CIS 422
Software Methodologies I

Chapter 12
Software Design

Professor: Juan J. Flores
jflore10@uoregon.edu

UNIVERSITY OF OREGON

Software Design

Main issues:

- decompose system into parts
- many attempts to measure the results
- design as product ≠ design as process

Overview

- Introduction
- Design principles
- Design methods
- Conclusion

Programmer's Approach to SE

Skip requirements engineering and design phases; start writing code

Programmer's Approach to SE

Why do programmers take this approach?

- Design is a waste of time
- We need to show something to the customer real quick
- We are judged by the amount of LOC/month
- We expect or know that the schedule is too tight

Programmer's Approach to SE

However

The longer you postpone coding, the sooner you'll be finished

Up front remarks

- Design is a trial-and-error process
- The process is not the same as the outcome of that process
- There is an interaction between requirements engineering, architecting, and design

Software Design – a Wicked Problem

- There is no definite formulation
- There is no stopping rule
- Solutions are not simply true or false
- Every wicked problem is a symptom of another problem

Overview

- Introduction
- Design principles
- Design methods
- Conclusion

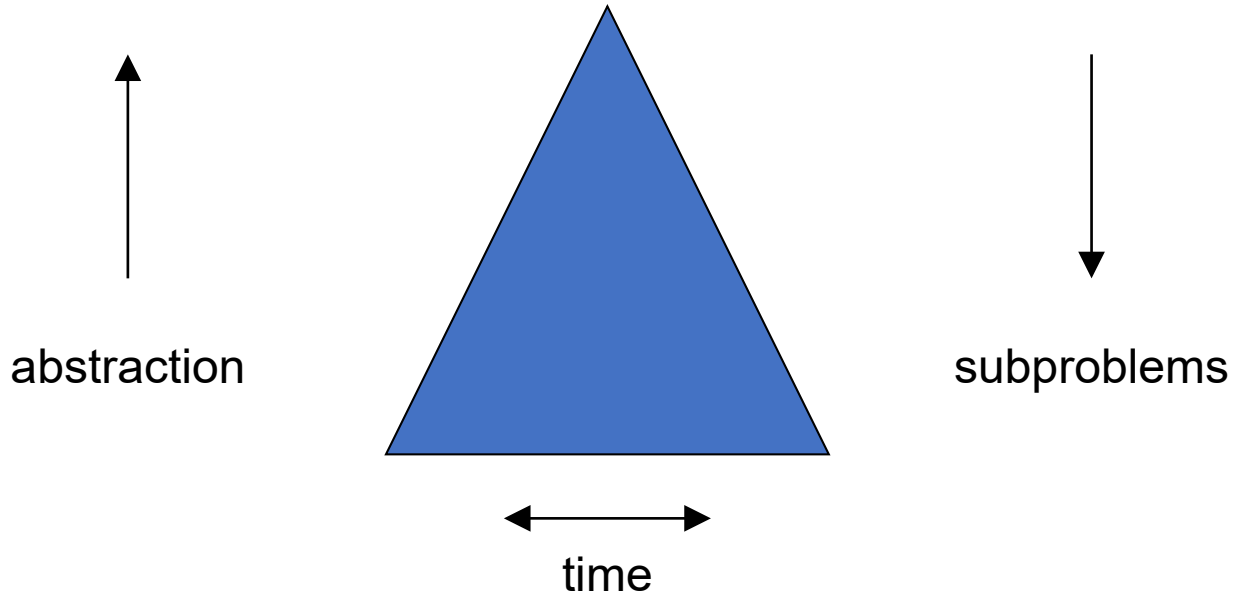
Design principles

- Abstraction
- Modularity, coupling, and cohesion
- Information hiding
- Limit complexity
- Hierarchical structure

Abstraction

Procedural abstraction:

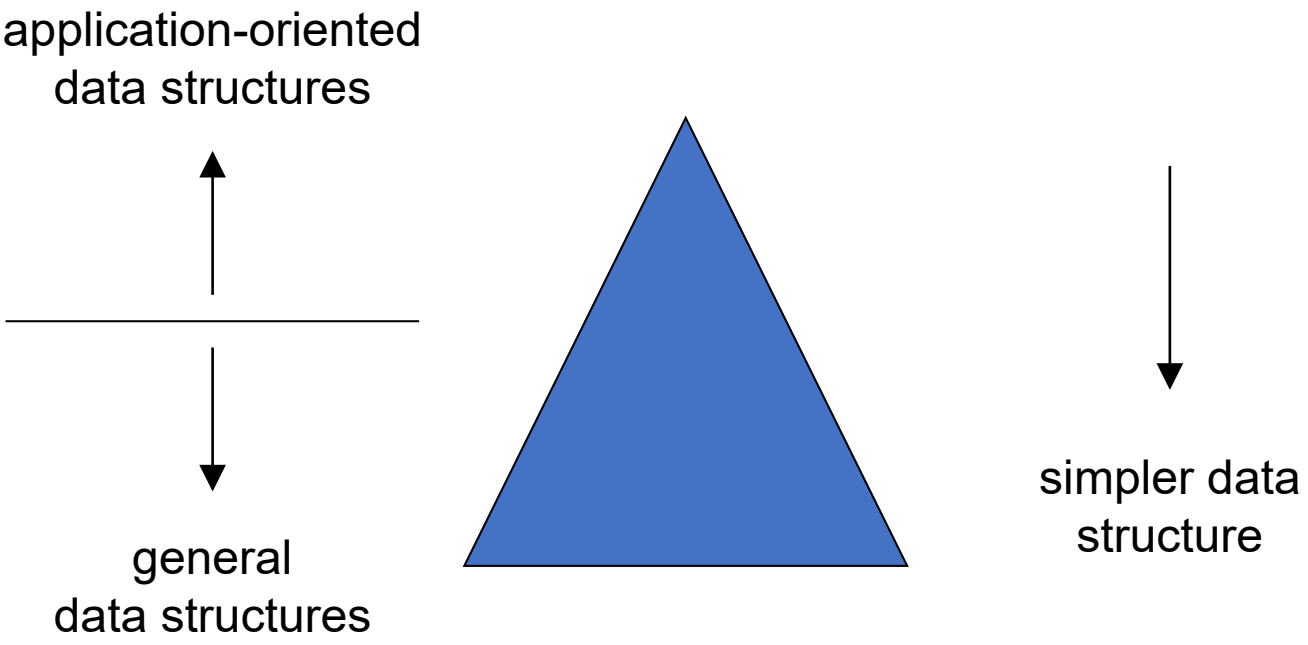
- natural consequence of stepwise refinement
- Name of procedure denotes sequence of actions



Abstraction

Data abstraction:

- aims at finding a hierarchy in the data

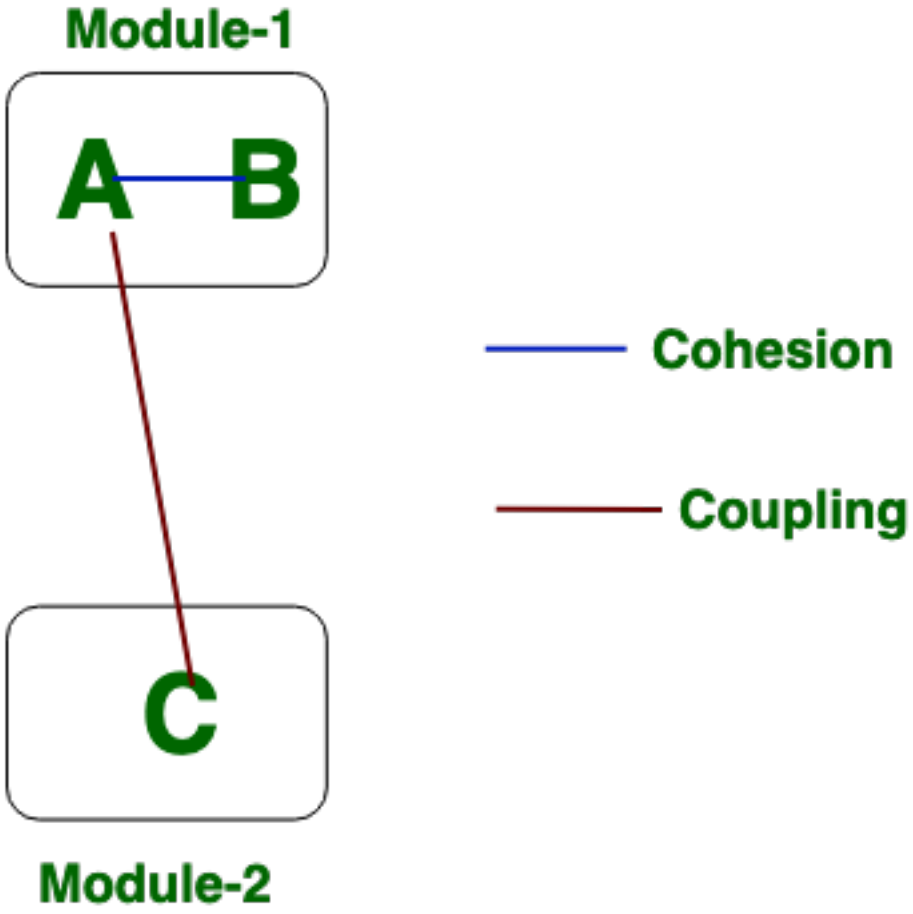


Modularity

- Structural Information – individual modules and their interconnections

- Cohesion and coupling
  - cohesion: how related the functions within a single module are
  - coupling: interdependencies between modules

Modularity



Cohesion Types

- Coincidental – no sense of organization
- Logical – component contains related functions
- Temporal – components activated at about the same time
- Procedural – elements executed in a certain sequence
- Communicational – elements operate on the same data
- Sequential – the output of one element is the input of the next one
- Functional – elements contribute to a single well-defined task

- Data – to cater for abstract data types or OOP

How to determine the cohesion type?

Describe the purpose of the module in one sentence

- Logical or communicational cohesion
  - if the sentence is compound, contains a comma or more than one verb
  - it probably has more than one function
- Temporal cohesion
  - if the sentence contains time-related words like “first”, “then”, “after”
- Logical cohesion
  - if the verb is not followed by a specific object
  - probably
- Temporal cohesion
  - Words like “startup” or “initialize”

Types of coupling

- Content - one module can modify the data of another module; control flow is passed from one module to the other module (avoid it)
- Common – modules have shared data (no reuse because of context sensitive behavior)
- External – components communicate through an external medium, such as a file
- Control – modules communicate by passing control information (flags)
- Stamp - complete data structures are passed
- Data – communicating through data

Coupling levels are technology dependent

Old programming languages

- Data coupling assumes scalars or arrays
- Stamp coupling passed records
- Control coupling assumes passing of scalar data

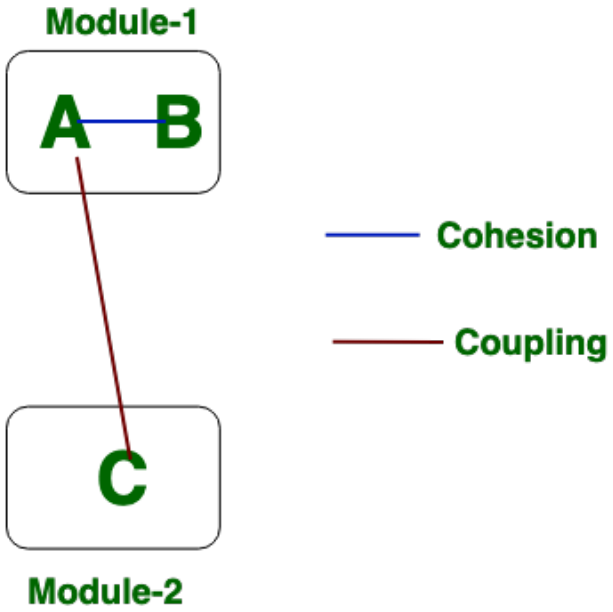
Nowadays

- Modules may pass complex data structures
- Modules may allow some modules access to their data, and deny it to others (so there are many levels of visibility)
- Coupling need not be commutative
  - A may be data coupled to B, while B is control coupled to A

Strong Cohesion & Weak Coupling

Simple interfaces

- Simpler communication
- Simpler correctness proofs
- Changes influence other modules less often
- Reusability increases
- Comprehensibility improves
- Less error prone



Information hiding

- Each module has a secret
- Design involves a series of decisions
  - who needs to know
  - who can be kept in the dark
- Information hiding is strongly related to
  - abstraction: if you hide something, the user may abstract from that fact
  - coupling: the secret decreases coupling between a module and its environment
  - cohesion: the secret is what binds the parts of the module together

Information hiding

Each module has a secret

- Module A implements heaps
- Module B uses heaps
- Module A may use arrays, binary, or binomial heaps
- Module B does not have to know

Point to ponder

- How many lines of code does this program contain?

```
#include <stdio.h>
#define NULL 0
main ()
{
  int i;
  for (i=0;i<10;i++) printf(%d”,i);
}
```

Complexity

- Measure certain aspects of software
  - lines of code
  - # of if-statements
  - Nesting depth
- Use those measures to assess or guide the design
- Interpretation
  - higher value
  - $\Rightarrow$ higher complexity
  - $\Rightarrow$ more effort required
  - = worse design
- Two kinds
  - **intra-modular:** inside one module
  - **inter-modular:** between modules

intra-modular

- Attributes of a single module

- Two classes:
  - measures based on size
  - measures based on structure

Sized-based complexity measures

- Counting lines of code
  - differences in verbosity
  - differences between programming languages
  - `a:= b` versus `while p^ <> nil do p:= p^`
- Halstead's "software science"
- Essentially counting operators and operands

Software science basic entities

- n_1: number of unique operators
- n_2: number of unique operands
- N_1: total number of operators
- N_2: total number of operands

Example program

```
public static void sort(int x []) {
  for (int i=0; i < x.length-1; i++) {
    for (int j=i+1; j < x.length; j++) {
      if (x[i] > x[j]) {
        int save=x[i];
        x[i]=x[j]; x[j]=save
      }
    }
  }
}
```

operator, 1 occurrence

operator, 2 occurrences

operator	# of Occurrences
`public`	1
`sort()`	1
`int`	4
`[]`	7
`{}`	4
`for {;;}`	2
`if ()`	1
`=`	5
`<`	2
…	…
$n_1 = 17$	$N_1 = 39$

Example program

```
public static void sort(int x []) {
  for (int i=0; i < x.length-1; i++) {
     for (int j=i+1; j < x.length; j++) {
         if (x[i] > x[j]) {
             int save=x[i];
             x[i]=x[j]; x[j]=save
         }
     }
  }
}
```

operand, 2 occurrences

operand, 2 occurrences

operand	# of occurrences
x	9
length	2
i	7
j	6
save	2
0	1
1	2
$n_2 = 7$	$N_2 = 29$

Other software science formulas

- Size of vocabulary: $n = n_1 + n_2$
- Program length: $N = N_1 + N_2$
- Volume: $V = N \log_2 n$
- Level of abstraction: $L = V^* / V$
- Approximation: $L' = (2/n_1)(n_2/N_2)$
- Programming effort: $E = V/L$
- Estimated programming time: $T' = E/18$
- Estimate of N: $N' = n_1 \log_2 n_2 : n_2 \log_2 n_2$
- For this example: $N = 68$, $N' = 89$, $L = .015$, $L' = .028$

Software science

- Empirical studies
  - reasonably good fit
- Critique
  - explanations are not convincing
  - results from cognitive psychology used wrong
  - is aimed at coding phase only (assumes this is an uninterrupted concentrated activity)
  - different definitions of “operand” and “operator”

Remember

- Project 1
  - Progress meetings on Tuesday Feb. 2nd and Thursday Feb. 4th
  - Project presentations on Friday Feb. 5th
  - Due on Wednesday Feb. 10th

- Midterm
  - Includes chapters 1, 2, 3, 5, 11, and 12
  - Enabled Friday to Sunday (Feb. 5 – 7)
  - Practice quiz already available on Canvas

Questions

- What is a module?
- What is cohesion?
- What is coupling?

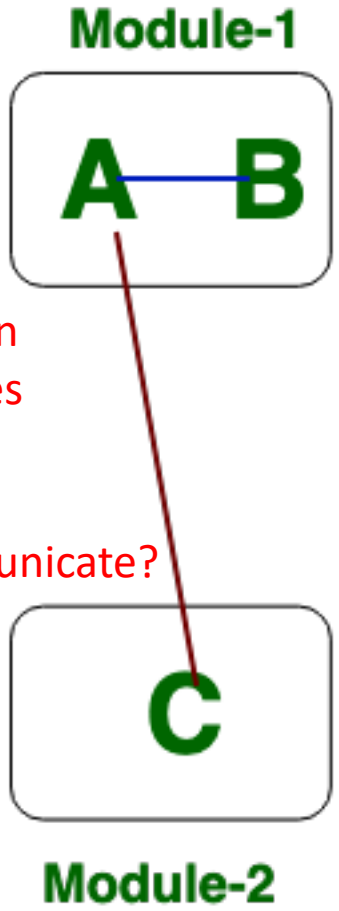
Modules

What is a module?

- A part of a program.
  - Programs are composed of one or more independently developed **modules**.
  - A single **module** can contain one or several routines.

- A module is a file containing Python definitions and statements
  - Python docs

Modules



Are the programs inside a module coherent?

How are programs in different modules related?

How do they communicate?

Cohesion

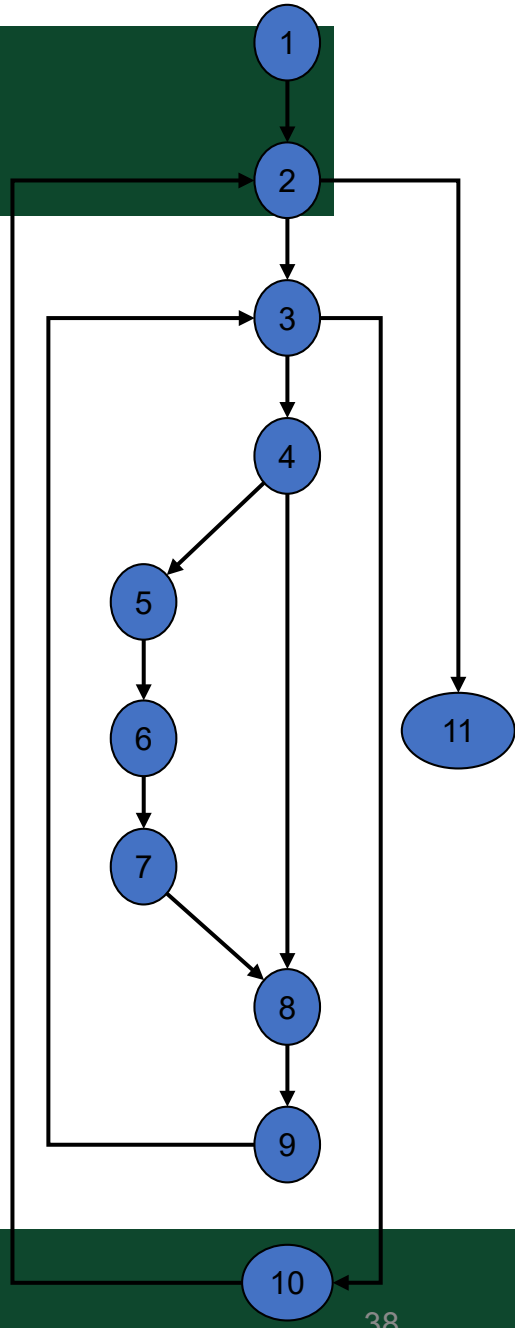
Coupling

Structure-based measures

- Based on
  - control structures
  - data structures
  - or both
- Example complexity measure based on data structures: average number of instructions between successive references to a variable
- Best known measure is based on the control structure: McCabe's cyclomatic complexity

Example program

```
public static void sort(int x []) {
  for (int i=0; i < x.length-1; i++) {
      for (int j=i+1; j < x.length; j++) {
          if (x[i] > x[j]) {
              int save=x[i];
              x[i]=x[j]; x[j]=save
          }
      }
  }
}
```



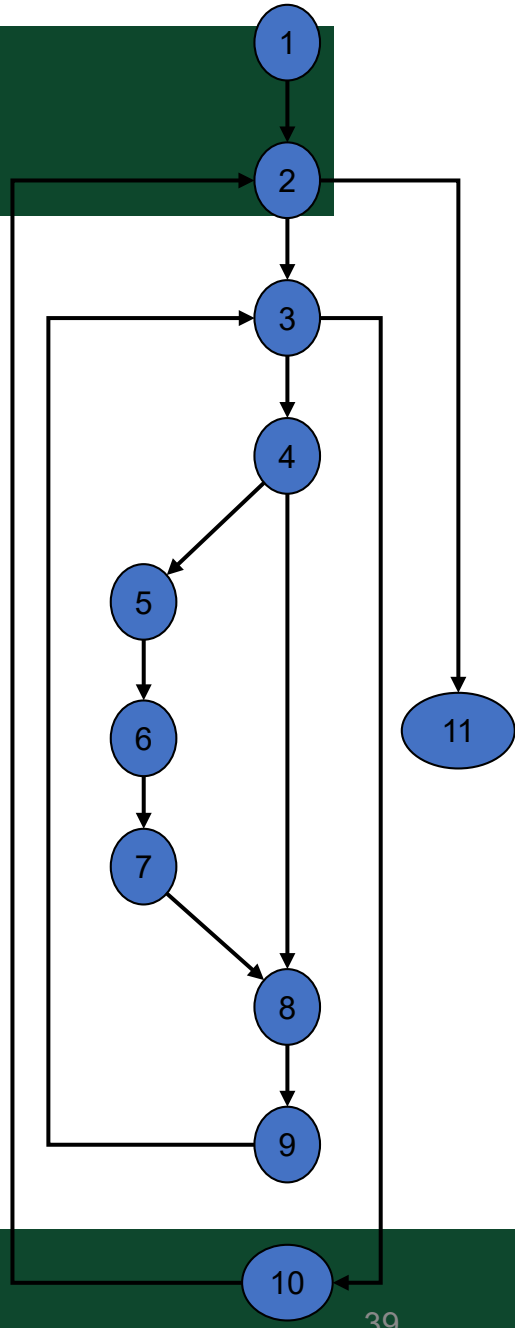
Cyclomatic complexity

e = number of edges (13)

n = number of nodes (11)

p = number of connected components (1)

CV = e - n + p + 1 (4)



Intra-modular complexity measures, summary

- Small programs -> measures correlate well with programming time
- LOC does equally well
- Complexity measures are not very context sensitive
- Complexity measures take into account few aspects

- Complexity *density* instead CV/LOC

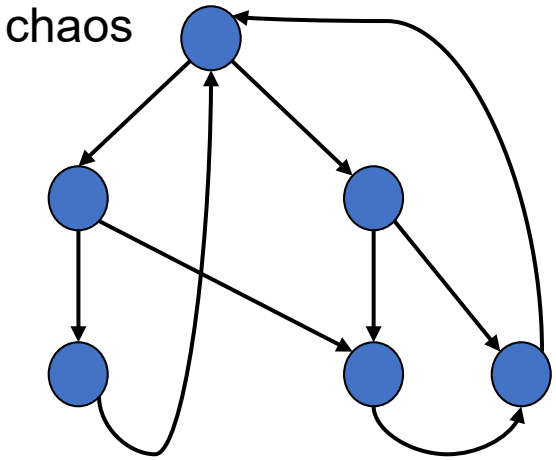
System structure: inter-module complexity

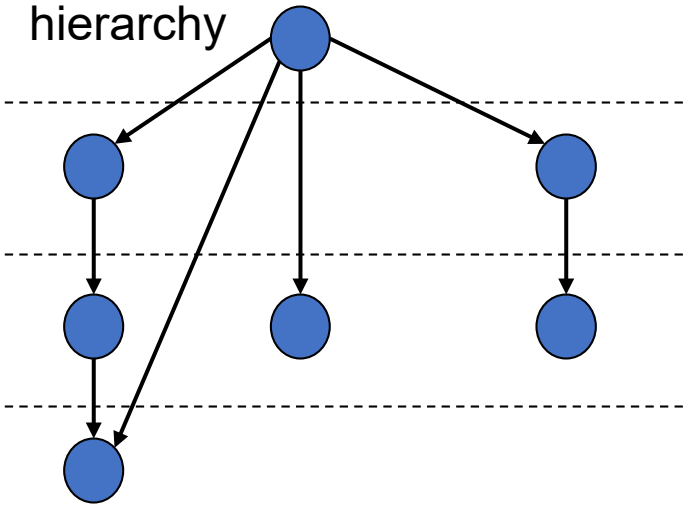
- Complexity of dependencies *between* modules
- Draw modules and their dependencies in a graph
- Edges connecting modules may denote:
  - A contains B
  - A precedes B
  - A uses B
- We are mostly interested in the latter type of relation

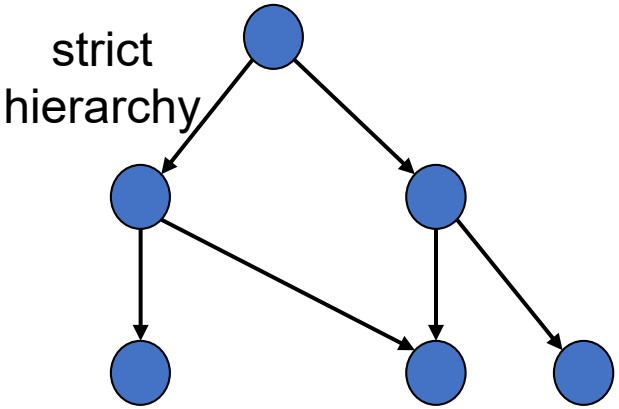
The *uses* relation

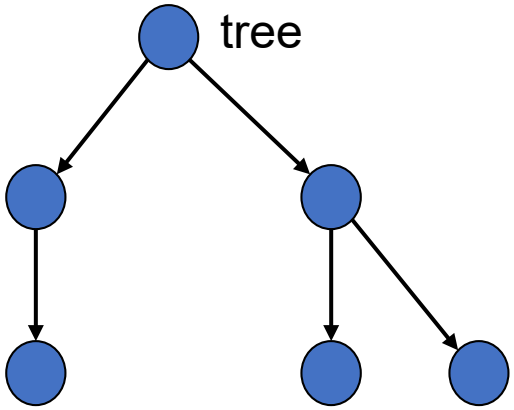
- In well-structured software dependencies appear as procedure calls
- *call-graph*
- Possible shapes of this graph:
  - chaos (directed graph)
  - hierarchy (acyclic graph)
  - strict hierarchy (layers)
  - tree

The *uses* relation

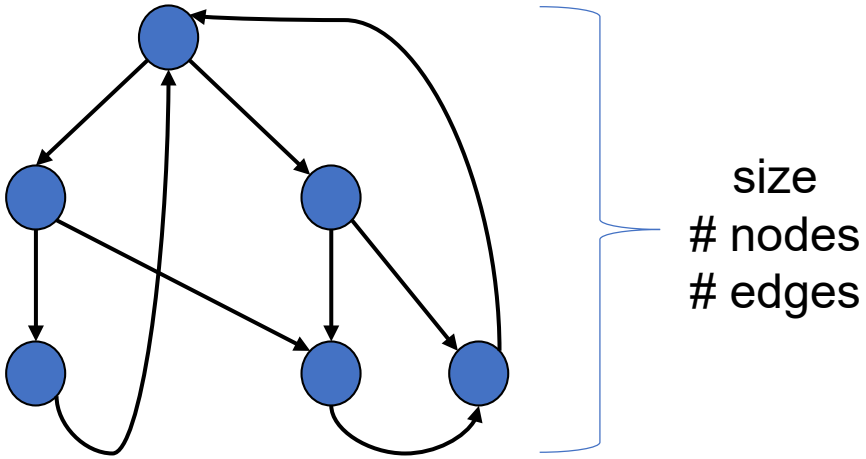


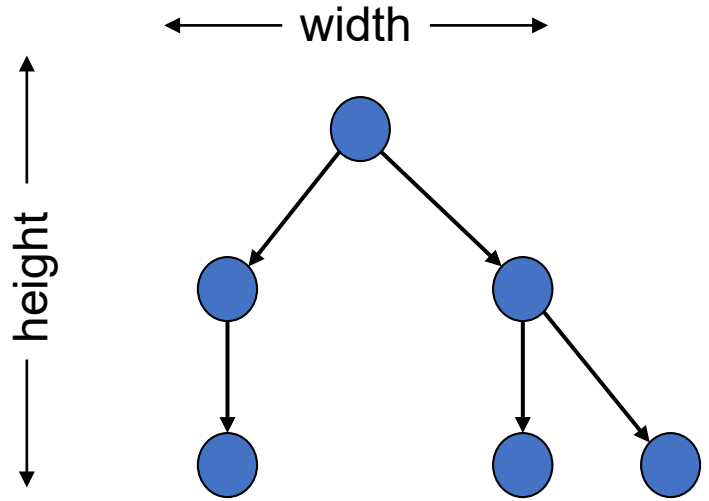




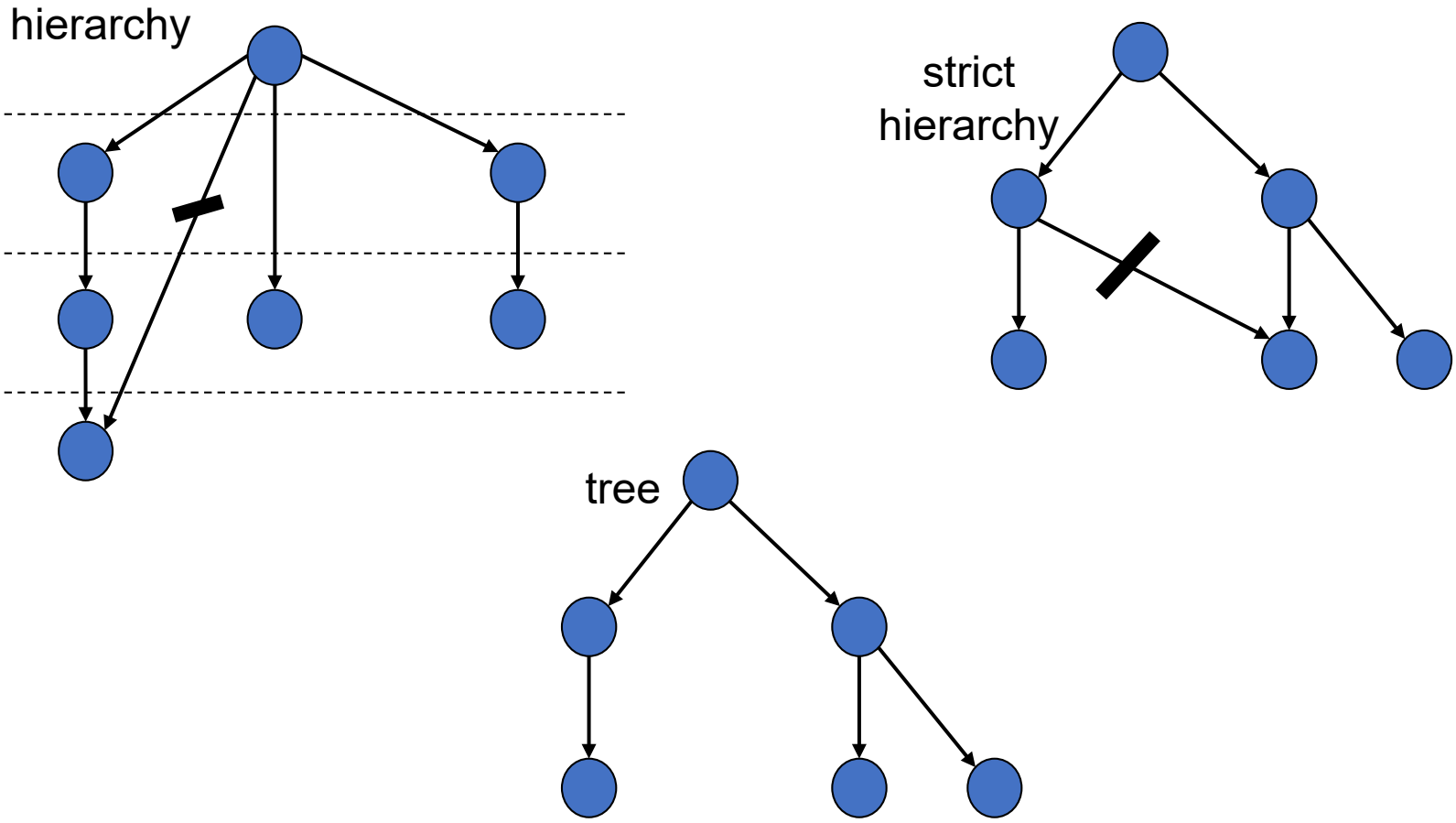


Measurements



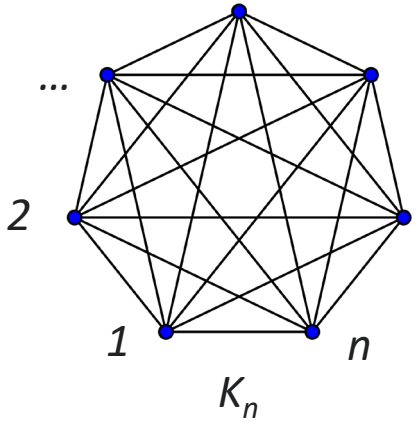


Deviation from a tree



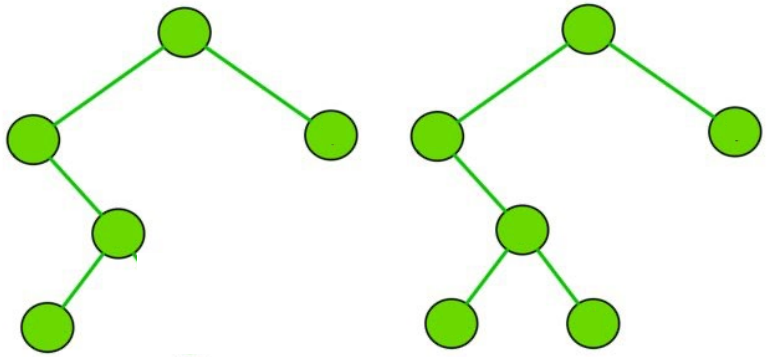
Tree impurity metric

- How many edges in a complete graph?
- First node connects to *n-1* other nodes
- There are n nodes, so total is *n(n-1)*, right?
- Wrong, you are counting every edge twice
- Edges(K_n)=*n(n-1)/2*



Tree impurity metric

- How many edges in a tree?



- Tree T_n has n vertices
- edges(T_n) = $n-1$

- Root has been added, edges(T_1) = $n-1 = 0$
- Induction: Assume the property holds for n, and prove for n+1
- Adding one node adds one edge
- edges(T_n)+1 = $n-1+1$ = n = edges(T_{n+1})

Tree impurity metric

- Complete graph with n nodes has n(n-1)/2 edges
- Tree with n nodes has (n-1) edges

- Extra edges = e – (n-1)
- Max. extra edges = n(n-1)/2 – (n-1)
  = (n(n-1) – 2(n-1))/2
  = ((n-1)(n-2))/2

- Tree impurity = number of extra edges / max. no. extra edges

$$m(G) = 2(e-n+1)/(n-1)(n-2)$$

Desirable properties of a tree impurity metric

m(G) is a "good" measure, in the measurement theory sense

- m(G) = 0 if and only if G is a tree
- $m(G_1) > m(G_2)$ if $G_1 = G_2$ + an extra edge
- if G_1 and G_2 have the same # of "extra" edges wrt their spanning tree, and G_1 has more nodes than G_2, then $m(G_1) < m(G_2)$
- $m(G) \leq m(K_n) = 1$, where G has n nodes, and K_n is the (undirected) complete graph with n nodes

Information flow metric

- Tree impurity metrics only consider the number of edges, not their “thickness”
- Henri & Kafura’s information flow metric takes this “thickness” into account
- based on notions of local and global flow
- we consider a later variant, developed by Shepperd

Shepperd's information flow metric

- *Local* flow from A to B (OR)
  - A invokes B and passes it a parameter
  - B invokes A and A returns a value
- *Global* flow from A to B (AND)
  - A updates some global structure
  - B reads that structure
- fan-in(M) = # (local and global) -> M
- fan-out(M) = # (local and global) M ->
- complexity(M) = (fan-in(M) * fan-out(M))^2

- Still, all flows count the same

Object-oriented metrics

- WMC: Weighted Methods per Class
- DIT: Depth of Inheritance Tree
- NOC: Number Of Children
- CBO: Coupling Between Object Classes
- RFC: Response For a Class
- LCOM: Lack of COhesion of a Method

Weighted Methods per Class

- measure for size of class
- WMC = $\sum$ c(i), i = 1, …, n (number of methods)
- c(i) = complexity of method i
- mostly, c(i) = 1

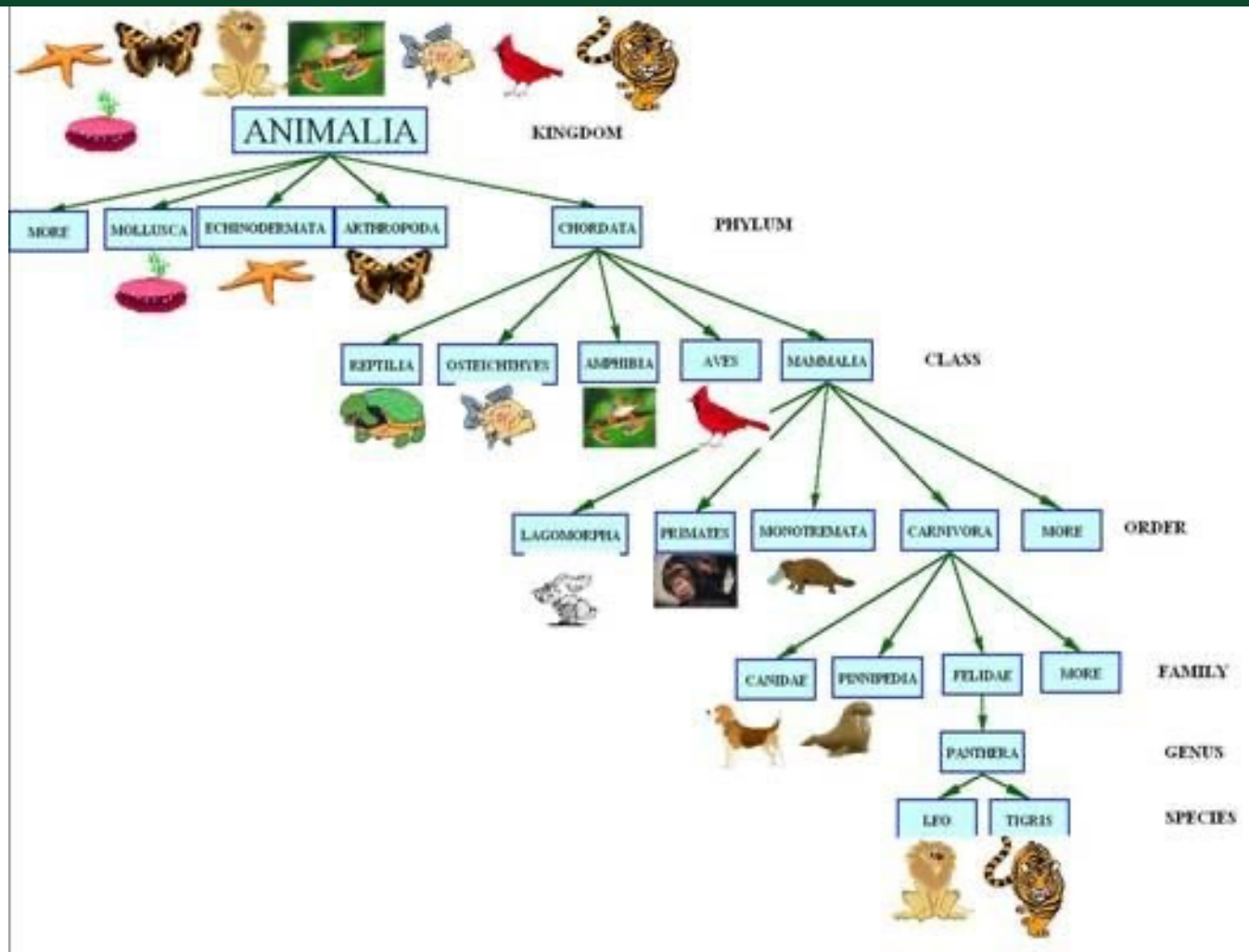
Depth of Class in Inheritance Tree

- DIT = distance of class to root of its inheritance tree
- DIT is somewhat language-dependent
- Widely accepted heuristic
  - Strive for a forest of classes
  - A collection of inheritance trees of medium height

Number Of Children

- NOC counts immediate descendants
- Higher values NOC are considered bad
  - possibly improper abstraction of the parent class
  - also suggests that class is to be used in a variety of settings

Number Of Children

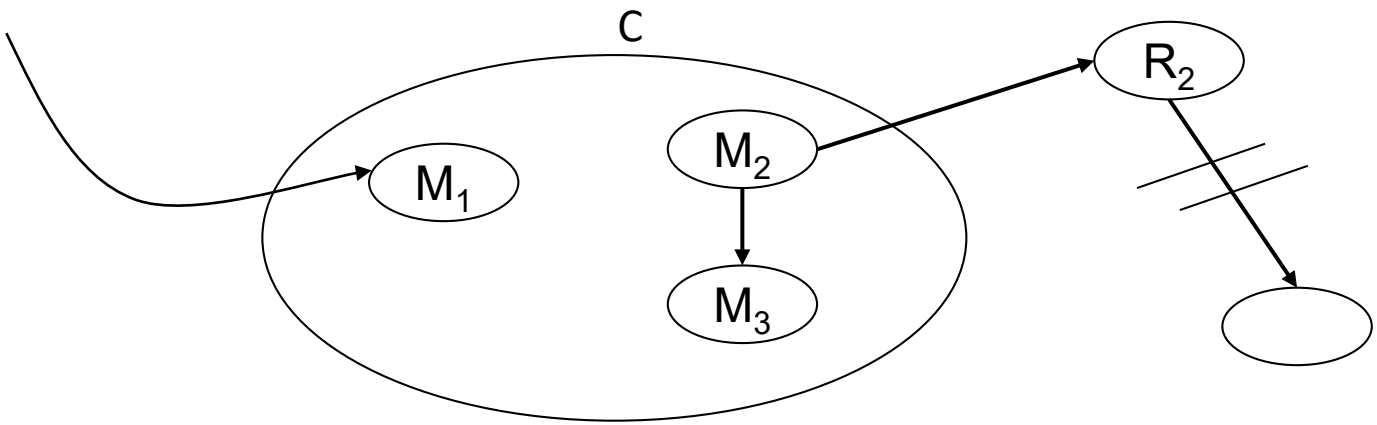


Coupling Between Object Classes

- Two classes are coupled if a method of one class uses a method or state variable of another class
- CBO = count of all classes a given class is coupled with
- High values
  - something is wrong
- All couplings are counted alike
  - refinements are possible

Response For a Class

- RFC measures the "immediate surroundings" of a class
- RFC = size of the "response set"
- ResponseSet(C) = {M} $\cup_i$ {Ri}



Lack of Cohesion of a Method

- Cohesion = glue that keeps the module (class) together
- All methods use the same set of state variables
  - OK
  - that is the glue
- Some methods use a subset of the state variables, and others use another subset
  - the class lacks cohesion
- LCOM = number of disjoint sets of methods in a class
- Two methods in the same set share at least one state variable

OO metrics

- WMC, CBO, RFC, LCOM most useful
  - Predict fault proneness during design
  - Strong relationship to maintenance effort
- Many OO metrics correlate strongly with size

Overview

- Introduction
- Design principles
- Design methods
- Conclusion

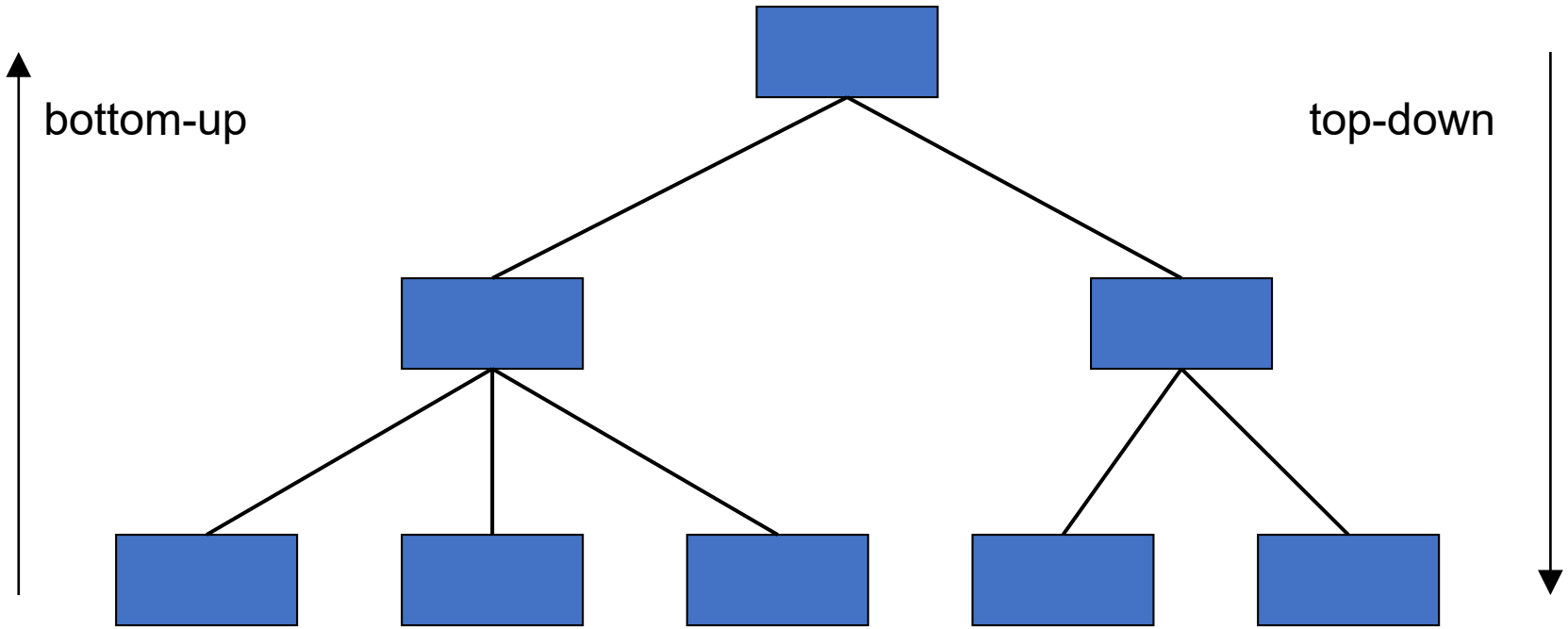
Design methods

- Functional decomposition
- Data Flow Design (SA/SD)
- Design based on Data Structures (JSD/JSP)
- OO is gOOd, isn't it

Design methods

- Decision tables
- Entity-Relationship
- Flowcharts
- Finite State Machines
- Jackson System Dev.
- Jackson Structured Prog.
- NoteCards
- OBJect
- OO Design
- Petri Nets
- Str. Analysis/Str. Design
- Str. Sys. An. & Des. Method

Functional decomposition



Functional decomposition

- Extremes: bottom-up and top-down
- Not used as such; design is not purely rational:
  - clients do not know what they want
  - changes influence earlier decisions
  - people make errors
  - projects do not start from scratch
- Rather, design has a yo-yo character
- We can only *fake* a rational design process
  - present the result of the design process as if it was conceived through a rational process

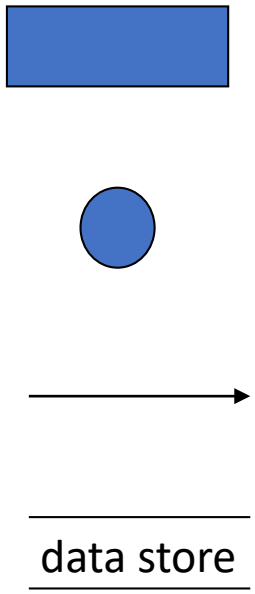
Data flow design

- Yourdon and Constantine (early 70s)

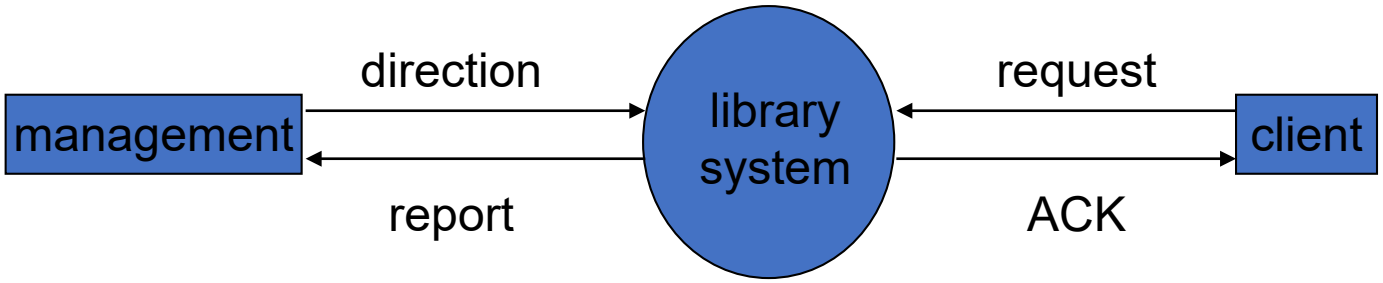
- Two-step process
  - Structured Analysis (SA -> logical design (**data flow diagrams**)
  - Structured Design (SD) logical design -> program structure (**structure charts**)

Entities in a data flow diagram

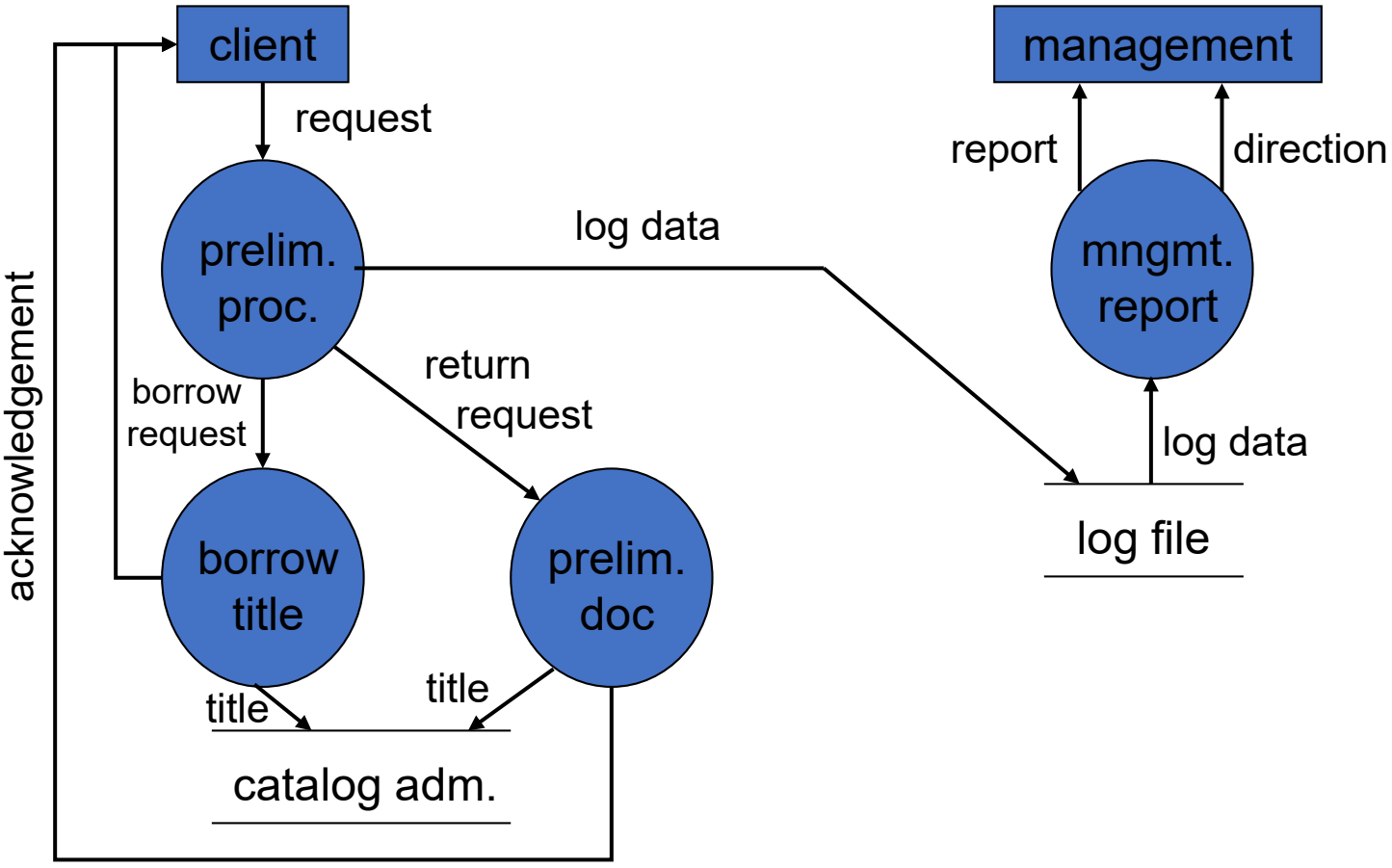
- external entities
- processes
- data flows
- data stores



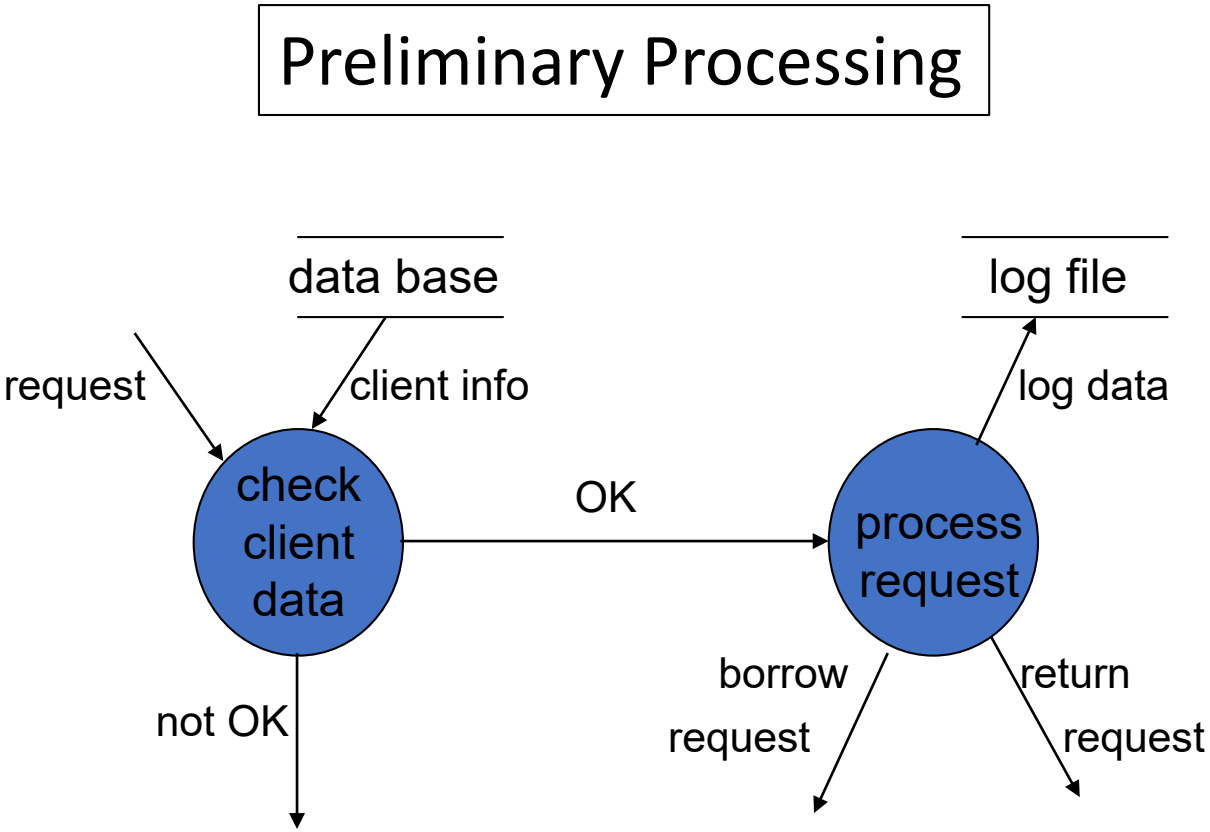
Top-level DFD: context diagram



First-level decomposition



Second-level decomposition



Entities in a data flow diagram

- Top-down decomposition stops
  - when a process becomes straightforward
  - does not warrant further expansion
- We produce minispecs
- Data dictionary is produced

Example minispec

```
Identification: Process request
Description:
1 Enter type of request
  1.1 If invalid, issue warning and repeat step 1
  1.2 If step 1 repeated 5 times, terminate
  transaction
2 Enter book identification
  2.1 If invalid, issue warning and repeat step 2
  2.2 If step 2 repeated 5 times, terminate
  transaction
3 Log client identification, request type and book
  identification
4 ...
```

Data dictionary entries

```
borrow-request = client-id + book-id
return-request = client-id + book-id
log-data = client-id + [borrow | return] + book-
  id
book-id = author-name + title + (isbn) +
  [proc | series | other]
```

Conventions:

[ ]: include one of the enclosed options

|: separates options

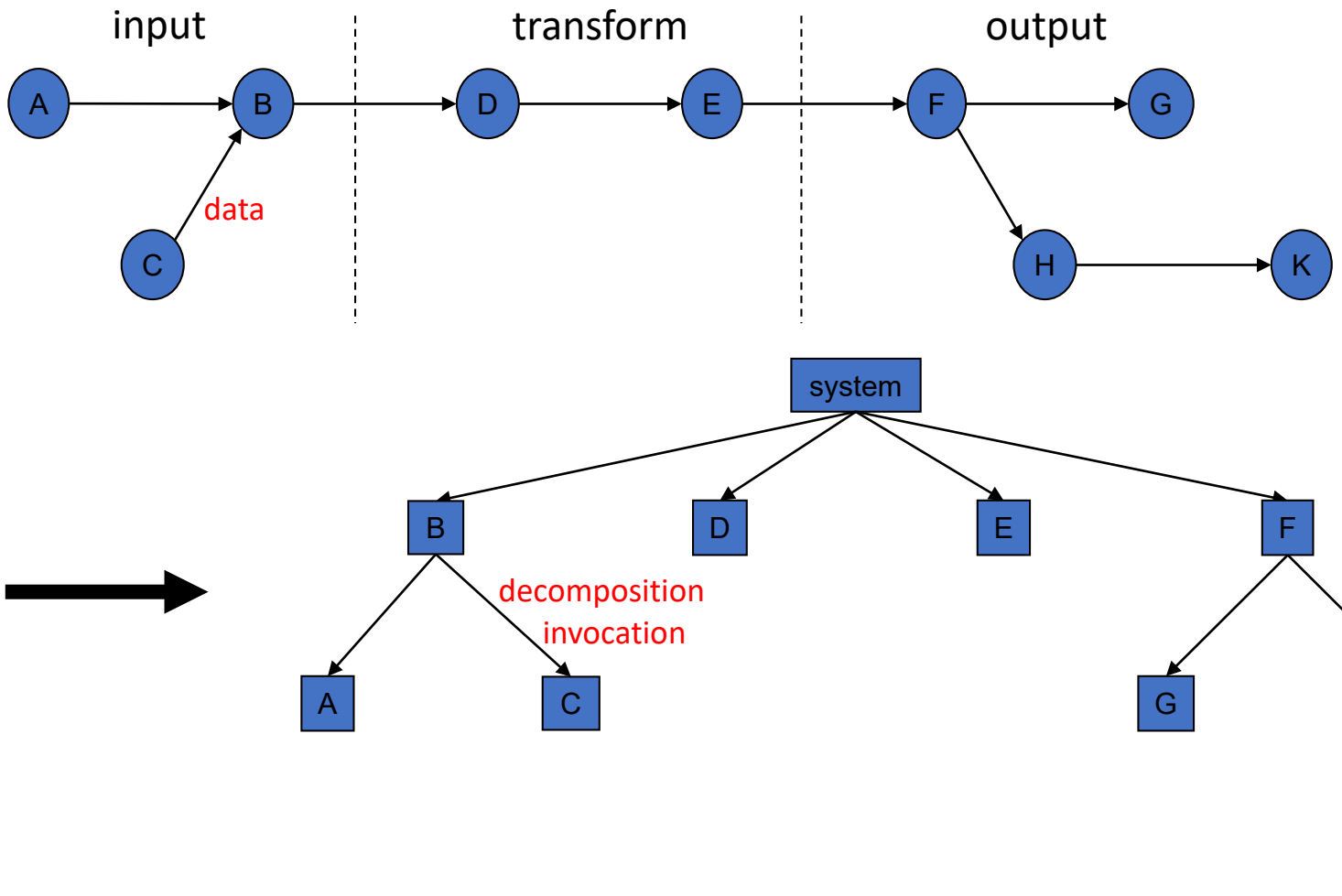
+: AND

(): enclosed items are optional

From data flow diagrams to structure charts

- result of SA: logical model, consisting f a set of DFD's, augmented by minispecs, data dictionary, etc.
- Structured Design = transition from DFD's to structure charts
- heuristics for this transition are based on notions of coupling and cohesion
- major heuristic concerns choice for top-level structure chart, most often: *transform-centered*

Transform-centered design



Design based on data structures

- JSP = Jackson Structured Programming
  (for programming-in-the-small)

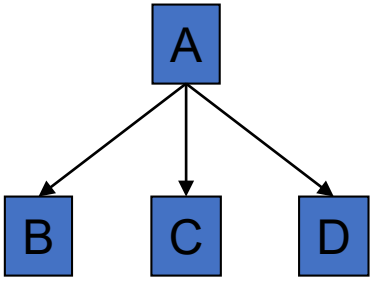
- JSD = Jackson Structured Design
  (for programming-in-the-large)

JSP

- Basic idea: good program reflects structure of its input and output
- Program can be derived from a description of the input and output
- Input and output are depicted in a *structure diagram* and/or *structured text/schematic logic* (a kind of pseudocode)
- Three basic compound forms: sequence, iteration, and selection

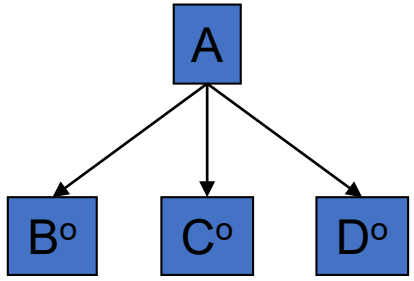
Compound components in JSP

sequence

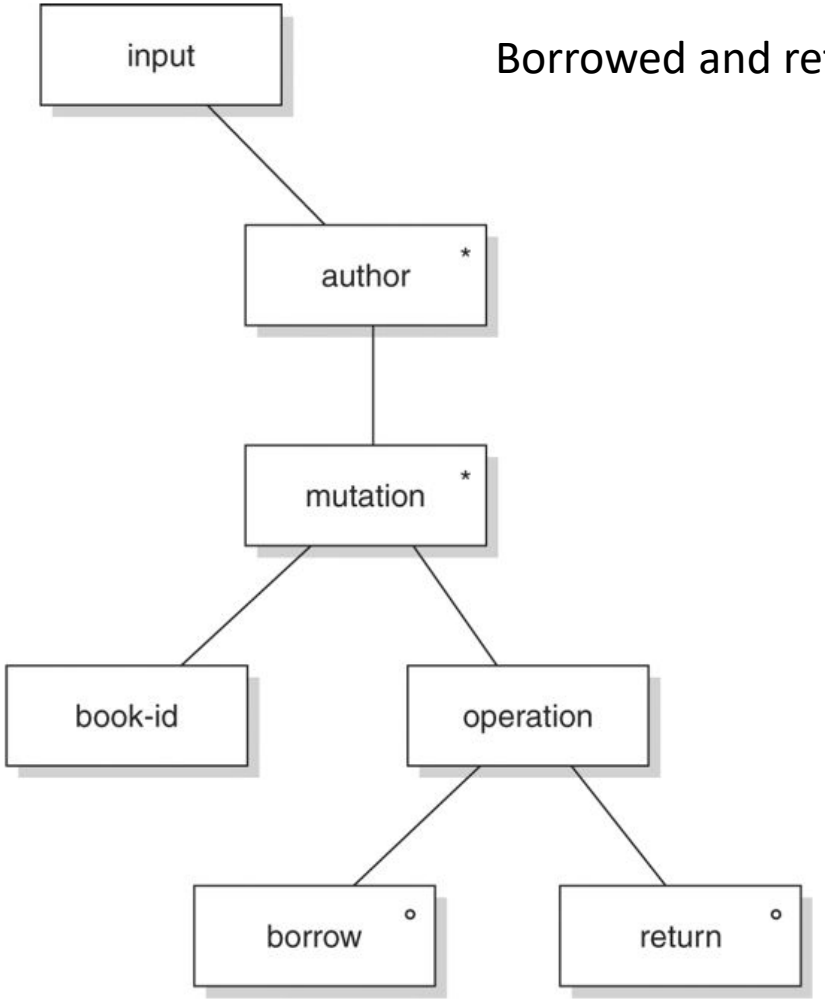


iteration

selection

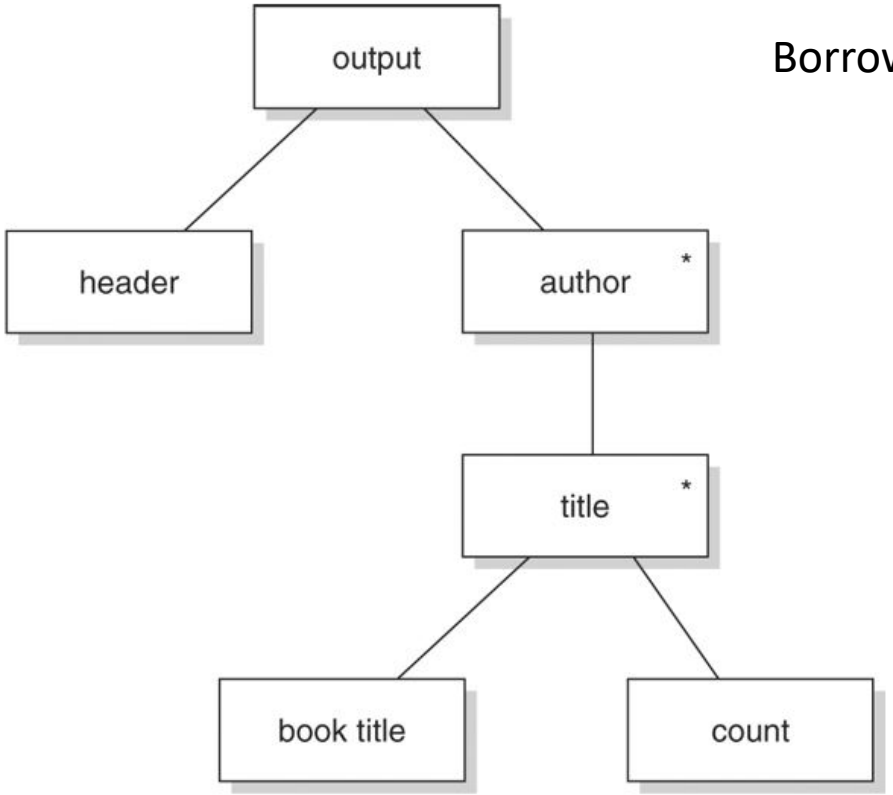


A JSP example



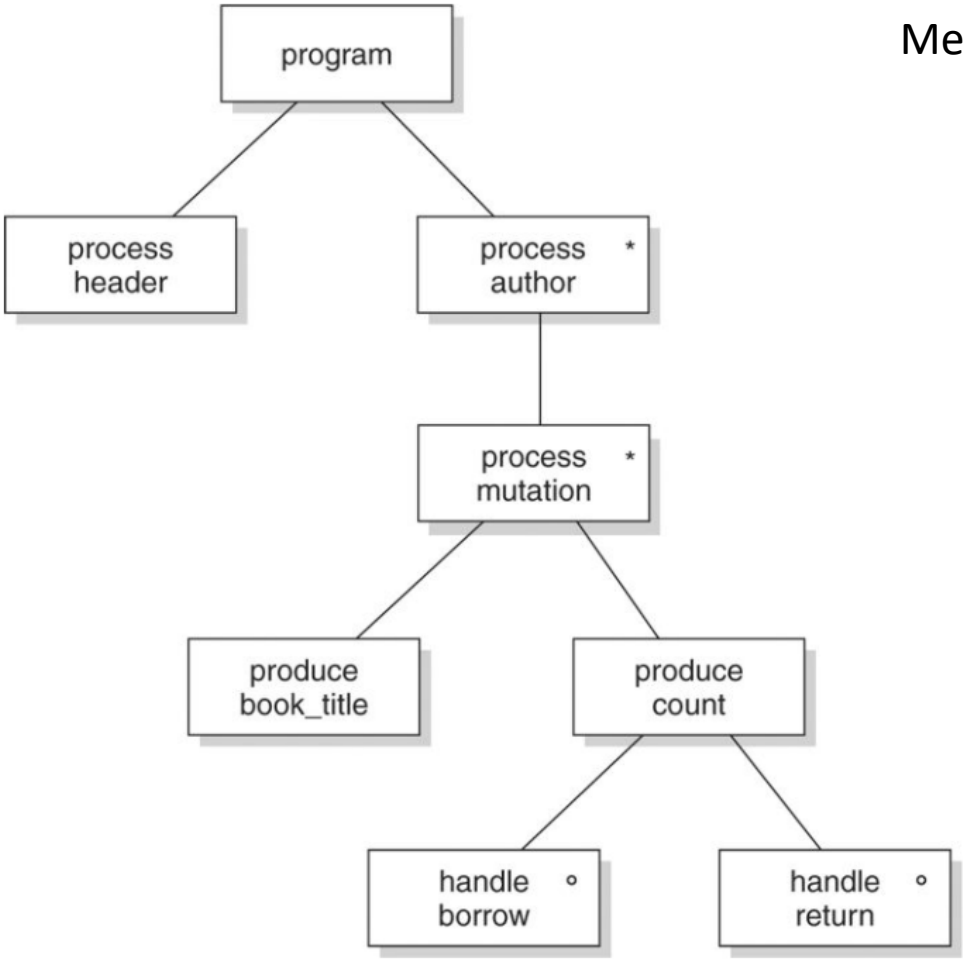
Borrowed and returned books Log

Another JSP example



Borrowed books Report

Another JSP example



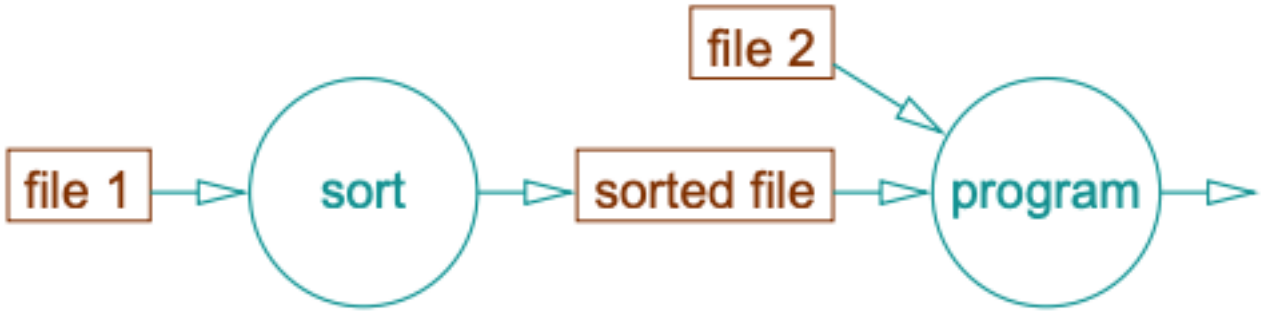
Merged diagrams

Another JSP example

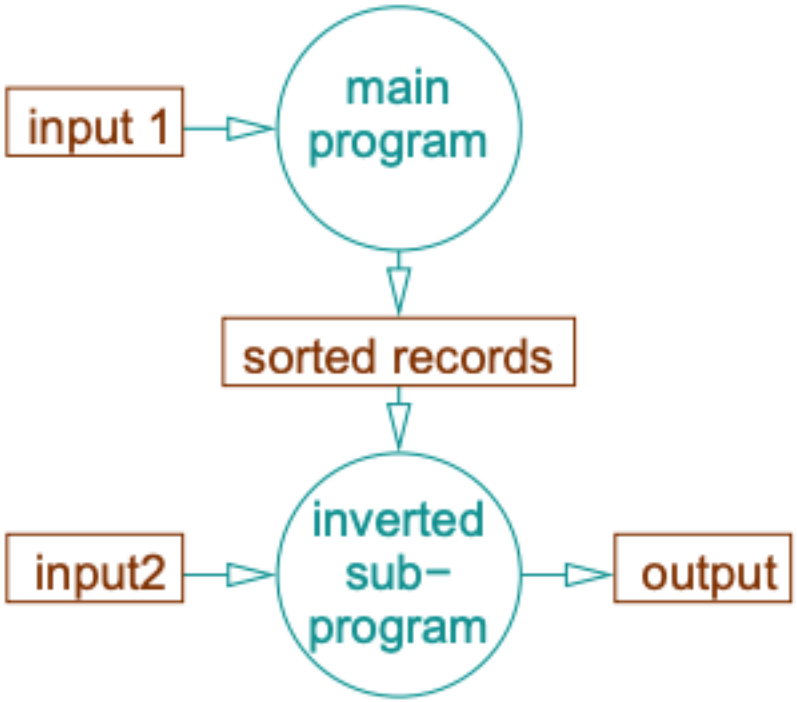
Minicode of merged diagrams

```
make header
until EOF loop
    process author:
        until end_of_author loop
            process_mutation:
                ...
        endloop
endloop.
```

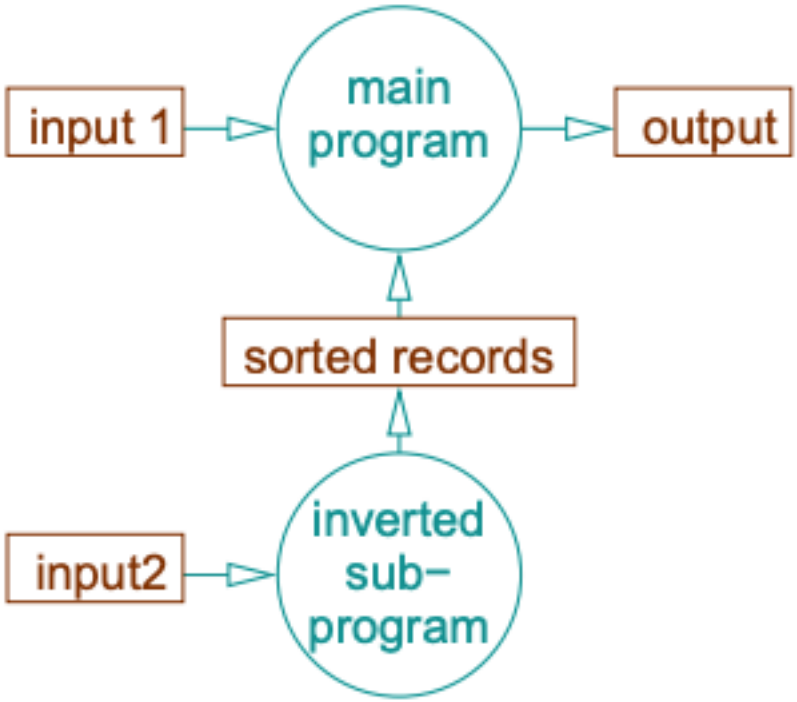
Structure clash



Inversion



main program sorts



subprogram sorts

Fundamental issues in JSP

- Model input and output using structure diagrams
- Merge diagrams to create program structure
- Meanwhile, resolve structure clashes, and
- Optimize results through program inversion

Difference between JSP and other methods

- Functional decomposition, data flow design:
  Problem structure $\Rightarrow$ functional structure $\Rightarrow$
  program structure

- JSP:
  Problem structure $\Rightarrow$ data structure $\Rightarrow$
  program structure

Jackson System Development

- Problem with JSP: how to obtain a mapping from the problem structure to the data structure?
- JSD tries to fill this gap
- JSD has three stages:
  - **modeling stage**: description of real world problem in terms of entities and actions
  - **network stage**: model system as a network of communicating processes
  - **implementation stage**: transform network into a sequential design

JSD's modeling stage

- JSD models the UoD as a set of entities
- For each entity, a process is created (models the life cycle of that entity)
- Life cycle is depicted as a *process structure diagram (PSD)*
- PSD's are finite state diagrams
  - roles of nodes and edges are reversed
  - in PSD nodes=transitions, edges=states

OO Analysis and Design

Three major steps:

1 identify the objects

2 determine their attributes and services

3 determine the relationships between objects

(Part of) Problem statement

Design the software to support the operation of a public library. The system has a number of stations for customer transactions. These stations are operated by library employees. When a book is borrowed, the identification card of the client is read. Next, the station's bar code reader reads the book's code. When a book is returned, the identification card is not needed and only the book's code needs to be read.

Candidate objects

- software
- library
- system
- station
- customer
- transaction
- book
- library employee
- identification card
- client
- bar code reader
- book's code

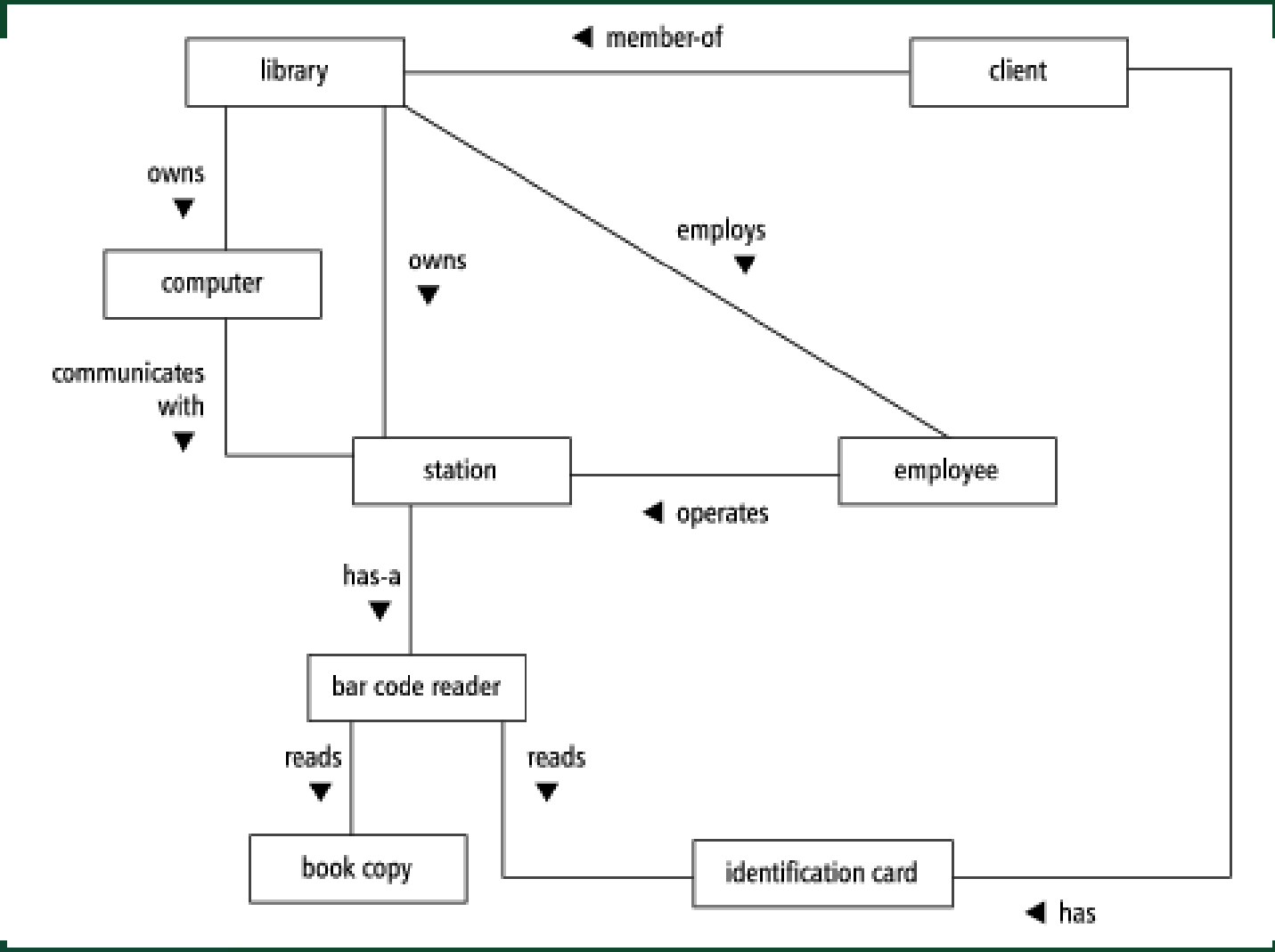
Carefully consider candidate list

- Eliminate implementation constructs, such as “software”
- Replace or eliminate vague terms: “system” $\Rightarrow$ “computer”
- Gather synonymous terms: “customer” and “client” $\Rightarrow$ “client”
- Eliminate operation names, if possible (such as “transaction”)
- Be careful in what you *really* mean
  - can a client be a library employee?
  - Is it “book copy” or “book”?
- Eliminate individual objects (as opposed to classes).
  - book’s code $\Rightarrow$ attribute of “book copy”

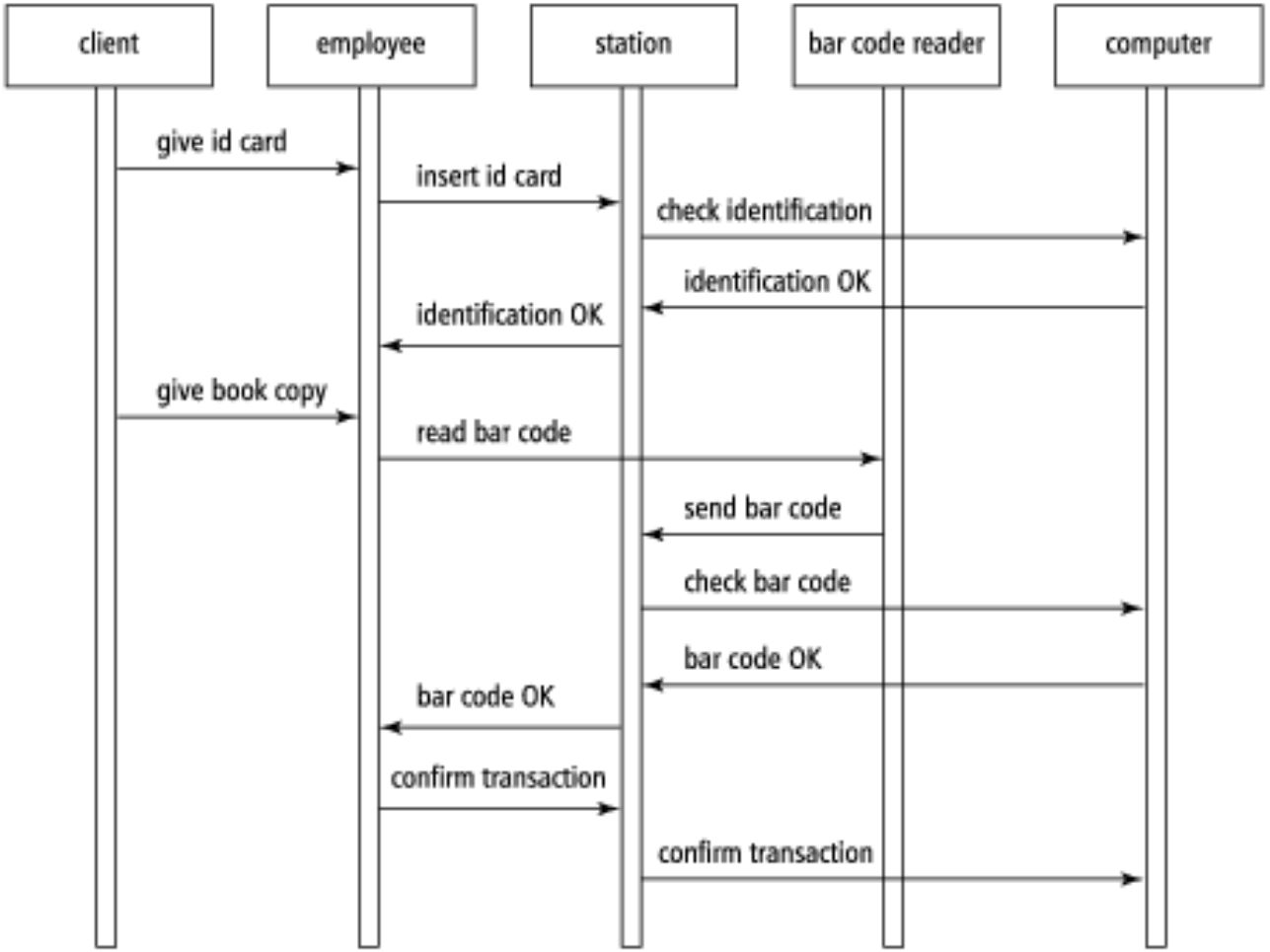
Relationships

- From the problem statement:
  - employee operates station
  - station has bar code reader
  - bar code reader reads book copy
  - bar code reader reads identification card
- Tacit knowledge:
  - library owns computer
  - library owns stations
  - computer communicates with station
  - library employs employee
  - client is member of library
  - client has identification card

Result: initial class diagram

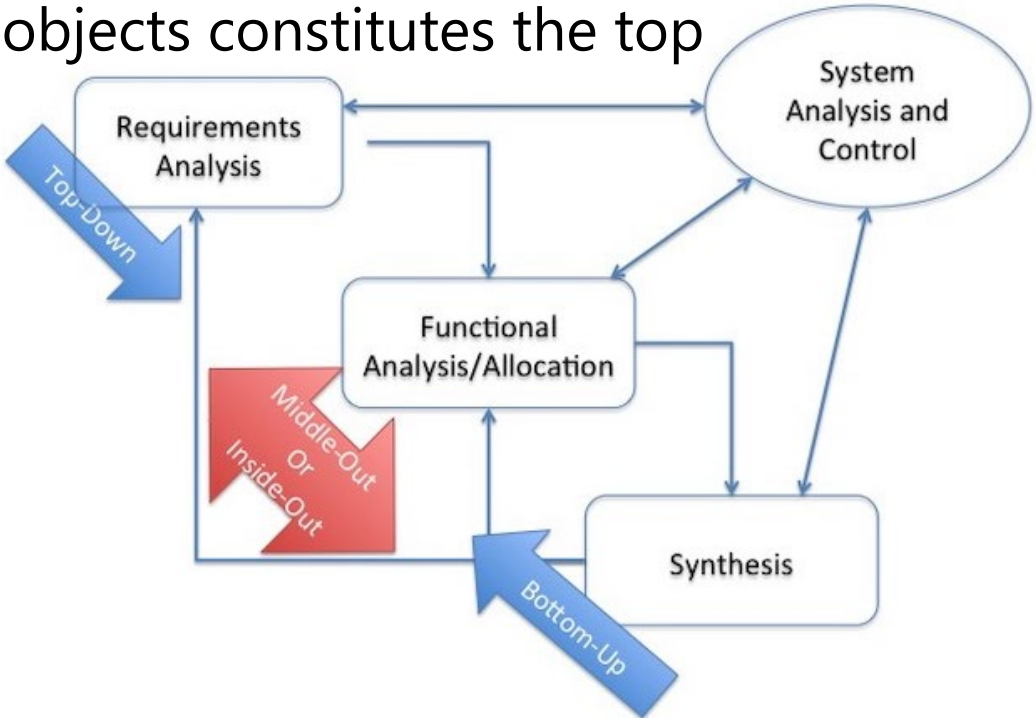


Usage scenario ⇒ sequence diagram



OO as middle-out design

- First set of objects becomes middle level
- To implement these, lower-level objects are required (class library)
- A control/workflow set of objects constitutes the top level



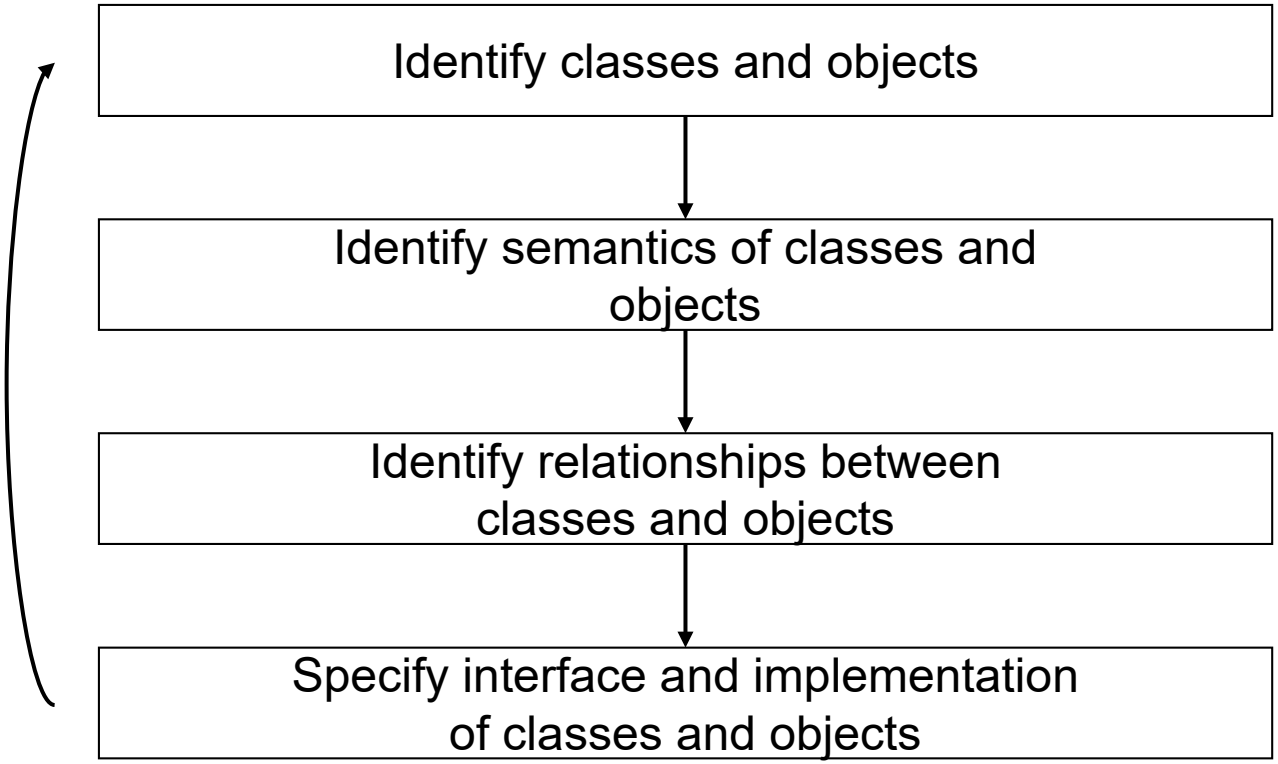
OO design methods

- Booch:
  Early, new and rich set of notations

- Fusion:
  More emphasis on process

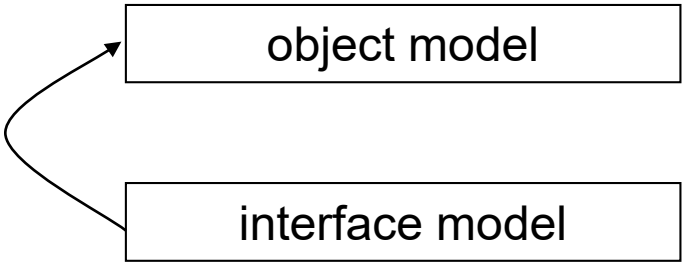
- RUP (Rational Unified Process):
  Full life cycle model associated with UML

Booch' method

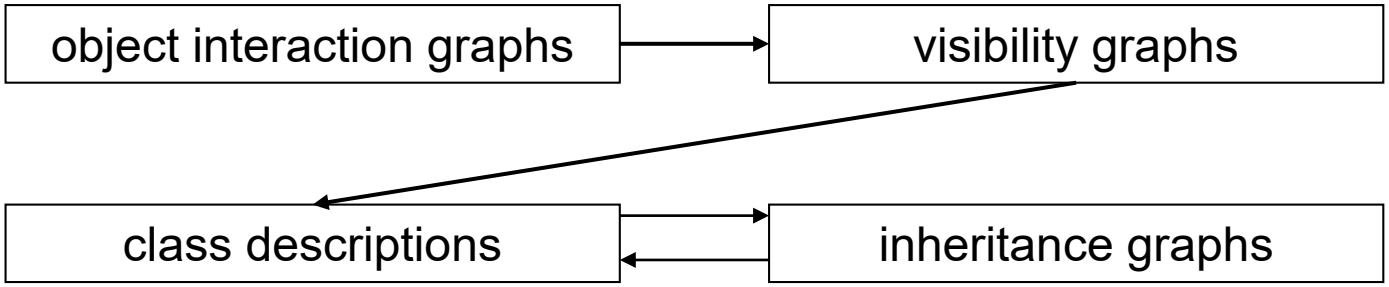


Fusion

Analysis



Design



RUP

- Nine workflows, a.o.
  - Requirements
  - Analysis
  - design
- Four phases
  - inception
  - elaboration
  - construction
  - transition
- Analysis and design workflow
  - First iterations: architecture
  - Analyze behavior - from use cases to set of design elements; produces black-box model of the solution
  - Design components: refine elements into classes, interfaces, etc.

Classification of design methods

- Simple model with two dimensions:

- Orientation dimension:
  - Problem-oriented: understand problem and its solution
  - Product-oriented: correct transformation from specification to implementation

- Product/model dimension:
  - Conceptual: descriptive models
  - Formal: prescriptive models

Classification of design methods (cnt'd)

	problem-oriented	product-oriented
conceptual	I ER modeling Structured analysis	II Structured design
formal	III JSD VDM	IV Functional decomposition JSP

Characteristics of these classes

- I: Understand the problem
- II: Transform to implementation
- III: Represent properties
- IV: Create implementation units

Caveats when choosing a particular design method

- Familiarity with the problem domain
- Designer's experience
- Available tools
- Development philosophy

Object-orientation: does it work?

- do object-oriented methods adequately capture requirements engineering?
- do object-oriented methods adequately capture design?
- do object-oriented methods adequately bridge the gap between analysis and design?
- are oo-methods really an improvement?

Design pattern

- Provides solution to a recurring problem
- Balances set of opposing forces
- Documents well-prove design experience
- Abstraction above the level of a single component
- Provides common vocabulary and understanding
- Are a means of documentation
- Supports construction of software with defined properties

Example design pattern: Proxy

- Context:
  - Client needs services from other component, direct access may not be the best approach
- Problem:
  - We do not want hard-code access
- Solution:
  - Communication via a representative, the *Proxy*

Example design pattern: Command Processor

- Context:
  - User interface that must be flexible or provides functionality beyond handling of user functions
- Problem:
  - Well-structured solution for mapping interface to internal functionality. All ‘extras’ are separate from the interface
- Solution:
  - A separate component, the *Command Processor*, takes care of all commands Actual execution of the command is delegated

Antipatterns

- Patterns describe desirable behavior
- Antipatterns describe situations one had better avoid
- In agile approaches (XP), *refactoring* is applied whenever an antipattern has been introduced

Example antipatterns

- *God class*: class that holds most responsibilities
- *Lava flow*: dead code
- *Poltergeist*: class with few responsibilities and a short life
- *Golden Hammer*: solution that does not fit the problem
- *Stovepipe*: (almost) identical solutions at different places
- *Swiss Army Knife*: excessively complex class interface

Overview

- Introduction
- Design principles
- Design methods
- Conclusion

Conclusion

- Essence of the design process
  - decompose system into parts
- Desirable properties of a decomposition
  - coupling/cohesion
  - information hiding
  - (layers of) abstraction
- There have been many attempts to express these properties in numbers
- Design methods
  - functional decomposition
  - data flow design
  - data structure design
  - object-oriented design