

## Vragen

- Wat wordt bedoeld met het ontwerpsprincipe: “Anticipate obsolescence”?
- Wat is het voordeel van “strong cohesion” en “weak coupling”?
- Wat is het gevolg van hoge complexiteit icm ontwerp?

## Intra-modular complexity measures

- for small programs, the various measures correlate well with programming time
- however, a simple length measure such as LOC does equally well
- complexity measures are not very context sensitive
- complexity measures take into account few aspects
  
- it might help to look at the complexity *density* instead

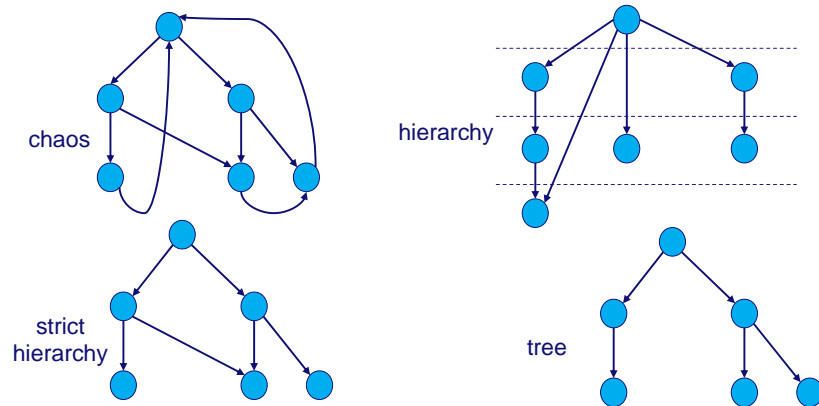
## System structure: inter-module complexity

- looks at the complexity of the dependencies *between* modules
- draw modules and their dependencies in a graph
- then the arrows connecting modules may denote several relations, such as:
  - A contains B
  - A precedes B
  - A uses B
- we are mostly interested in the latter type of relation

## The *uses* relation

- In a well-structured piece of software, the dependencies show up as procedure calls
- therefore, this graph is known as the *call-graph*
- possible shapes of this graph:
  - chaos (directed graph)
  - hierarchy (acyclic graph)
  - strict hierarchy (layers)
  - tree

## In a picture



## OO Metrics

- **WMC:** “weighted methods per class” based on cyclomatic complexity, size, etc. per method
- **DIT:** “depth of class in inheritance tree” distance to top of inheritance tree
- **NOC:** “number of children” counts direct descendants of a class

## OO Metrics

- **CBO:** “coupling between object class” counts the number of class a class is connected to via method or variable
  - afferent coupling: dependence of a package on its environment
  - efferent coupling: dependence of the environment on a package
- **RFC:** “response for a class”
- **LCOM:** “lack of cohesion of a method”

## Design methods

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

## List of possible design methods

- Decision tables
- E-R
- Flowcharts
- FSM
- JSD
- JSP
- LCP
- Meta IV
- NoteCards
- OBJ
- OOD
- PDL
- Petri Nets
- SA/SD
- SA/WM
- SADT
- SSADM
- Statecharts

## 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

## Data flow design

- Yourdon and Constantine (early 70s)
- nowadays version: two-step process:
  - Structured Analysis (SA), resulting in a logical design, drawn as a set of data flow diagrams
  - Structured Design (SD) transforming the logical design into a program structure drawn as a set of structure charts

## Design based on data structures (JSP & JSD)

- 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 almost mechanically from a description of the input and output
- input and output are depicted in a *structure diagram* and/or in *structured text/schematic logic* (a kind of pseudocode)
- three basic compound forms: sequence, iteration, and selection)

## 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

## JSD: Jackson Structured Design

- 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 which models the life cycle of that entity
- This life cycle is depicted as a *process structure diagram (PSD)*; these resemble JSP's structure diagrams
- PSD's are finite state diagrams; only the roles of nodes and edges has been reversed: in a PSD, the nodes denote transitions while the edges denote states

## OOAD methods

- 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

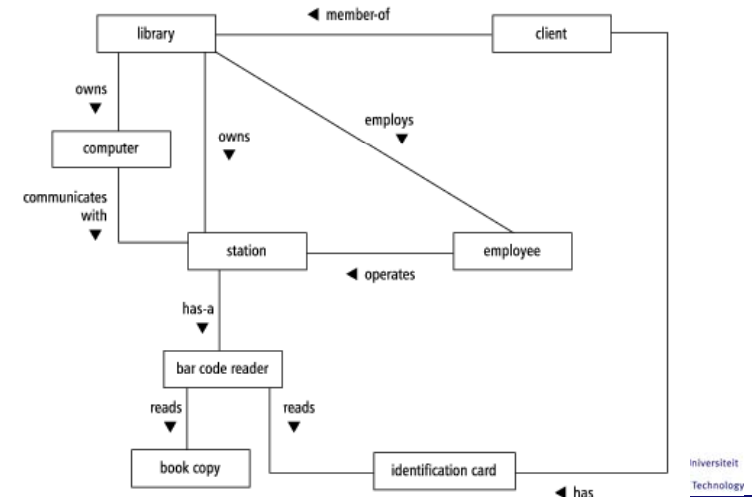
## Carefully consider candidate list

- eliminate implementation constructs, such as “software”
- replace or eliminate vague terms: “system” ⇒ “computer”
- equate synonymous terms: “customer” and “client” ⇒ “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” rather than “book”?
- eliminate individual objects (as opposed to classes). “book's code” ⇒ attribute of “book copy”

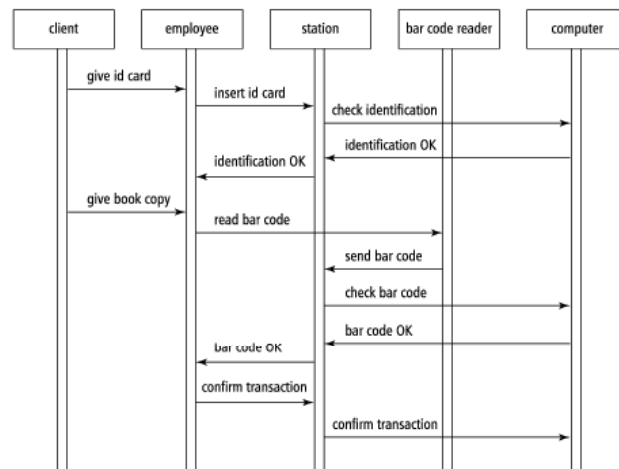
## Relationships

- 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



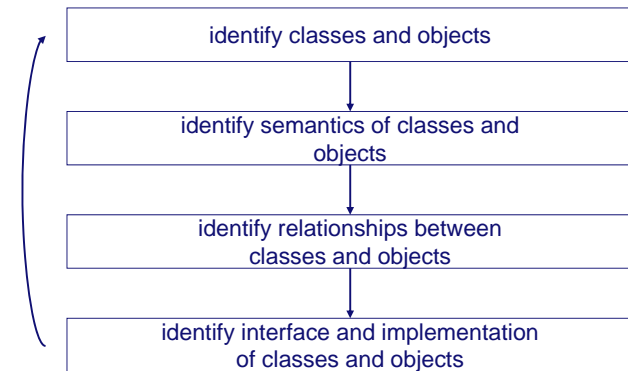
## OO as middle-out design

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

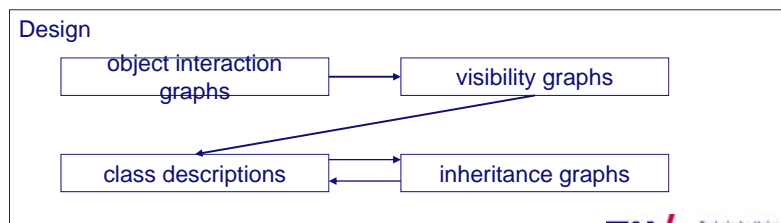
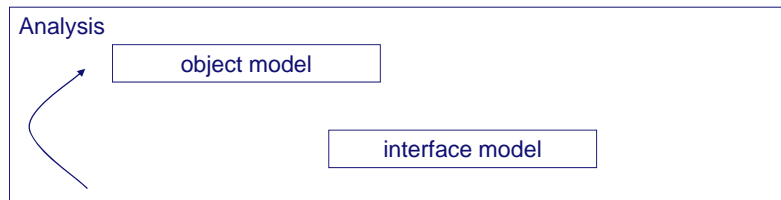
## OO design methods

- **Booch:** early, new and rich set of notations
- **Fusion:** more emphasis on process
- **RUP:** full life cycle model associated with UML

## Booch' method



## Fusion



## RUP

- **Nine workflows, a.o. requirements, analysis and design**
- **Four phases: inception, elaboration, construction, transition**
- **Analysis and design workflow:**
  - **First iterations:** architecture discussed in ch 11
  - **Next: analyze behavior:** from use cases to set of design elements; produces black-box model of the solution
  - **Finally, 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?

## Complexity

- measure certain aspects of the software (lines of code, # of if-statements, depth of nesting, ...)
- use these numbers as a criterion to assess a design, or to 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

## 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

## 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

## 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

## Techniques for making good design decisions

- Using priorities and objectives to decide among alternatives
  - Step 1: List and describe the alternatives for the design decision.
  - Step 2: List the advantages and disadvantages of each alternative with respect to your objectives and priorities.
  - Step 3: Determine whether any of the alternatives prevents you from meeting one or more of the objectives.
  - Step 4: Choose the alternative that helps you to best meet your objectives.
  - Step 5: Adjust priorities for subsequent decision making.