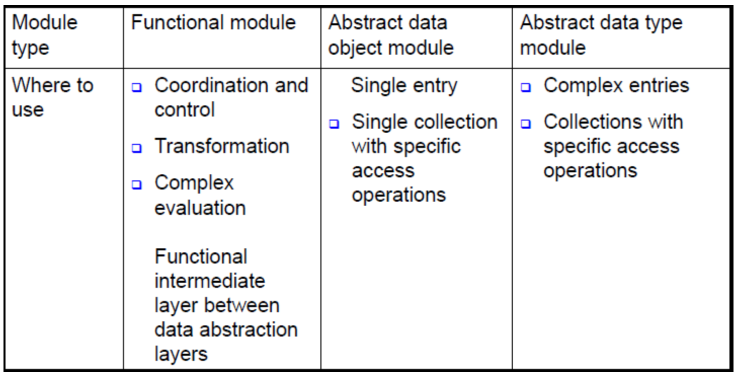
  The 1st one is more secure.
- If a query operation is used as a security query(就是验证什么是否操作得当), then there must also be a corresponding exception
- 2 Standard application of data abstraction
  Complex entry: aggregate components to a logical component
  Collection: set of entries, 就是实现 list, stack, heap, network node 这种
- Adt, abstract(opaque, encapsulated) type vs open(transparent, normal 就普通的数据类型) type
- Task of 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 module is an ado; adt module is a template for ados
- adts with variable semantics
  ados are generated when declaration is elaborated in the body of a client module
  就是用的人直接声明几个变量 ado，需要 underlying programming language knows type declarations
  Number of ados known at <u>compile time</u>
- adts with reference semantics
  adt create and delete ados
  client use in the statement part(creation, initialization,…感觉就是通过 adt 来操作,adt 会记住所有 ados)
  number of generated ados is <u>determined at runtime</u>
- Client programmer <u>must know</u> whether adt has variable or reference semantics.
- Comparison:

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

- Goodness of adts(same for ado)
  1. Localize dangers
  2. Loose coupling between modules
  3. Adaptability
  4. Many different realization possibilities
  5. Applications: entries, collections
  6. How to group interface operations
  7. Interfaces and security
  8. Information Hiding (body interface) is used for a semantical purpose
- When to use f,ado,adt



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

- Relations are on usability (or import) level.
- If imported module is
  1. f: usability of functions
  2. ado: usability of access operation
  3. adt: usability of type / creating operations and / or access operations

- contains-relation: 单实体箭头，我用谁实现我自己, no nesting
- local usability: 点虚线单箭头，我可以被谁用（只画有道理的），可以指向自己
- general usability: 双实线箭头

  General (or common) module：
  1. designed with the purpose of multiple use
  2. or later generalized
  3. more often <u>**data abstraction**</u> than functional module

  Hang a common module into an architecture
  1. need <u>not know realization</u>: Information hiding on architecture level
  2. <u>has to know</u> whether local or general use
- local usability: bound to underlying contains-relation

  general usability: no underlying structure relation
- 3 concepts necessary for OO:
  1. extending interfaces (programming by extension): is_a – relation
  2. variant (or difference) programming
  3. uniform processing elements of similar type: dispatching
- object-based: use of data abstraction: ados

  adt-based: use of adts

  object-oriented: use of <u>inheritance</u> and <u>dispatching</u>
- OO needs to be integrated with others(locality, general usability etc.)
- Below are equivalent terms

OO	Terms
Object	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

- Is_a relation: adt employee extends adt person
- Variant programming:

  Make use of procedure cal_salary of Person in employee’s own cal_salary
- Example of OO:

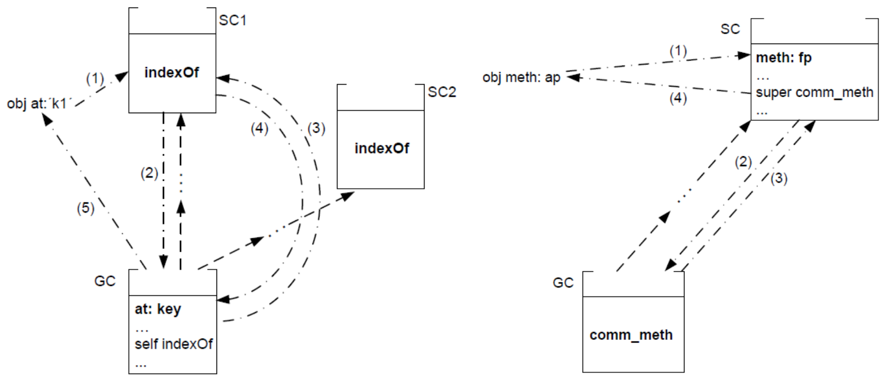
```
class Employee is a Person;
end Employee;
method calc_salary
begin
        self.Person.calc_salary(S);
end calc_salary
```
- OO is_a 是单虚线箭头，从 super class 指向 subclass
- This/Self, super: 这里就是自己没有就找 super class 里，不过注意，如果 super class 里函数是 self.test,而你自己有 test，那么就是你调用自己的 test 的结果
- Dispatching: o.calc_salary 会自动找 o 所在的那个 class 来调用
- Static vs dynamic binding

  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
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

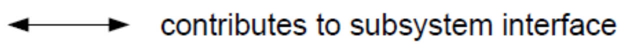
- Abstract classes(就是 java 中的 abstract class, has abstract type and abstract procedure)
    1. Abstract class has no objects
    2. “abstract” different from meaning “abstract” in adts
    3. common protocol for an inheritance structure
    4. abstract procedures have no body

  pro: uniform processing of objects can be formulated independent of class hierarchy
- OO has great reuse
    1. extension: reusing <u>interface of superclass</u>, difference to specify
    2. variant programming: reuse of <u>method code</u>, difference to program
    3. dispatching / dynamic binding: common scheme for processing elements without regarding specific types (不用写 case – statements)
    4. reusing classes for defining objects
    5. reusing class hierarchies by extending them specifically
- subclass c3, superclass c2; c3 is specialization of c2, c2 is generalization of c3
- inheritance is one concept for layering besides others(就是可以作为分层依据)
- single inheritance: tree; multiple inheritance: dags(directed acyclic graph)
- calculation mechanism 调用原理



短实线是 is_a relation；点虚线调用函数
左边是自己没有的找父类，但最后是 self 的情况。注意 4 回到父类，因为是对父类 self 调用的应答
右边是自己没有的通过 super 找父类，父类完成并返回

- OO is bottom-up development
- Reuse difficulties in OO:
    1. make clear which reusable classes exist
    2. extend inheritance hierarchy (thereby existing classes may be changed)
    3. all hierarchies are open for extension, irrespective of systems under development belongs to
    4. extensions may not be interesting for everybody
    5. reuse demands for “worldwide” hierarchy administration: unrealistic

    6. if only specific classes are needed: how to get the specific subhierarchies
    7. melting different inheritance hierarchies: difficult
- OO only has adt modules, only has is_a relation to model similarities
  OO is the opposite of locality(local importance, local use)
  虽然两个都是树结构，但含义差别很大
  OO and general usability looks similar, but structure differently(感觉这两个比较就是废话)
  Traditional architectures(指 local/general usability, contains) hide realization; while OO let you see part of realization(毕竟你要各种 super)
- Subsystem:
  A subsystem is a collection of logically related modules put into a new coarser unit. (need interfaces of the subsystem) It is determined which (resources of which) modules build up the interface of the subsystem. The internal subarchitecture is hidden.
- 3 typical examples(就是 subsystem 总共三大类)
    1. Subtrees of contains tree
    2. Aggregation of related interfaces
    3. Arbitrary subarchitecture with distinction between outside and inside
- 实线双箭头,i.e.用于 interface 了 

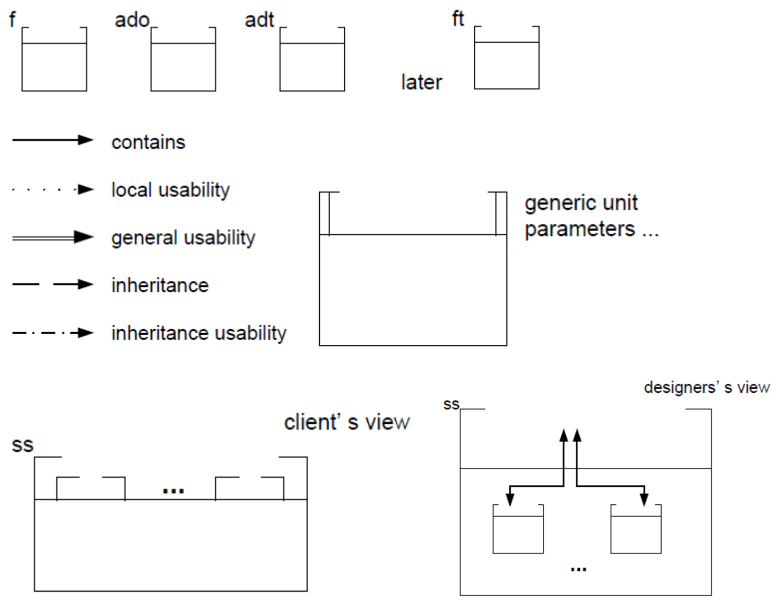
- Two-level design suffices in most time: 1, overview design; 2, big component design
- Module vs subsystems
  Analogy:
    1. export interface
    2. export interface is part of the body
    3. body hidden
    4. body allows different structuring principles, global and local resources
    5. import interface introduces resources of other components
  Difference:
    1. subsystems have aggregated interfaces
    2. subsystems realization includes design and later implementation
       modules are immediately implemented
    3. subsystems introduce hierarchical design, modules flat design
       however, modules appear on any design level
- Genericity:
  Integer_stack is an ado, G_item_stack is a generic ado
  Integer_stack is a generic instance
- 其实就是 java 里面的泛型
  Genericity and parameterization:
  Generic unit is a template, and is only designed and implemented once
  Generic unit→ specific unit→ bind into system
- Generic unit is no architectural unit(就是 template,并不实际在架构里面的实例 unit)
  By instantiation of it we get a unit
  Genericity corresponds to design process rather than design results
- Genericity in programming languages can have different mechanism:

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

- Generic ado→ado
  Generic adt→ adt - - ->ados
- Generic can be applied on subarchitecture or subsystems
- Architecture diagrams:
  a) overview to understand a complete system or its subsystems
  b) to discuss their structure (here locality structure, there subsystem or inheritance hierarchy)
  c) to detect reusable components, local or global reuse "patterns"
  d) experienced designer makes only diagrams in the first steps
  e) big system: overview diagram, subsystem diagrams(就是可以有好几个图，一个大的，和几个各子系统的)
- architecture diagram language
  components:
  a) modules: f, ado, adt, ft
  b) subsystems
  c) generic components and their instantiations

  Relations:
  a) structure relations: contains, in_a
  b) import relations: local import, general import, inheritance import

  consistency conditions
- 图例



- MIL: module interconnection language(detailed context-free syntax description)

```
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

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
```

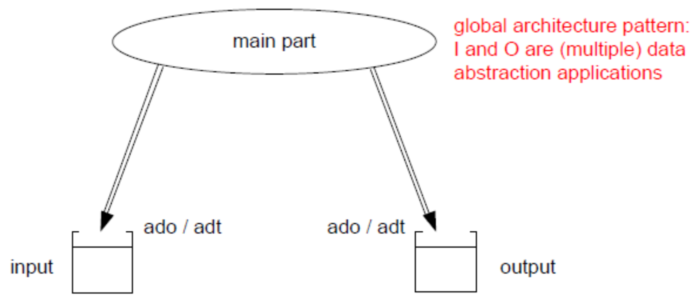
- Consistency constraints = context sensitive syntax of architectural languages and their relation to programming language (for bodies)
  Here, we focus on consistency constraints to be seen on:
  a) Graphical level(architecture diagram language)
  b) Detailed specifications level(MIL)
  c) Or regarding realization of bodies
- Consistency constraints for architecture diagrams
  a) A component (module, subsystem) is a “contained” in at most one component
  b) In local usability, an edge from a grand parent to grand child node is not allowed.
  c) Cyclic local usability is allowed, but no ado in this cycle
  d) General usability is acyclic

    e) General component cannot be a local component at the same time
    f) …

- MIL level: Export consistency
    a) A f module is built up only from proc or func specifications
    b) an ado only from access operations
    c) an adt has an opaque type and access operations or a creation / deletion together with access operations
    d) if the ado is to be used more than once, there has to be an initialization operation.
    e) …
- MIL level: export-import consistency
    a) Import cannot be more comprehensive than export. It is allowed, however, to import only a nonempty part of B's interface.(所有 B 的出口，必须存在人用，但是 A 可以只用一部分)
    b) …
- Programming language level: interface-body consistency
    a) Every imported resource is used
    b) …

Ch05

- Evident method rules for architecture
    a) A contains relation has a parallel local usability(就是实线单箭头和点虚线永远是同时存在的)
    b) General component often has more than one targeting general usability
    c) Every component should have one design decision.
    d) …
- Different forms of data exchange, f modules M1, M2
    a) One write one read on the ado: entry
    b) Write and read, on a collection: collection
    c) All writes, then all reads, loose coupling: file
- I/O are data abstraction application



I/O are containers from / to which we get / put data or activity to organize / carry out I or O processes

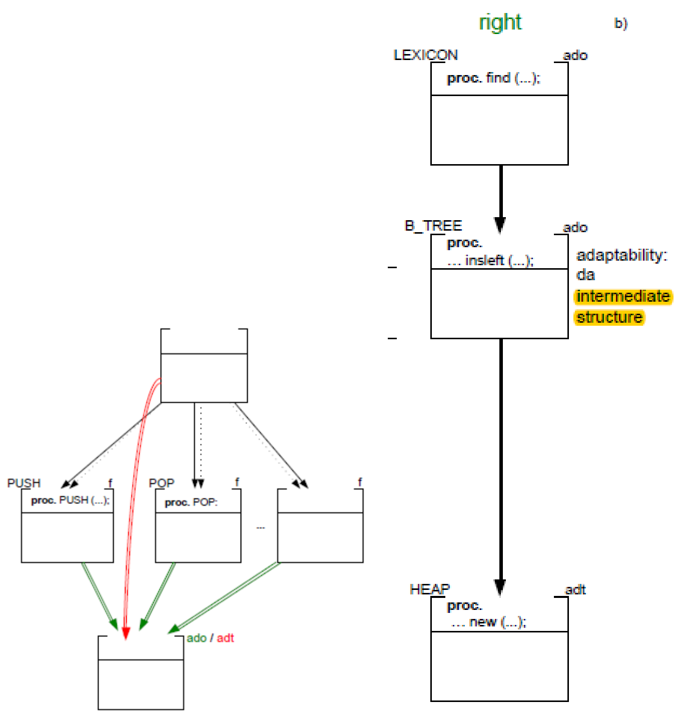
- How to choose type of a module:
  state / memory → ado
  template for states → adt

transformation, computation, I/O behaviour → f

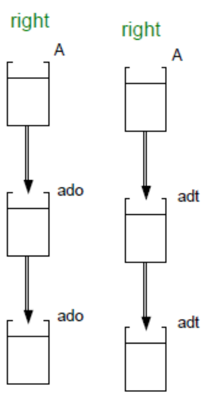
- 1, f modules action-oriented: does not mean that they get active from themselves, are invoked from above

  2, ado / adt passive: may invoke other modules below, even functional ones

  3, iterators in the interface of a da module: no clear distinction between passive (da module) and active (iterator)
- Possible:



  注意到 push, pop 其实都是操作，而且不可能是作用于全局的，所以要有对应的 ado 才对
- Always cut horizontally, not vertically

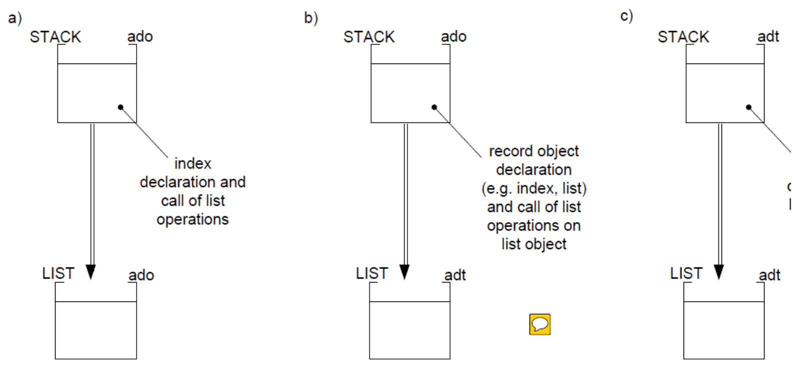


- Onion model: above is functional modules and local usability, below is da modules and general usability

  Top-down may not recognize common modules, so use bottom-up
- Adt-ado, 2 level:

  Not allowed both are collection or entries

  Allowed one entry and another collection
- 总共只有下面三种情况，就是没有 adt=>ado 的


a)
STACK
ado
index declaration and call of list operations
LIST
ado
b)
STACK
ado
record object declaration (e.g. index, list) and call of list operations on list object
LIST
adt
c)
STACK
adt
LIST
adt

- Where is object of ado (用的是 variable semantics，即 adt 自己不管，在 client)
    - a) realization complete in the body of the ado STACK
      memory data container, bookkeeping info, altogether there
      using programming language constructs (array, index)
    - b) ado STACK is based on an ado LIST:
      bookkeeping in STACK, data container in LIST
    - c) ado STACK is based on an adt LIST:
      bookkeeping in STACK, LIST object in STACK
    - d) adt STACK
      memory (container + bookkeeping info) in the client module which uses STACK (more precisely next slide)
- where is object of adt
  Variable semantics adt:
    1. objects are created via <u>declaration in the body of client module</u>
    2. runtime stack of the underlying PL implementation administrates objects (creation, deletion)

  Reference semantics adt:
    1. objects are created via <u>creation operation and deleted via deletion operation</u>
    2. below the adt there is a subarchitecture, eventually heap of the underlying programming language
- 3 ways to do multiple data abstractions(entry and collection)
    1. Simple solution, Entries implicit: only provide collection operations, client can only use one entry at the same time and can only use as part of the collection
    2. Dirty solution, open entries: (就是写成 record… 的长列,都可访问操作)no data abstraction, entry data seen by client, module has mixed characters of adt and ado
    3. Clean solution, subsystem
- Variable or reference semantics for entry E is important on architecture and on realization
- Difference of variable and reference semantics
  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

- Hiding subsystem’s internals ≠ locality

  Subsystems: hiding by scope/visibility outside/inside; interfaces are composed

  Locality: hiding by scope/visibility; modules can be seen but no use of inner modules
- Kind of a subsystem

  Module: determined

  Subsystem: often determined(就看占主导的那个 module 的类型，e.g. f subsystem)

  Complete system: mixed(经常是一部分 ado，一部分 f)
- Difference access has different complexity
  1. determined access (e.g. FiFo, LiFo)
  2. context-based via determined components (e.g. name)
  3. full associative access (via arbitrary component values, delivering a set of entries)
- OO summary

  Pro:

  its inheritance hierarchies = classification + similarities

  Cons:

  Classification not easy to build, extend, or restructure, sometimes multicriterial classification needed, not expressible
- 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
- Problems of OO
  1. No distinction of components according to purpose
  2. Component has 1 design decision- often violated
  3. Objects only on “heap”, which is a big abstraction from reality
  4. No strict information hiding(可以看见接口下面的)
  5. Genericity not available
  6. Reorganization difficult
  7. Class hierarchy does not contain much information
  8. …

Ch06

- High-level languages: ada,java

  Low-level : FORTRAN, C, Assembler
- Not possible to recursion on FORTRAN

  Not possible to OO on C
- 3 classes of programming languages:

   1. languages without module construct: no hierarchies, arbitrary relations, no checking consistency: FORTRAN, C, Cobol, Basic, Assembler
   2. block-structured languages without module construct: locality by nesting, visibility / scope, no independent compilation, dynamic storage allocation (stack, heap): Standard-Pascal, Algol
   3. languages with module concepts: independent compilation, more or less checking between units, multi-paradigmatic: Ada, Java, C++

- translating to FORTRAN
  可以写 body 中的函数，但是头部 interface 都是只能注释
  FORTRAN programs are unstructured "heap" of units
  So no module relation contains, local usability, general usability, just simulation by layout, comments, and discipline
  subsystems by two-fold simulation
  genericity only by macro mechanisms
- translation to C
  1 module to 2 files
  Interface: header file
  Body: implementation file
  Simulation similar to FORTRAN
   1. subsystems via double simulation
   2. genericity only by macro
   3. OO not available

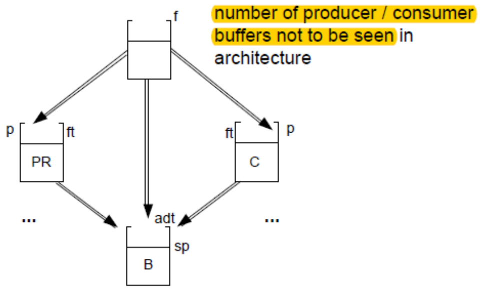
  Simulation of FORTRAN and C meet consistency constraints by disciplines and bodies are hidden
- Translation to Pascal
  Similar: aggregate mentally, compose textually, layout, and use comments, apply discipline
  Translation is not simpler than the former 2
  Has no nesting for modules
  Has no import relations
  "modules" connected by local and general usability is in the upmost block
- Translation to Ada
  Language with modules, imports, multiparadigmatic language
  Contains relation by nesting
  Export interface, is-contained clause, local import are as comments
  general components: library units
  general usability: Ada import clauses
  architectural language is conceptually much richer than programming languages
  Ada has generic packages

Ch07

- 本章主要是 extension on the architecture language
- Specifying component interaction by traces(add on the architecture diagram)
  Or use sequence diagram(就是几个竖线表示 users，然后横向时间顺序画箭头写

message)

- Concurrent systems using
  F module producer, f module consumer, ado module buffer
  Producer and consumer has own thread, need synchronization
- More than one producer/consumer, and unknown at development time, then use ft module.这样就不用画图画很多个 f 了，只画一个 ft



solution with producer / consumer types

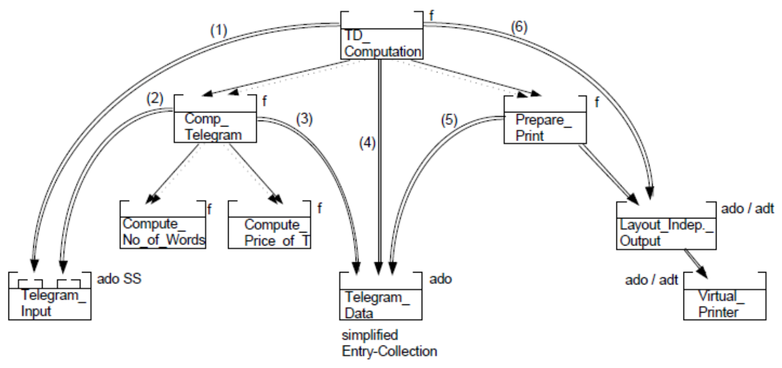
- Add component for start/stop the system: control functionality,一个 stopcomponent，连带着和他相关的 component 也都有了 stop 的能力，比如 by call or sign
- Below are concrete and abstract component connections:

1. After extension, there are different forms of usability.
   a. So the contains edge can mean getting signal, get trigger, entry call for interrupt or ready or completion or activation. Need to add annotations at the usability edges
2. Use registering and callback, delegate activation to component which knows about state change, instead of checking myself over and over
   a. Usage: GUI user input, active databases
3. Indirect use: Client calls superclass, superclass calls subclasses via dispatching
4. Register and broadcasting, this kind of dynamic use should be planned at design time

- 3 specific application-oriented styles
  Pipelines
  Data flow architectures
  Loosely coupled systems

Ch08

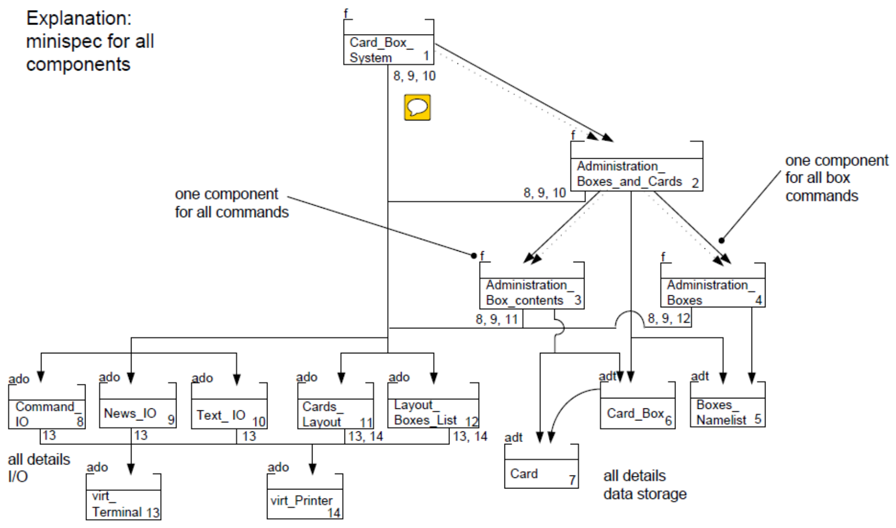
- F 到 f 就是都有虚实相伴的 contains，但是 f 到 ado 就是单独实线单箭头。
- New solutions:

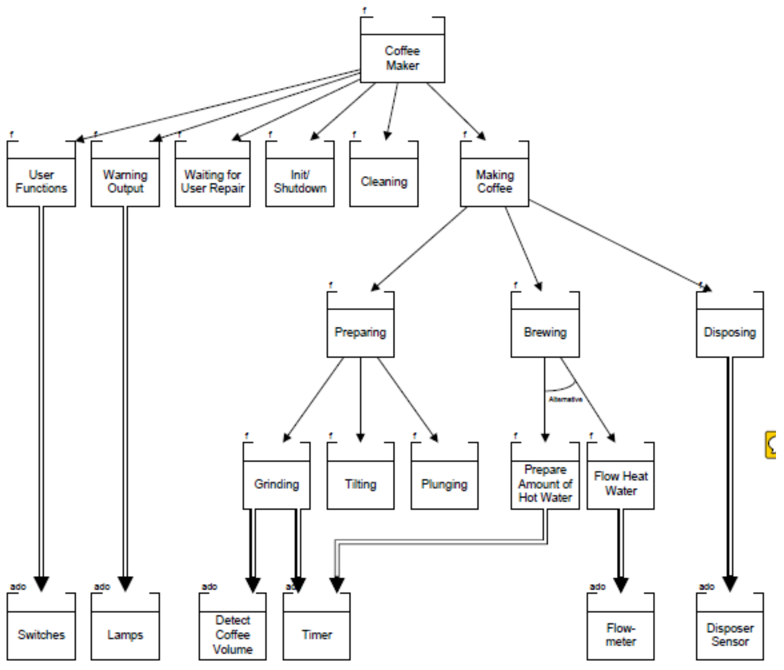


(1) Open / Close / File empty
(2) Read Telegram Data Components
(3) Store Triple
(4) Open / Close / Sort
(5) Reset / Read Current Triple
(6) Open / Close

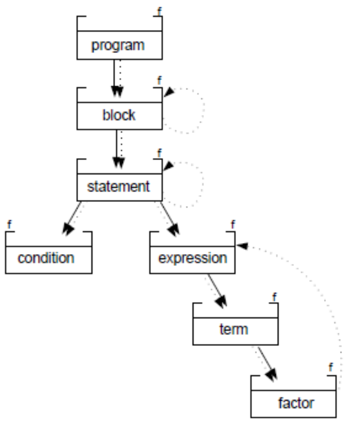
Discuss:
- Realization Change
- Functionality Change
- Estimate Changes

Explanation:
minispec for all
components

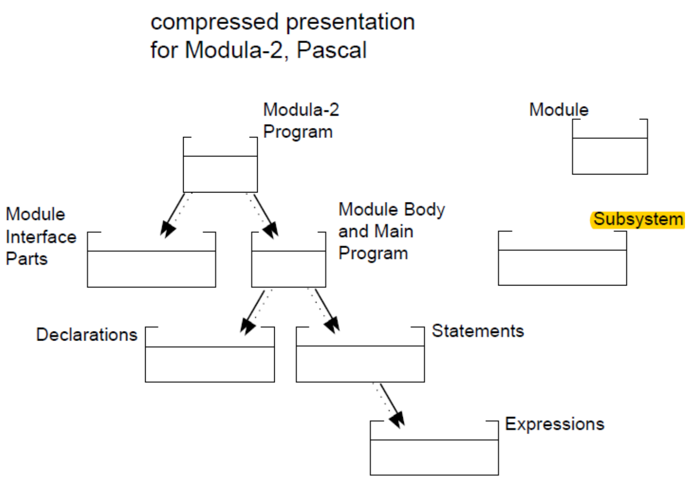


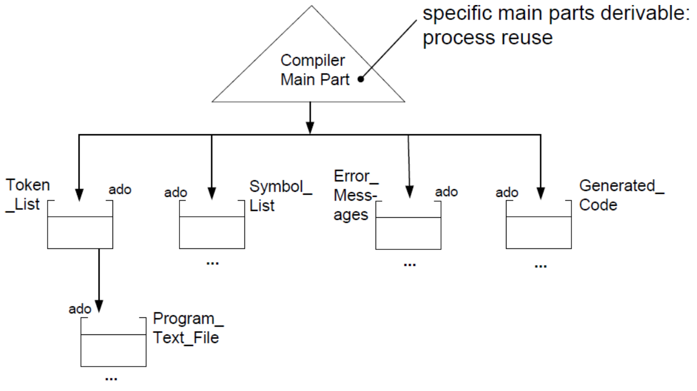


- Compiler: recursive descent   one, with all in one procedure



压缩后：

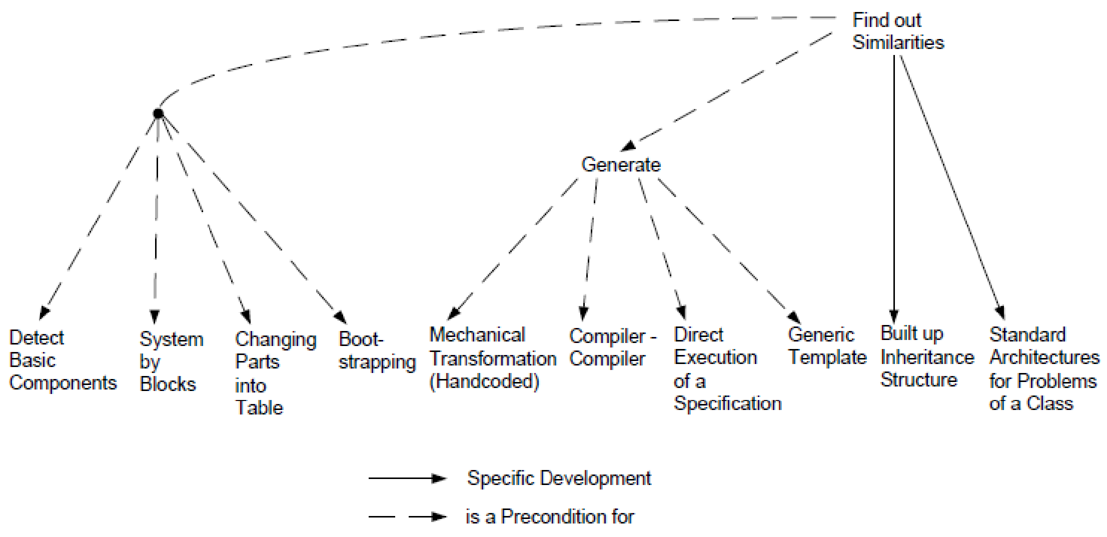


- 

Ch09

- Transformation requirement specification to architecture
  Functional requirement specification:system processes, application data→upper part: like main functions, business data entities and controlling UI
  Nonfunctional requirement specification: efficiency params, storage, runtime, hardware →further layers and components: details of storage, communication structure, basic components
- Nonfunctional requirement specification are related to the complete architecture

- Architecture-4 consideration
    1. Maintainability: effort for changes, detect where changes are forethought
    2. Reuse: general components, generic components, basic layers
    3. Understandability: functional decomposition, design decisions
    4. Quality assurance: check against requirement, robustness, user friendliness, portability, adaptability
- Strategies for adaptable architectures



Find out Similarities
Generate
Detect Basic Components
System by Blocks
Changing Parts into Table
Boot-strapping
Mechanical Transformation (Handcoded)
Compiler - Compiler
Direct Execution of a Specification
Generic Template
Built up Inheritance Structure
Standard Architectures for Problems of a Class
Specific Development
is a Precondition for

Exercises

- Parametric polymorphism (...), allows a single piece of code to be typed “generically,” using variables in place of actual types, and then instantiated with particular types as needed
  Ad-hoc polymorphism, by contrast, allows a polymorphic value to exhibit different behaviors when “viewed” at different types.
  https://stackoverflow.com/questions/6730126/parametric-polymorphism-vs-ad-hoc-polymorphism
- Validation vs verification:
  software verification ensures that the output of each phase of the software development process effectively carry out what its corresponding input artifact specifies(built it right)
  software validation ensures that the software product meets the needs of all the stakeholders(built the right thing)
  https://en.wikipedia.org/wiki/Software_verification_and_validation#Validation_vs._verification
- What abstract in adt and ado means:
  a) interface more abstract than realization
  b) abstraction from a variety of realizations
- In OO:
  Object is ado
  Class(which can be derived to other classes, e.g. stack) is adt
- Adt in functional programming:

```
module Stack (Stack, empty, isEmpty, push, top, pop) where

empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a
top :: Stack a -> a
pop :: Stack a -> (a,Stack a)

newtype Stack a = StackImpl [a] -- opaque! Just a comment
empty = StackImpl []
isEmpty (StackImpl s) = null s
push x (StackImpl s) = StackImpl (x:s)
top (StackImpl s) = head s
pop (StackImpl (s:ss)) = (s,StackImpl ss)
```

https://wiki.haskell.org/Abstract_data_type

- Haskell, function, take a list and return a list

```
plus :: [Int] ->[Int]
plus(x:xs) = 2*x+1 : plus xs
plus[ ] = [ ]
```

- Use map, and shorten code

```
map :: ( Int -> Int ) -> [Int] -> [Int]
map funtion (x:xs) = funtion x : map funtion xs
map funtion []= []

multiply2andPlus1 :: Int -> Int
multiply2andPlus1 x = 2*x+1

multiply2 :: Int -> Int
multiply2 x = 2*x

plus1 :: Int -> Int
plus1 x = x+1
```

- Difference, OO classes vs Haskell type classes

  Type classes are like interfaces/abstract classes, not classes itself

  There is no inheritance and data fields (so type classes are more like interfaces than classes)....

  Just defines a type class by specifying a set of function or constant names, together with their respective types, that must exist for every type that belongs to the class. So no implementation

  https://wiki.haskell.org/OOP_vs_type_classes#Type_classes_vs_classes

- Functional vs imperative

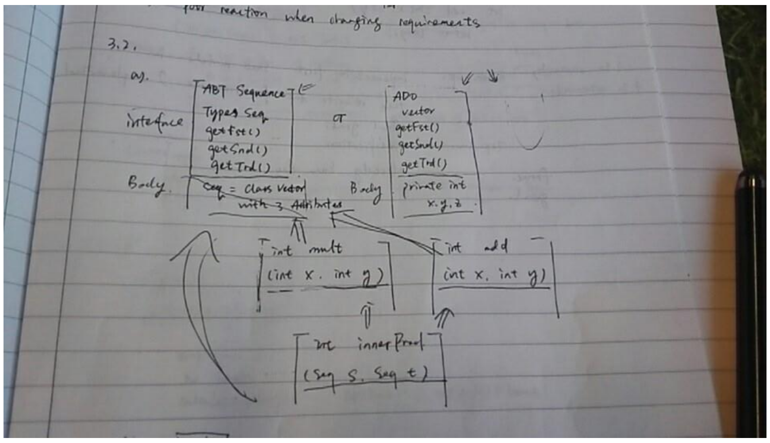
  Imperative: Programs as statements that directly change computed state

  Functional: Treats computation as the evaluation of mathematical functions avoiding state

and mutable data, rely heavily on recursion
e.g. gcd

```
selfGCD :: Integral f => f -> f -> f
selfGCD a b
  | b == 0              = a
  | otherwise           = selfGCD b (mod a b)
```

- Characteristics of module
  1. is logical unit
  2. has tight internal coupling and loose coupling to other modules
  3. is often capable of reuse
  4. offers export interfaces to clients, while body details are hided
  5. represents design decisions; architecture is sum of all decisions
  6. is unit consisting of data and operations
  7. can import from other modules for realization
- F module vs da module
  Functional modules :
    1. trandform input to output data
    2. “activityoriented”
    3. I/O dta apper in the export interface
    4. have more than one operation in the interface
    5. no state(memoryless)
    6. formalization with pre and post condition for every operatio

  Data abstraction modules:
    1. see only operations on data and not the detailed representation of data or
    2. operation realization.
    3. have a state(memory), which is defined by the current values of the variables.
    4. “passive”
    5. formalization with logical pre and post conditions or algebraic equations.
- Waterfall model:
  The waterfall model is a sequential (non-iterative) design process, in which progress is seen as flowing steadily downwards through the phases of conception, initiation, analysis , design , construction, testing , production/implementation and maintenance
- 

- Formal specification description, use math

- Bottom-up vs top-down
  Bottom up : do the first time, make mult, add with identity , and compose innerproduct, then really implement the small functions, organizing parts of the solution space into larger chunks
  Top down: know very well, just go and implement mult, add, and so on. decomposition of the problem space into sub-problems
- Adt:
  variable semantics: type ITEM_STACK_TYPE is private
  reference semantics: STACK_DENOTOR_TYPE
  且之后所有函数过程也都是这种区别
- Semantics, signature, type of data 助教的理解是这样的
  F: int -> int signature of f
  X|-> x+ 1 if implementation of f

  Types of data:
  X: in Seq
  Y: out int

  Generic datatype:
  Type Seq
  getFirst: Seq -> item
- Subsystem 特点:
  (1) It is determined which (resources of which) modules build up the interface of the subsystem.
  (2) The internal subarchitecture is hidden.
  (3) Different from modules, subsystem have aggregated interfaces.
  (4) The realization includes design and later implementation.
  (5) In subsystem the hierarchical design and modules flat design will be introduced
  (6) Strongly coupled in the system, loose to the others
- Genericity:
  Generic unit is a template, and is only designed and implemented once
  Generic unit→ specific unit→ bind into system
  Generic unit is no architectural unit(就是 template,并不实际在架构里面的实例 unit)
  By instantiation of it we get a unit
  Genericity corresponds to design process rather than design results
- Generic type in Haskell

  ```
  data Month = January | February | March | April | May | June | July
               | August | September | October | November | December
               deriving (Enum, Eq, Ord, Show)

  mymax :: Ord a => a -> a -> a
  mymax x y = if x > y then x else y
  ```

  助教的例子

Datatype list a = Nil|Cons a (list a)

Length:: list a - > Int

Length[] = 0

Length (x:xs) = 1+ length xs

- Architecture characteristics
  a) Has to map requirements
  b) Determines long-term quality of system
  c) One architecture can answer different requirements
  d) Organization of development
  e) Organization of change
  f) Design errors have huge influence
  g) Built plan as result of design
  h) Complete plan(components, and relations)
  i) Consistent to requirement
  j) Independent from programming languages
  k) A result of an intellectual process
- Batch: telegram
  Interactive: card-box
  Embedded: coffee maker
- Code generation
  Pros:
  1. increase quality and reduce hand written
  2. faster
  3. abstraction for the users

  e.g.
  java automatically generate getters and setters
- Automata describes the transition between states according to the inputs, while sequence diagram describes the message exchange between different objects
- Class diagram, structural; sequence and automata, behaviors