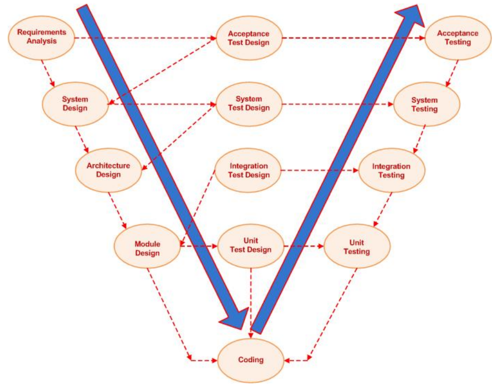
Software Design
Chapter 12

Mark van den Brand

V model for software development



Software design

- Programmer's approach:
  - Skip requirements engineering and design phases
  - Start writing code
- Why?
  - 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

Software design

- Design is a trial-and-error process
- There is an interaction between requirements engineering, architecting, and design
- Design traps:
  - There is no definite formulation
  - There is no stopping rule
  - Solutions are not simply true or false
  - There may be a whole range of possible (good) solutions

Process of design

- *Design* is a problem-solving process whose objective is to find and describe a way:
  - To implement the system's *functional requirements*...
  - While respecting the constraints imposed by the *quality, platform and process requirements...*
    - including the budget
  - And while adhering to general principles of *good quality*

Design Principle 1: Divide and conquer

- Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things
  - Separate people can work on each part.
  - An individual software engineer can specialize.
  - Each individual component is smaller, and therefore easier to understand.
  - Parts can be replaced or changed without having to replace or extensively change other parts.

Design Principle 2: Increase cohesion where possible

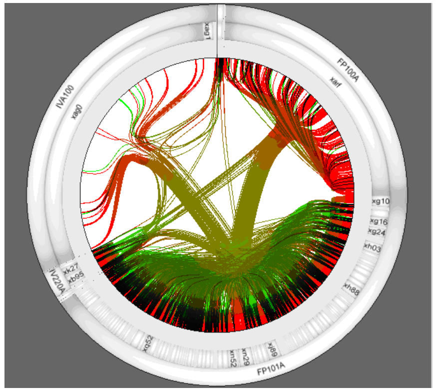
- A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things
  - This makes the system as a whole easier to understand and change
  - Type of cohesion:
    - Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

Design Principle 3: Reduce coupling where possible

- *Coupling* occurs when there are *interdependencies* between one module and another
  - When interdependencies exist, changes in one place will require changes somewhere else.
  - A network of interdependencies makes it hard to see at a glance how some component works.
  - Types of coupling:
    - Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, External

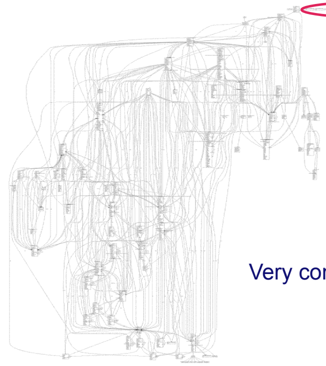
Cohesion and coupling

- **Dependencies**
  - **A lot of open spaces**
  - **1216 modules not called by other modules**
  - **This may be dead code**
  - **651 modules indeed dead (confirmed)**



Cohesion and coupling

Automatic model extraction shows:



Potentially unused classes!

Very complex!

Design Principle 4: Keep the level of abstraction as high as possible

- **Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity**
  - **A good abstraction is said to provide *information hiding***
  - **Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details**

Design Principle 5: Increase reusability where possible

- **Design the various aspects of your system so that they can be used again in other contexts**
  - **Generalize your design as much as possible**
  - **Follow the preceding three design principles**
  - **Design your system to contain hooks**
  - **Simplify your design as much as possible**

Design Principle 6: Reuse existing designs and code where possible

- Design with reuse is complementary to design for reusability
  - Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
    - *Cloning* should not be seen as a form of reuse
  - Use frameworks/libraries as much as possible

Design Principle 7: Design for flexibility

- Actively anticipate changes that a design may have to undergo in the future, and prepare for them
  - Reduce coupling and increase cohesion
  - Create abstractions
  - Do not hard-code anything
  - Leave all options open
    - Do not restrict the options of people who have to modify the system later
  - Use reusable code and make code reusable

Design Principle 8: Anticipate obsolescence

- Plan for changes in the technology or environment so the software will continue to run or can be easily changed
  - Avoid using early releases of technology
  - Avoid using software libraries that are specific to particular environments
  - Avoid using undocumented features or little-used features of software libraries
  - Avoid using software or special hardware from companies that are less likely to provide long-term support
  - Use standard languages and technologies that are supported by multiple vendors

Design Principle 9: Design for Portability

- Have the software run on as many platforms as possible
  - Avoid the use of facilities that are specific to one particular environment
  - E.g. a library only available in Microsoft Windows

Questions

- Why is *design* necessary?
- Is *design* the same as programming?
- Why is *low coupling* and *high cohesion* good?
- Is *code cloning* a good form of *re-use*?

Design Principle 10: Design for Testability

- Take steps to make testing easier
  - Design a program to automatically test the software
    - Discussed more in Chapter 13
    - Ensure that all the functionality of the code can by driven by an external program, bypassing a graphical user interface
  - In Java, you can create a main() method in each class in order to exercise the other methods

Design Principle 11: Design defensively

- Never trust how others will try to use a component you are designing
  - Handle all cases where other code might attempt to use your component inappropriately
  - Check that all of the inputs to your component are valid: the *preconditions*
    - Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking
    - Example: 75% of the code is used to parameter checking

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
- Data abstraction
  - aimed at finding a hierarchy in the data

Modularity

- Structural criteria which tell us something about individual modules and their interconnections
- Modern programming languages support modularity
- Cohesion and coupling
  - cohesion: the glue that keeps a module together
  - coupling: the strength of the connection between modules
  - keep track of this via measuring!

Modularity

- Calculating quality metrics on the source code

***Fan Out** (# modules called)*

Layer	Unit	Module	LOCs	omment	Blanks	Source	IFs	LOOPs	McCabe	Fan_in	Fan_out	CLN	NBR	RSA	RSI	CVR	id
'CobolPr	'XOFC'	'XOFC27	17480	2333	310	14837	256	63	320	0	21	768	8	0.27200	0.60300	0.71600	0
'CobolPr	'XOFC'	'XOFC27	16331	1627	463	14241	194	105	300	1	53	887	8	0.53000	0.55900	0.71700	1
'CobolPr	'XOFC'	'XOFC27	8722	707	313	7702	105	25	131	1	20	715	8	0.82900	0.51500	0.86100	3
'CobolPr	'XOFC'	'XOFC27	11391	911	598	9882	87	26	114	1	31	771	8	0.69600	0.49500	0.73000	2
'CobolPr	'XUFC'	'XUFC06	1688	249	104	1335	47	2	50	1	0	66	8	0.21700	0.19400	0.27200	8
'CobolPr	'XIFC'	'XIFC050	1100	147	65	888	33	5	39	1	5	25	8	0.15900	0.51700	0.59900	6
'CobolPr	'XJFC'	'XJFC630	3000	185	199	2616	11	12	24	1	7	176	8	0.27600	0.51400	0.66200	4
'Rekenre	'XARF'	'XARF03	384	49	31	304	10	0	11	1	0	7	4	0.09800	0.34300	0.44000	14
'Datalaye	'XU06'	'XU0610	671	106	56	509	8	0	9	1	0	4	2	0.02300	0.63000	0.64500	11
'CobolPr	'XNFC'	'XNFC86	1112	133	143	836	6	0	7	1	6	115	8	0.54300	0.40900	0.56400	5
'CobolPr	'XNFC'	'XNFC86	630	109	52	469	2	1	4	1	8	48	8	0.37900	0.16700	0.37900	7
'Datalaye	'XU06'	'XU0610	180	36	34	110	2	0	3	1	0	2	2	0.13900	0.00000	0.13900	9
'Rekenre	'XARF'	'XARF02	130	10	28	92	2	0	3	1	0	3	4	0.48900	0.00000	0.48900	13
'Datalaye	'XU06'	'XU0610	177	41	33	103	2	0	3	1	0	2	2	0.13900	0.00000	0.13900	10
'Rekenre	'XARF'	'XARF03	119	10	28	81	0	0	1	1	0	7	4	0.94800	0.00000	0.94800	15
'Rekenre	'XARF'	'XARF03	117	10	28	79	0	0	1	1	0	3	4	0.59800	0.00000	0.59800	16
'Rekenre	'XARF'	'XARF03	119	10	28	81	0	0	1	1	0	7	4	0.94300	0.00000	0.94300	12

***Fan In** (called by # modules)*

Types of cohesion

- Coincidental cohesion
  - elements are grouped into components in a random manner, no relation between components
- Logical cohesion
  - elements realize logical related tasks, for instance all procedures dealing with the processing of input
- Temporal cohesion
  - elements are independent but are active at the same moment in time, for instance everything related to initialization
- Procedural cohesion
  - elements are executed in a given order

Types of cohesion

- Communicational cohesion
  - elements operate on the same (external) data
- Sequential cohesion
  - sequence of elements where output of one is input for other
- Functional cohesion
  - elements contribute to a single function

- Data cohesion (for abstract data types)

Types of coupling

- Content coupling
  - change of data by another component
- Common coupling
  - shared data
- External coupling
  - files
- Control coupling
  - flags
- Stamp coupling
  - shared knowledge on data formats
- Data coupling
  - simple data

strong cohesion & weak coupling ⇒ simple interfaces ⇒

- simpler communication
- simpler correctness proofs
- changes influence other modules less often
- reusability increases
- comprehensibility improves

Information hiding

- Design involves a series of decision: for each such decision, wonder who needs to know and 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

Questions

- What is meant by "*design defensively*"?
- What is the consequence of high complexity in design?

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 ⇒ higher complexity ⇒ more effort required (= worse design)
- Two kinds:
  - intra-modular: inside one module
  - inter-modular: between modules

Modularity

- Calculating quality metrics on the source code

Fan Out *(# modules called)*

Layer	Unit	Module	LOCs	omment	Blanks	Source	IFs	LOOPs	McCabe	Fan_in	Fan_out	CLN	NBR	RSA	RSI	CVR	id
'CobolPr	'XOFC'	'XOFC27	17480	2333	310	14837	256	63	320	0	21	768	8	0.27200	0.60300	0.71600	0
'CobolPr	'XOFC'	'XOFC27	16331	1627	463	14241	194	105	300	1	53	887	8	0.53000	0.55900	0.71700	1
'CobolPr	'XOFC'	'XOFC27	8722	707	313	7702	105	25	131	1	20	715	8	0.82900	0.51500	0.86100	3
'CobolPr	'XOFC'	'XOFC27	11391	911	598	9882	87	26	114	1	31	771	8	0.69600	0.49500	0.73000	2
'CobolPr	'XUFC'	'XUFC06	1688	249	104	1335	47	2	50	1	0	66	8	0.21700	0.19400	0.27200	8
'CobolPr	'XIFC'	'XIFC050	1100	147	65	888	33	5	39	1	5	25	8	0.15900	0.51700	0.59900	6
'CobolPr	'XJFC'	'XJFC630	3000	185	199	2616	11	12	24	1	7	176	8	0.27600	0.51400	0.66200	4
'Rekenre	'XARF'	'XARF03	384	49	31	304	10	0	11	1	0	7	4	0.09800	0.34300	0.44000	14
'Datalaye	'XU06'	'XU0610	671	106	56	509	8	0	9	1	0	4	2	0.02300	0.63000	0.64500	11
'CobolPr	'XNFC'	'XNFC86	1112	133	143	836	6	0	7	1	6	115	8	0.54300	0.40900	0.56400	5
'CobolPr	'XNFC'	'XNFC86	630	109	52	469	2	1	4	1	8	48	8	0.37900	0.16700	0.37900	7
'Datalaye	'XU06'	'XU0610	180	36	34	110	2	0	3	1	0	2	2	0.13900	0.00000	0.13900	9
'Rekenre	'XARF'	'XARF02	130	10	28	92	2	0	3	1	0	3	4	0.48900	0.00000	0.48900	13
'Datalaye	'XU06'	'XU0610	177	41	33	103	2	0	3	1	0	2	2	0.13900	0.00000	0.13900	10
'Rekenre	'XARF'	'XARF03	119	10	28	81	0	0	1	1	0	7	4	0.94800	0.00000	0.94800	15
'Rekenre	'XARF'	'XARF03	117	10	28	79	0	0	1	1	0	3	4	0.59800	0.00000	0.59800	16
'Rekenre	'XARF'	'XARF03	119	10	28	81	0	0	1	1	0	7	4	0.94300	0.00000	0.94300	12

Fan In *(called by # modules)*

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

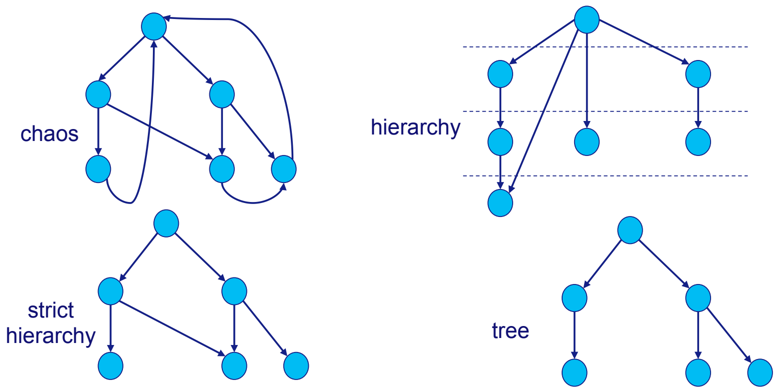
System structure: inter-module complexity

- 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 classes 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

Interesting web page

- http://www.smartdraw.com/resources/tutorials/

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

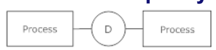
JSD: Jackson Structured Design

- **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**

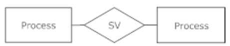
JSD's modeling stage

- **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**

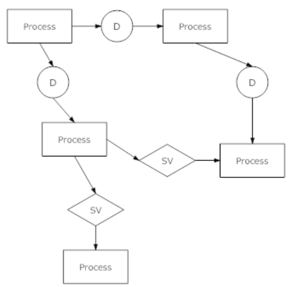
- **DataStreams connect processes and specify what information is passed between them:**



- **State Vectors are an alternative way of connecting processes. They specify the characteristic or state of the entity being changed by a process:**



- **Network diagram:**



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 isnot 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

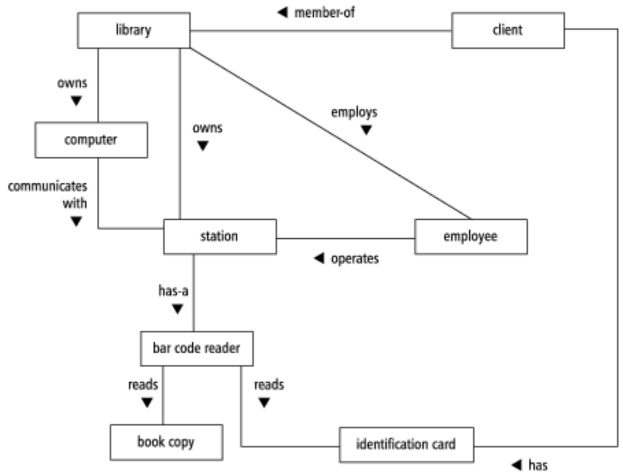
Carefully consider candidate list

- 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"

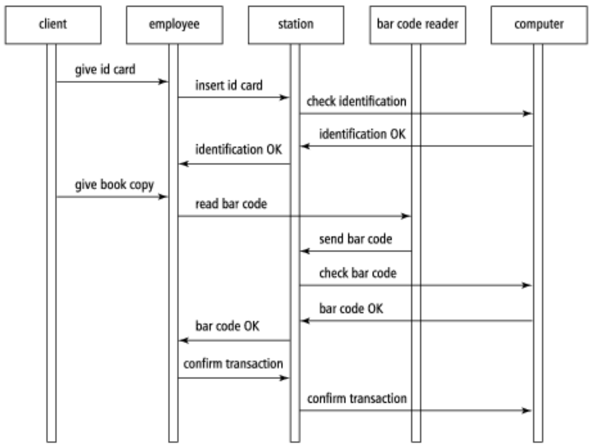
Relationships

- 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



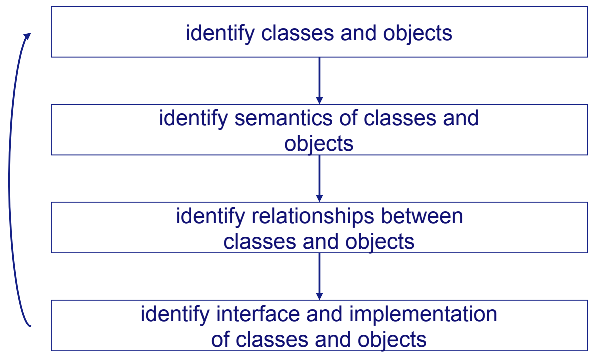
OO as middle-out design

- 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

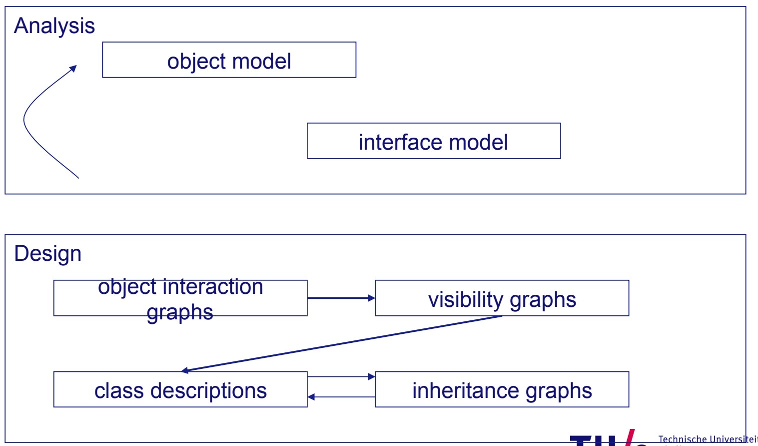
OO design methods

- 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 ⇒ higher complexity ⇒ 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.