Contents

Project Management

- Project Management Activities
- Project Scheduling
- Cost Estimation
- Version- and Configuration Control
- Maintenance

Typical Work-load Distribution

See lecture ...

Not yet available in Powerpoint

Project Management Activities

- Project acquisition
- Project planning
  - Resource assessment
  - Risk and option analysis
- Cost estimation
- Project scheduling
  - Work breakdown structure
  - Effort distribution
  - Resource assignment
- Project tracking and control
- Risk management
- Reporting

Project Resources

- People
  - Required skills
  - Availability
  - Duration of tasks
  - Start date
- Hardware and software tools
  - Description
  - Availability
  - Duration of use
  - Delivery date

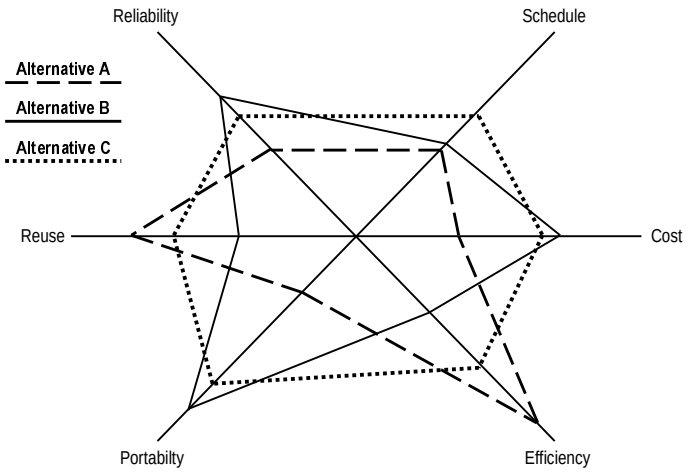
Risks and Option Analysis

- Compare different development alternatives
- Evaluate their risks
- Select best alternative
- ➡ Tools
  - Polar graph
  - Decision tree
  - Forms
  - ...

Top Ten Project Risks

- Staff deficiencies
- Unrealistic schedules and budgets
- Developing the wrong functions
- Developing the wrong interface
- Over-engineering
- Changing requirements
- Externally developed items
- Externally performed tasks
- Performance problems
- Assumptions on technology

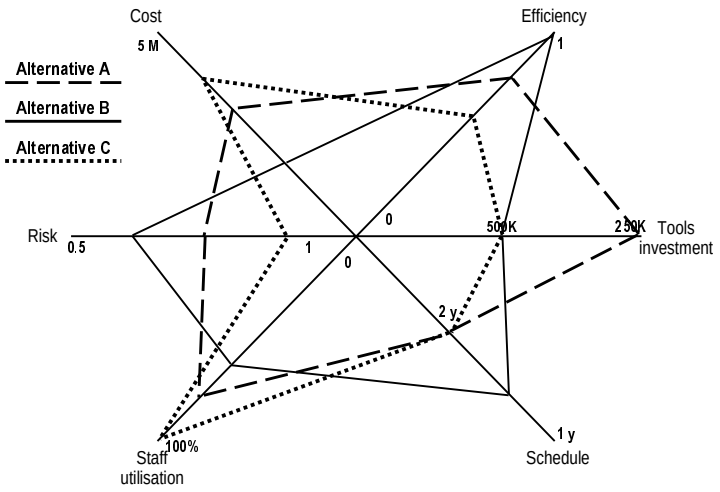
Polar Graphs



Analysis Example

Alternative	Cost (SEK)	Relative efficiency	Tools investment	Schedule (years)	Staff utilization (%)	Risk
A	7.500.000	0.8	250.000	2	85	0.75
B	8.500.000	1	500.000	1.3	70	0.6
C	7.000.000	0.6	500.000	2	100	0.9

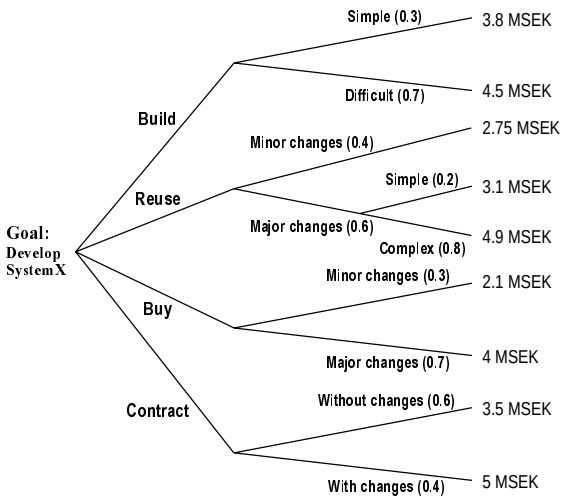
Analysis Example (Polar Graph)



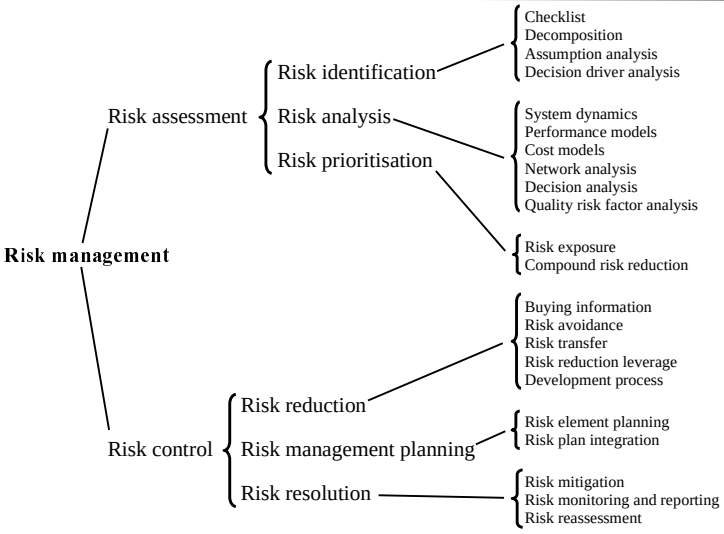
Analysis Example (Forms)

Objectives	**Develop a software components catalogue**
Constraints	Within one year Must support all existing component types Must cost less than 1 MSEK
Alternatives	Buy existing information retrieval (IR) software Buy a database and develop the catalogue using the query language Develop a special-purpose catalogue
Risks	May be impossible within the given constraints Catalogue functionality may be inappropriate
Risk resolution strategy	Develop a prototype to clarify requirements Commission a consultants report on existing IR systems Relax the time constraints
Results	IR systems are too inflexible Identified requirements cannot be met The prototype using a DBMS may be enhanced to to a complete system Special-purpose catalogue development is not cost effective
Plans	Develop the catalogue using the existing DBMS by enhancing the prototype and building a GUI
Commitment	Fund further 12 months of development

Analysis Example (Decision Tree)

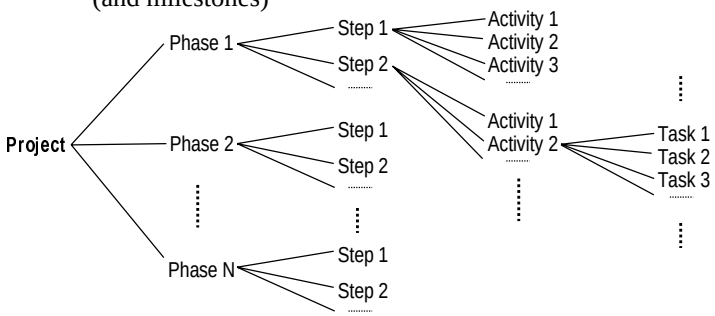


Risk Management



Define Work Breakdown Structure

- *Activity*
  - Major work unit
  - Start date
  - End date
  - Consumes resources
  - Results in work products (and milestones)
- *Project function*
  - Continue throughout the project
  - Not bound to specific phases



Schedule Activities

- Almost all activities depend on the completion of some other activities
- Many activities can be performed in parallel
- Track usage of resources
- ➡ Organisation necessary to balance work-load, costs, and duration
  - PERT chart (activity network/task graph)
  - ➡ Critical path
  - ➡ Project time-line (Gantt chart)
  - ➡ Resource table

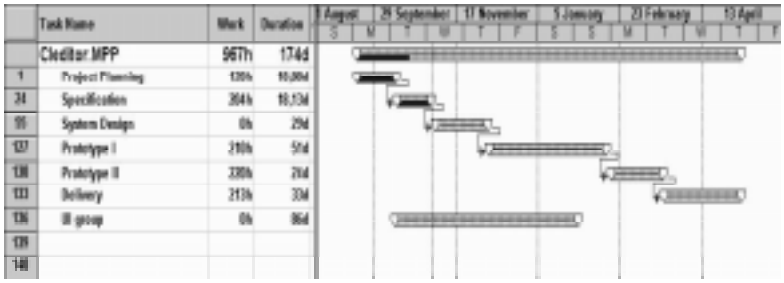
PERT Charts

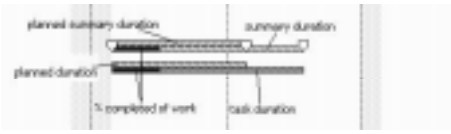
Program **E**valuation and **R**eview **T**echnique

- Graph
  - Nodes = activities/tasks and estimated duration
  - Edges = dependencies
- Compute
  - Slack time = available time - estimated duration
  - Critical path

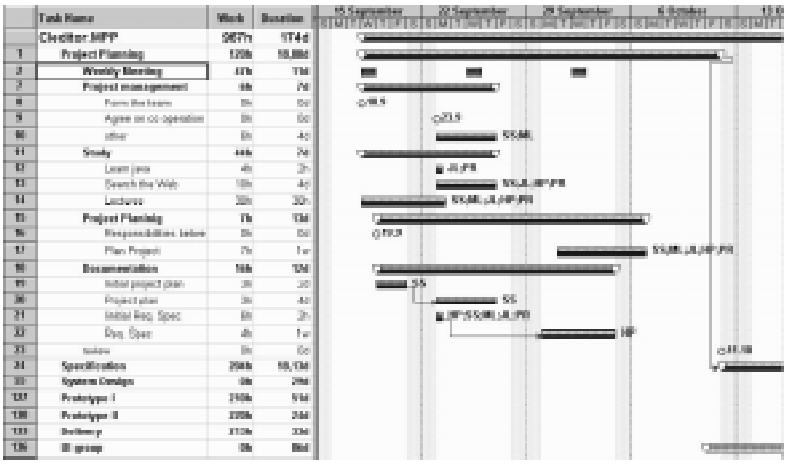
    A path is critical when it contains an activity that, if delayed, will cause a delay of the whole project.

A Gantt Chart (Project Time Line)





Another Gantt Chart



Resource Tables

See lecture ...

Not yet available in Powerpoint

Cost Estimation

- Approach
  - Decompose problem
  - Check for experiences/ data on subproblems
  - Make qualified estimations
  - (Make at least two independent estimates)
- Problems:
  - What are good measures?
  - Do the estimates effect the result?
  - Does the type of software effect the result?
  - Does the project environment effect the result?
  - ...

➡ Use empirical and historical data

➡ Algorithmic cost modelling
  - COCOMO (based on LOC)
  - FP (based on *function points*)

COCOMO

- **C**onstructive **C**ost **M**odeling [Boehm 81]
- Based on publicly available historical data of 63 TRW projects
- Basic assumptions:
  - Requirements change only slightly during the project
  - There is good project management
  - The historical data is representative
  - Assigning more resources to the project does NOT result in linear decreasing development time
- Basic model:
  - $\texttt{Effort} = a \cdot (\texttt{KDSI})^b$

    `KDSI` = Kilo Delivered Source Instructions (≈ LOC - comments)
    The `a` and `b` factors vary depending on the type of project
    `Effort` is measured in `PM` (Person Months = 152h of work)

COCOMO Project Types

- OM: Organic Mode projects
  - Small teams which are familiar with the type of application
  - Development in a familiar environment
- EM: Embedded Mode projects
  - Large and inexperienced teams
  - Many constraints
- SDM: Semi Detached Mode projects
  - Between OM- and EM projects

COCOMO Basic Model

- PM: Person Months
  - = 2.4 $(KDSI)^{1.05}$ for OM projects
  - = 3 $(KDSI)^{1.12}$ for SDM projects
  - = 3.6 $(KDSI)^{1.20}$ for EM projects
- TDEV: Time for DEVelopment
  - = 2.5 $(PM)^{0.38}$ for OM projects
  - = 2.5 $(PM)^{0.35}$ for SDM projects
  - = 2.5 $(PM)^{0.32}$ for EM projects
- N: Number of personnel
  - = PM / TDEV

This is Too Simplistic!?

- There are many *cost drivers* that effect effort
  - Programming language
  - Development methods
  - Tools and environments
  - Experience and capabilities of the development team
  - Available time
  - Requirements volatility
  - ...

COCOMO Intermediate Model

- Takes into account 15 cost drivers, which are ranked on a scale from *very low* to *extra high*
  - Product attributes (e.g. required reliability)
  - Computer system attributes (e.g. time/space constraints)
  - Personnel attributes (e.g. language experience)
  - Project attributes (e.g. tools usage)
- PM: Person Months

  $= 3.2\ (KDSI)^{1.05} \times \Pi C_i$ for OM projects

  $= 3\ (KDSI)^{1.12} \times \Pi C_i$ for SDM projects

  $= 2.8\ (KDSI)^{1.20} \times \Pi C_i$ for EM projects
- $\Pi C_i \in [0.09..9.42]$
- TDEV and N as before

Intermediate COCOMO Summary

- Works quite well in practice
- TRW data is publicly available
- Needs KLOC as input
- Problems:
  - Estimating KLOC in early project stages
  - Comparison of projects using different LOC counts
  - Outdated metrics base (70s)
- Solutions:
  - Cross-check using an other estimation technique
  - Standardised LOC counts
  - Continuos model calibration

COCOMO II

- Recent new version of COCOMO
- Three stage estimation
  - Stage 1: Application Composition
    - Estimation base: Object points
    - Single standard project type
    - No cost drivers
  - Stage 2: Early Design
    - Estimation base: Function points
    - Six project type factors
    - Few cost drivers (6)
  - Stage 3: Postarchitecture
    - Estimation base: Function points or KLOC
    - Six project type factors
    - Cost drivers (16) similar to original COCOMO intermediate model

Function (Feature) Points

- Estimate functionality captured in requirements

# User inputs	x	(3,4,6)	=	
# User outputs	x	(4,5,7)	=	
# User inquiries	x	(3,4,6)	=	
# Files	x	(7,10,15)	=	
# External interfaces	x	(5,7,10)	=	
(# Algorithms	x	(3,4,6)	=	) ← Feature points only

Count-total →

$$FP = \text{Count-total} \times [\,0.65 + 0.01 \times \Sigma F_i\,]$$

Adjustment factors
($F_i \in \{0,\ldots,5\}$; $i = 1..14$)

Cost Estimation Results

"Today, a software cost estimation model is doing well if it can estimate development costs within 20% of actual costs, 70% of the time, and on its own turf (that is, within the class of projects to which it has been calibrated) ....This is not as precise as we might like, but it is accurate enough to provide a good deal of help in software engineering economic analysis and decision making."

[Boehm 81]

The Project Plan

1 Introduction
- 1.1 Project overview
- 1.2 Project deliverables
- 1.3 Evolution of the SPMP
- 1.4 Reference Materials
- 1.5 Definitions and acronyms

2 Project Organisation
- 2.1 Process model
- 2.2 Organisational structure
- 2.3 Organisational boundaries and interfaces
- 2.4 Project responsibilities

3 Managerial Process
- 3.1 Management objectives and priorities
- 3.2 Assumptions, dependencies and constraints
- 3.3. Risk management
- 3.4 Monitoring and controlling mechanisms
- 3.5 Staffing plan

4 Technical Process
- 4.1 Methods, tools and techniques
- 4.2 Software documentation
- 4.3 Project support functions

5 Work Packages, Schedule, and Budget
- 5.1 Work packages
- 5.2 Dependencies
- 5.3 Resource requirements
- 5.4 Budget and resource allocation
- 5.5 Schedule

According to ESA PSS-05-0 (see [ESA 96])

Version and Configuration Control

- Systems change over time
- ➡ Different versions over time
- Systems are used
  - ... in different environments
  - ... for different purposes
  - ... by different kinds of users
  - ... together with various other systems
- ➡ Different versions at the same time
- ➡ Different sets of consistent versions (configurations)

Versioning Problems

- Double Maintenance Problem
- Shared Data Problem
- Simultaneous Update Problem

Storing Versions

- Naive: Separate files for each version
- Version handling by numbering schemes
- ➨ Double Maintenance Problem
- Solution: One original version plus *deltas*
  - Forward deltas
  - Backward deltas
  - Forward and backward deltas
- ➨ Shared Data Problem
- ➨ Simultaneous Update Problem
- Solution: Check-in/check-out mechanism
- Still a problem: Merging versions

Merging Versions

See lecture for an example ...

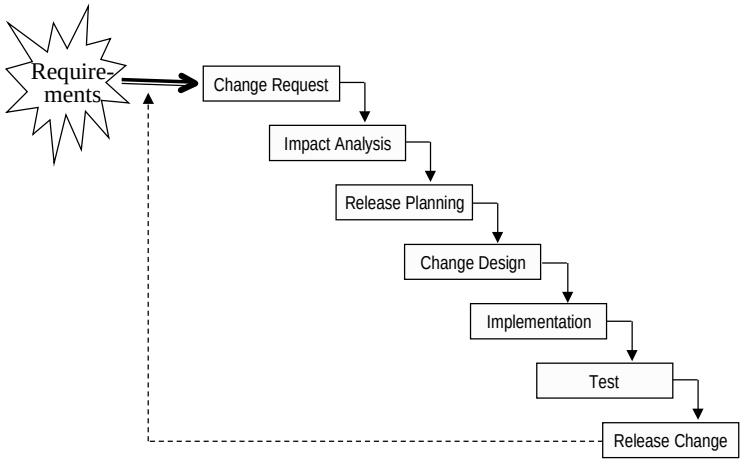
Not yet available in Powerpoint

Tools for Version and Configuration Control

- General:
  - History- and log-files
  - Hierarchical file systems
  - ...
  - File comparators
  - Patch generators
- Version Control:
  - Modification tracking
  - Control of development branches
  - Efficient storage and retrieval
  - Access control
  - Merging versions
  - ...
- ➨ SCCS, RCS, CVS, ...
- Configuration control:
  - Dependency management and control
  - System creation
  - Integration with version control
  - ...
- ➨ Make, makefile generators, ...

Maintenance

The most expensive part of the software lifecycle!



Change Request Forms

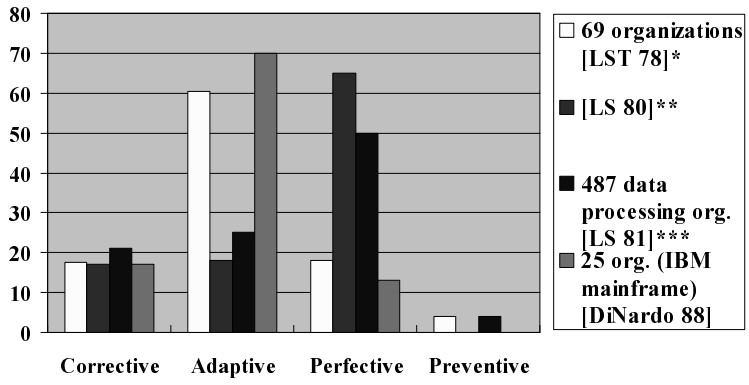
See lecture for an example ...

Not yet available in Powerpoint

Maintenance

- Kinds of maintenance
  - Corrective
  - Adaptive
  - Perfective
  - Preventive
- Promoters of maintenance
  - Modular or oo design
  - Clear interfaces
  - Readable code
  - Good documentation
  - ...

Types of Maintenance

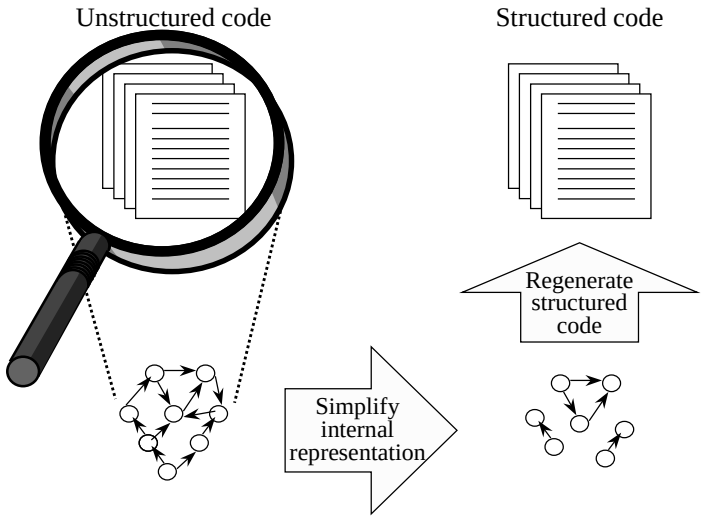


*See [Schach 97].
**See [Somm 96].
***See [Pfleeger 98].

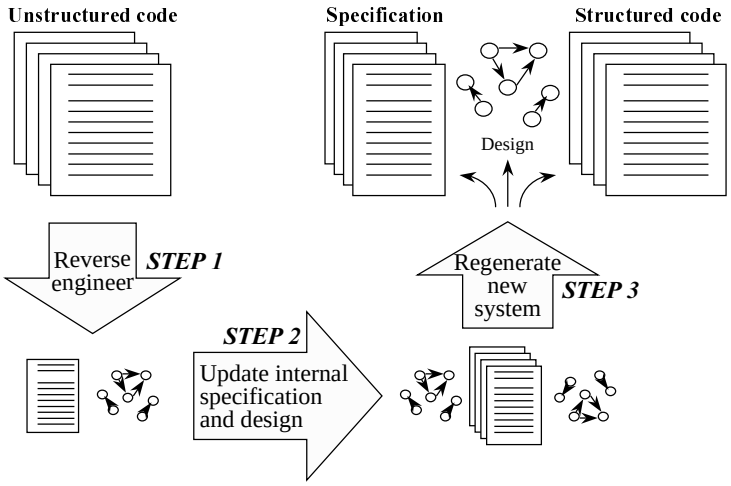
Maintenance Buzzwords

- Software restructuring
  - Code reorganisation
- Design recovery
  - Reconstruct the design from existing code
- Reengineering/ reverse engineering
  - Restore and enhance missing documents

Reverse Engineering



Reengineering



Contents

- Introduction ✔
- Requirements Engineering ✔
- UI Design ✔
- Project Management ✔
- **Software Design**
- Detailed Design and Coding
- Quality Assurance

Software Design

- Introduction
- Modularization and Metrics
- Classical Design Approaches
- (Module Interconnection Languages)

Design Activities

Transform the logical model (➔ RE) into a physical model, in sufficient detail to permit its physical realization.

- Architectural design
  - ❑ Identify the systems components
  - ❑ Structure the system components
  - ❑ Assign functionality to components
  - ❑ Assign data to components
  - ❑ Plan for future changes
  - ➨ Define the structure of the implementation
- Detailed design
  - ❑ Refine the (architectural) components
  - ❑ Choose specific data structures
  - ❑ Choose specific algorithms
  - ➨ Define the logic of the implementation

Design Principles

- Abstraction
- Encapsulation
- Information Hiding
- Structuring
- ➨ Modularization

Characterisation of Modules

- Logic entities which fulfil certain tasks
- Simple entities, i.e. their tasks can be described clearly and briefly
- Units containing data and/or operations
- Provide resources usable by other modules
- Their realisation is encapsulated
- May use resources from other modules

Modules ...

- ... are replaceable
- ... are free of side-effects
- ... can be tested separately
- ... can be compiled separately
- ... can be developed independently

➡ Modules are abstract/virtual machines

Advantages of Modular Systems

- Easier to understand
  - Only few modules must be studied
- Easier to develop and to test
  - Independence
- More portable
  - System dependencies reside in a few dedicated modules
- Easier to maintain
  - Changes can be traced to few modules
- Easier to reuse
  - Clear dependencies

Measuring Module Quality

Reliable and early data with significant impact on quality.

- Classical metrics:
  - LOC
  - Cyclomatic number (McCabe)
  - Control variable complexity (McClure)
  - Software science (Halstead)
  - ➨ “Wrong” understanding of module
  - ➨ Late applicability
- More useful:
  - Coupling
  - Cohesion
  - Fan-in/fan-out
  - Graph-oriented metrics
  - Weighted methods per class
  - Depth/width of inheritance trees
  - ...

Coupling

- Measures the degree of independence between different modules
  - Content coupling
  - Common coupling
  - (External coupling)
  - Stamp coupling
  - Data coupling

bad ↓ good

➡ Each module should communicate with as few as possible other module

➡ Communicating modules should exchange as few as possible data

➡ All communication must be explicit

How to Uncouple Modules

- Data coupling
  - Exchange only necessary information
  - Do not pass data through several modules
- Stamp coupling
  - Do not encapsulate unrelated data
- Control coupling
  - Limit control information in interfaces
- Common/External coupling
  - Pass data explicitly as parameters
  - Divide complex data into independent parts that can be exclusively used of different modules
  - Hide data
- ~~Content coupling~~ USCH!

Cohesion

- Measures “relatedness” of the resources encapsulated in one module
  - Coincidental cohesion
  - Logical cohesion
  - Temporal cohesion
  - Procedural cohesion
  - Communicational cohesion
  - Sequential cohesion
  - Functional/informational cohesion

bad ↓ good

➡ Each element in a module should be a necessary and essential part of one and only task

Coupling / Cohesion Summary

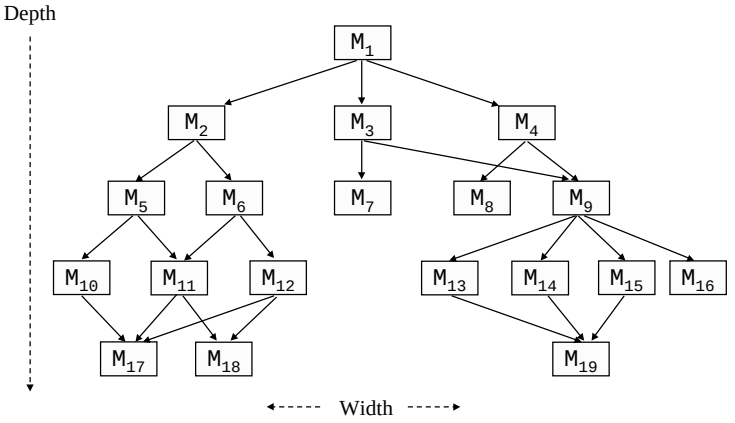
- The modules of a system should be highly cohesive and loosely coupled

➡ Good modularization

Problem:

How can coupling or cohesion be measured?

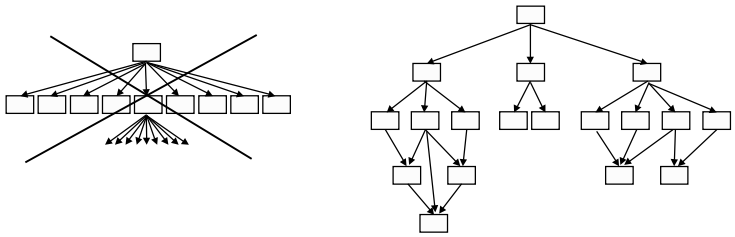
Fan-in vs Fan-out



M_9: Fan-in = 2, fan-out = 4

Fan-in vs Fan-out Rules

- Minimise structures with high fan-out
- Strive for fan-in as depth increases



Structured Design

- Evolved from top-down design, modularity, and structured programming
  - Stevens, Myers, Constantine (74)
  - Yourdon, Constantine (79)
  - Page-Jones (80)
- Systematic development of a design (*structure chart*) from a DFD
  - Transform analysis
  - Transaction analysis

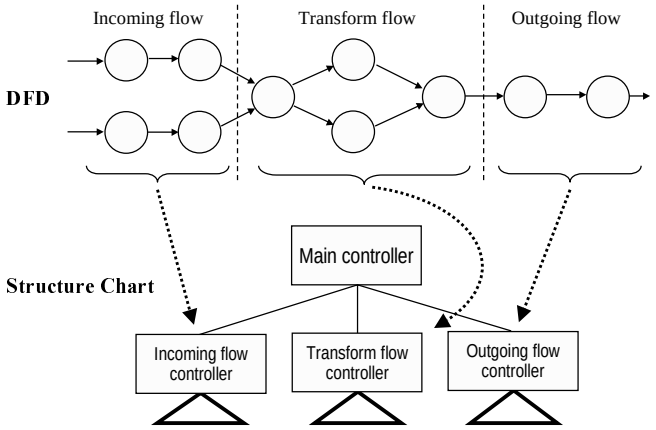
Structure Charts

See lecture for an example ...

Not yet available in Powerpoint

Transform Analysis

- One or more inputs are transformed into one or more outputs ("and" semantics)



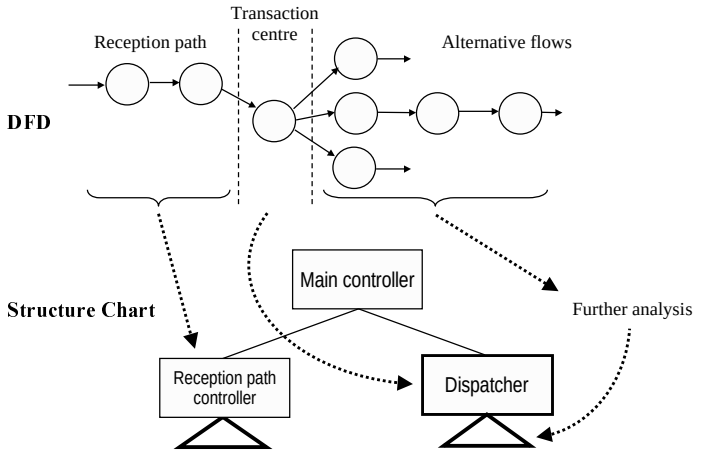
Transform Analysis Example

See lecture for an example ...

Not yet available in Powerpoint

Transaction Analysis

- Dataflow splits into alternatives (“or” semantics)



Transaction Analysis Example

See lecture for an example ...

Not yet available in Powerpoint

Structured Design Summary

- Systematic approach to derive a design from the analysis results
- DFDs as input
- Transform- and transaction analysis
- Factoring and refinement (rules exist)

✚ Good support for functional decomposition	– No support for data abstraction
✚ Systematic approach	– Data spread over the whole system
✚ Incorporates metrics	– Only sequential systems
✚ Uses design heuristics	– Metrics and heuristics are mainly of syntactic nature
✚ Tool support	– No design “decisions”

The (Architectural) Design Document

Service Information
- a Abstract
- b TOC
- c Document status and history

1 Introduction
- 1.1 Purpose
- 1.2 Scope
- 1.3 Glossary
- 1.4 References
- 1.5 Overview

2 System Overview

3 System Context
- 3.i External interface i

4 System Design
- 4.1 Design method
- 4.2 Decomposition description *(views)*

5 Component description
- 5.i Component i
  - 5.i.1 Type
  - 5.i.2 Purpose
  - 5.i.3 Function
  - 5.i.4 Subordinates
  - 5.i.5 Dependencies
  - 5.i.6 Interfaces
  - 5.i.7 Resources
  - 5.i.8 References
  - 5.i.9 Processing
  - 5.i.10 Data

6 Feasibility and Resource Estimates

7 Software Requirements Vs. Components Traceability Matrix

Slightly adapted from ESA's Software Engineering Standards PSS-05-0 (see [ESA 96])

A Traceability Matrix

- Relates requirements to design artefacts
- ➡ Shows dependencies
- ➡ Supports change management

	Module 1	Module 2	Module 3	Module 4	Module ...
Requ 1	X	X		X	
Requ 2	X		X		X
Requ 3				X	
Requ 4	X				
Requ ...				X	

- ➡ Useful for other traceability purposes

Contents

➪Detailed Design and Coding

- ➪Detailed Design Activities
- ➪Approaches to Detailed Design
- ➪Coding Style and Guidelines

Detailed Design Activities

Give sufficient information, so that the implementation teams can do a good job.

- Choose specific data structures and algorithms
- Refine the components from architectural design
- Define HOW
- Comments are NOT enough:

```
procedure replaceText( var text: TextFile; oldWords, newWords: WordList);
(* Replace in the text text all occurrences of the i-th word in oldWords by      *)
(* the i-th word in newWords; oldWords and newWords must have the same  *)
(* length                                                                        *)
```

Open Questions

- What are the word delimiters?
  - blank, EOL, EOF, TAB
  - `.´, `,´, `;´, `:´, ...
  - `_´, `&´, ...
- Is the matching case sensitive?
- Must replacements have the same length?
- How to solve conflicts?
  - Several different replacements for the same old word
  - Some words in *newWords* appear also in *oldWords*
  - Assume the following:

    text: . . . ABC . . . ; *oldWords*: AB, BC; *newWords*: X, Y
    alternative1: . . . XC . . .
    alternative2: . . . AY . . .

Approaches to Detailed Design

- Informal
  - Structured Egnlish
- Semi-formal
  - Program Design Languages (PDLs)
  - Diagrammatical techniques
- Formal
  - Formal Specifications (e.g. Z, VDM, ...)
  - Pre-/postconditions & invariants (sometimes called *programming by contracting*)

Programming by Contracting

Clients and servers of services "sign" contracts, i.e. servers guarantee the effects of their services offered, if and only if clients use these services correctly.

function getPosition(a: **array of** Element; el: Element) **return integer**;
(* Returns the relative position of *el* in *a* *)
precondition ∃ i ∈ [a'First..a'Last]: a[i] = el (* such an element exists *)
postcondition a[getPosition(a, el)] = el **and** a = a.old
(* *getPosition* really returns the position of *el* in *a* and *a* is unchanged *)

You could even specify that the array must be sorted in ascending order to allow for a faster algorithm by adding the following to the precondition:

and ∀ i,j ∈ [a'First..a'Last]: i < j ⇒ a[i] < a[j]

The Detailed Design Document

Service Information
- a Abstract
- b TOC
- c Document status and history

PART 1—General Description
- **1 Introduction**
  - 1.1 Purpose
  - 1.2 Scope
  - 1.3 Glossary
  - 1.4 References
  - 1.5 Overview
- **2 Project Standards, Conventions and Procedures**
  - 2.1 Design standards
  - 2.2 Documentation standards
  - 2.3 Naming conventions
  - 2.4 Programming standards
  - 2.5 Software development tools

PART 2—Component Design Specifications
- **I Component i *(its name)***
  - I.1 Type
  - I.2 Purpose
  - I.3 Function
  - I.4 Subordinates
  - I.5 Dependencies
  - I.6 Interfaces
  - I.7 Resources
  - I.8 References
  - I.9 Processing
  - I.10 Data

Appendix A: Source Code Listings

Appendix B: Software Requirements Vs. Components Traceability Matrix

Slightly adapted from ESA's Software Engineering Standards PSS-05-0 (see [ESA 96])

Implementation

- Transform the detailed design into concrete programming language code
- Ensure that this code correctly implements the detailed design

OOPS! Many modern programming languages contain detailed design elements, e.g. Eiffel

Programming Style

- Strive for simplicity and clarity
- Use significant names and consistent typing
- Describe each component through a prologue
  - General functionality
  - Interface
  - All important data and restrictions
  - History
- Commit to effective coding and commenting guidelines
- Use simple statement constructions and program layout
- Encode input and output to simplify data transfer and error recovery
- Strive for efficient code, but not at the cost of readability and simplicity (Jackson´s optimisation rules)
  - Don´t do it
  - For experts: Don´t do it now, first produce a complete, correct, and clear non-optimised version

Programming Guidelines

- Use separate files for each module, class, macro, inline, ... definition
- Use separate files for the definition/specification and implementation when possible
- Call operations only when all preconditions are satisfied (this is the caller´s responsibility)
- Separate policy and implementation (e.g. the scaling itself and setting the scaling factor)
- Don´t use modes (provide separate operations)
- Don´t (over-) use typecasting
- Avoid pointers to pointers
- Commit to effective naming conventions

Contents

➪Quality Assurance

- ➪Introduction
- ➪Testing Phases and Approaches
- ➪Black-box Testing
- ➪White-box Testing

What is Quality Assurance?

QA is the combination of planned and unplanned activities to ensure the fulfillment of predefined quality standards.

- Constructive vs analytic approaches to QA
- Qualitative vs quantitative quality standards
- Measurement
  - ❑ Derive qualitative factors from measurable quantitative factors
  - ➨ Software Metrics

Approaches to QA

- Constructive Approaches

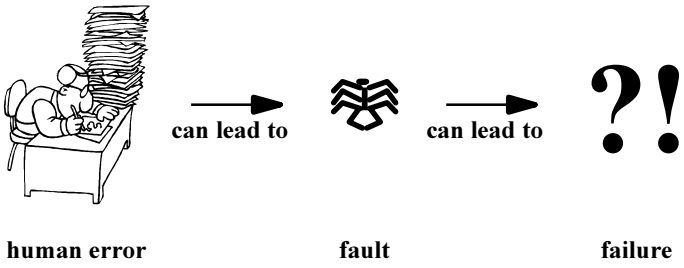
  Usage of methods, languages, and tools that ensure the fulfillment of some quality factors.

  - Syntax-directed editors
  - Type systems
  - Transformational programming
  - Coding guidelines
  - ...
- Analytic approaches

  Usage of methods, languages, and tools to observe the current quality level.

  - Inspections
  - Static analysis tools (e.g. *lint*)
  - Testing
  - ...

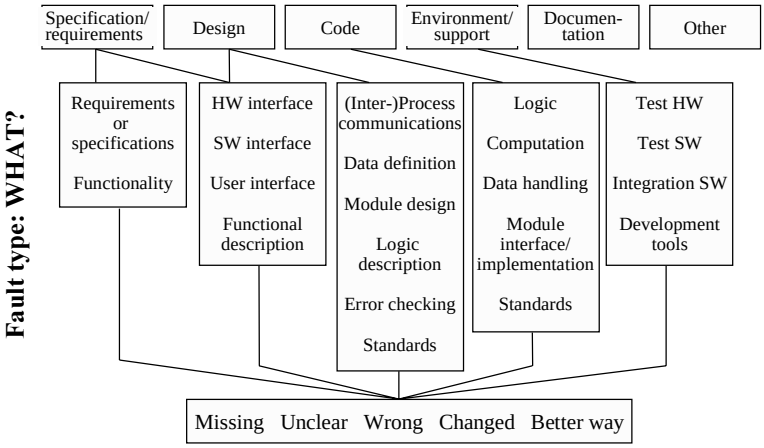
Fault vs Failure



- Different types of faults
  - ➨ Different identification techniques
  - ➨ Different testing techniques
- ➨ Fault prevention and -detection strategies should be based on expected fault profile

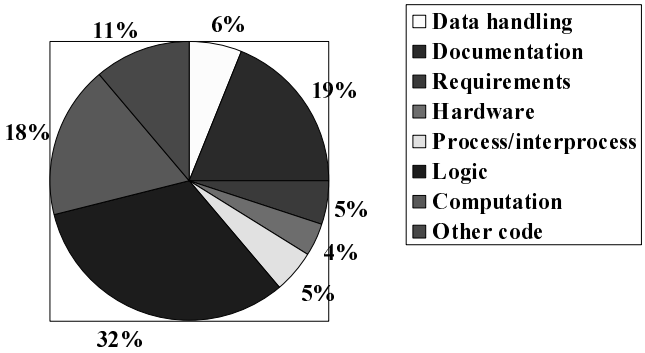
HP´s Fault Classification

Fault origin: WHERE?



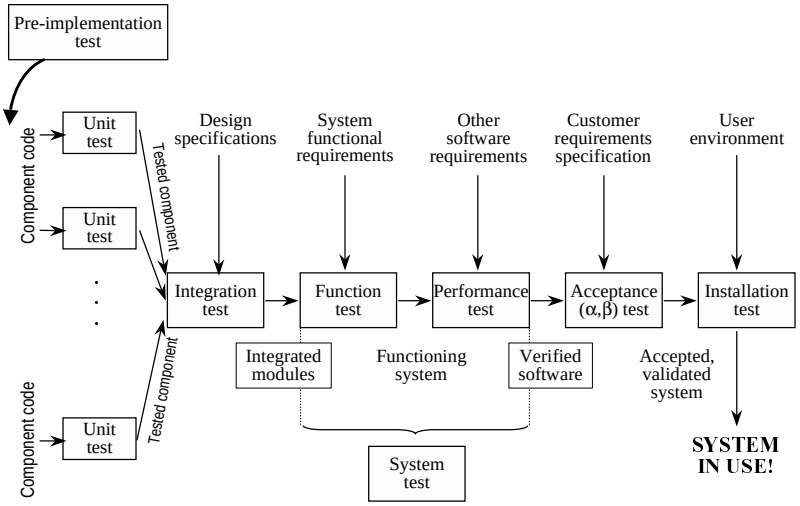
Fault mode: WHY?

Fault Profile of a HP Division



See [Pfleeger 98].

Testing Phases



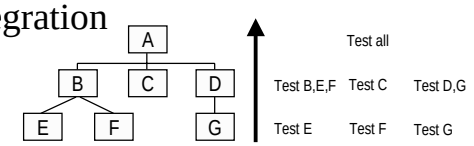
Pre-Implementation Testing

- Inspections
  - See guest lecture
- Walkthrough
  - In teams
  - Examine source code/detailed design
- Reviews
  - More informal
  - Often done by document owners

- Advantages
  - Effective
  - High learning effect
  - Distributing system knowledge
- Disadvantages
  - Expensive

Integration Testing

- Different strategies affect
  - Design strategy
  - Time to first working prototype
  - Amount of parallelism
  - Additional work for test drivers/-stubs
- Bottom-up integration



- Top-down integration
- Big-bang integration
- Sandwich integration (combined td/bu)

Testing vs “Proofing” Correctness

- Verification
  - Check the design/code against the requirements
  - ➡ Are we building the product right?
- Validation
  - Check the product against the expectations of the customer
  - ➡ Are we building the right product?
- Testing

  Testing is the process in which a (probably unfinished) program is executed with the goal to find errors. [Myers 76]

  Testing can only show the presence of errors, never their absence. [Dijkstra 6?]

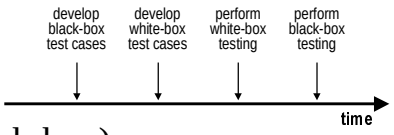
  - ➡ Testing can neither proof that a program is error free, nor that it is correct

Testing Principles

- Construction of test suites
  - Plan tests under the assumption to find errors
  - Try typical and untypical inputs
  - Build classes of inputs and choose representatives of each class
- Carrying out tests
  - Testers ≠ implementers
  - Define the expected results before running a test
  - Check for superfluous computation
  - Check test results thoroughly
  - Document test thoroughly
- Simplify test
  - Divide programs in separately testable units
  - Develop programs test friendly
- Each test must be reproducible

Test Methods

- Structural testing (white-box, glass-box)
  - Uses code/detailed design is to develop test cases
  - Typically used in unit testing
  - Approaches:
    - Coverage-based testing
    - Symbolic execution
    - Data flow analysis
    - ...



- Functional testing (black-box)
  - Uses function specifications to develop test cases
  - Typically used in system testing
  - Approaches:
    - Equivalence partitioning
    - Border case analysis
    - ...

Test Preparation

- Exhaustive testing is prohibited, because of the combinatorial explosion of test cases
- ➡ Choose representative test data

```
for i := 1 to 100 do
   if a = b then
      X
   else
      Y;
```

i	paths to test	#tests
1	X, Y	2
2	XX, XY, YX, YY	4
3	XXX, XXY, ...	8
	...	
100		2^{100}

$2 * 2^{100} - 2 > 2{,}5 * 10^{30}$

- ➡ With 10^6 tests/sec this would take $8*10^{16}$ years
- ➡ Choose test data (*test cases*)

How to Choose Test Data

- Example 1

```
if ((x + y + z)/3 = x) then
   writeln( "x, y, z are equal")
else
   writeln( "x, y, z are unequal");
```

Test case 1: x=1, y=2, z=3
Test case 2: x=y=z=2

- ➡ Both paths must be tested!

- Example 2

```
if (d = 0) then
   writeln( "division by zero")
else
   x = y/n;
(*-----------------------------*)
x = y/n;
```

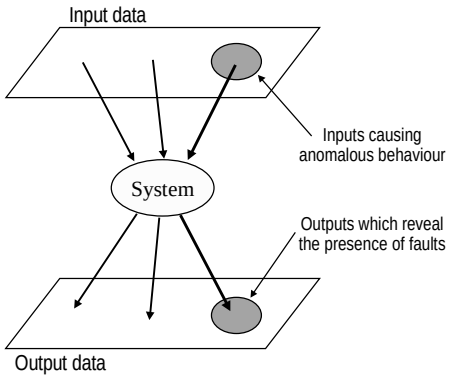
- ➡ How can I know there is a "path"?

Test Case Development

- Problems:
  - Systematic way to develop test cases
  - Find a satisfying set of test cases
- Test case ≠ test data
- Test data: Inputs devised to test the system
- Test case:
  - Situation to test
  - Inputs to test this situation
  - Expected outputs
  - ➨ Test are reproducible

➨ Equivalence partitioning
➨ Coverage-based testing

Equivalence Partitioning



Input- and output data can be grouped into classes where all members in the class behave in a comparable way.

➨ Define the classes
➨ Choose representatives
- Typical element
- Borderline cases

```
x ∈ [25 .. 100]  → class 1: x < 25
                 → class 2: x >= 25 and x <= 100
                 → class 3: x > 100
```

Equivalence Partitioning Example

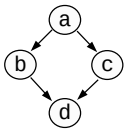
See lecture ...

Not yet available in Powerpoint

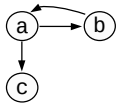
Coverage-based Testing

- Derive test cases from the structure of the code
  - Build the flow graph of the code
  - Cover the graph with tests as densely as possible
- Flow graphs:

```
if a then
   b
else
   c;
d
```



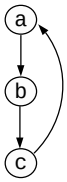

```
while a do
   b;
c
```



Statement Coverage

- Every statement is at least executed once in some test

```
if a then
   b;
c
```



➡ With `a=true` all statements are executed, but `a=false` is never tested!

Branch Coverage

- For every decision point in the graph, each branch is at least chosen once

```
if (X or not (Y and Z) and ... then
                  = a
   b;
c
```

➡ With `a=true` and `a=false` all paths are executed, but all combinations of conditions are never tested!

Condition Coverage

- Test all combinations of conditions in boolean expressions at least once

```
if (X or not (Y and Z) and ... then
   b;
c := (d + e * f - g) div op( h, i, j);
```

➡ Why in boolean expressions only?

Expression Coverage

- Each expression must take so many values that it cannot be replaced by a simpler one where the test still produces the same results

```
c := d + e - f;
```

test with `e=f` ⇒ could be simplified to `c := d;`
test with `d=f` ⇒ could be simplified to `c := e;`

➡ Choose `d≠e≠f`

➡ Only feasible with tool support

Coverage-based Testing

- Advantages
  - Systematic way to develop test cases
  - Simple model of underlying program
  - Measurable results (the coverage)
  - Extensive tool support
    - Flow graph generators
    - Test data generators
    - Bookkeeping
    - Documentation support
- Disadvantages
  - Code must be available
  - Does not (yet) work well for data-driven programs

Branch Coverage Example

See lecture ...

Not yet available in Powerpoint

Further Testing Techniques

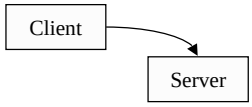
- Data flow analysis
- Symbolic execution
- Mutation analysis
- Regression testing

Testing Tools / Support

- Test data generators
  - Input: Program + testing strategy
  - Output: Sets of input data
- Profilers
  - Instrument code to collect run-time data
    - Time spent in operations
    - Number of calls to operations
    - Number of loop iterations
    - ...
    - ➨ Find bottle-necks
    - ➨ Indicate dead code
- Simulators
  - Common in hard-/software systems and/or real-time systems
  - Emulate critical parts by software

Testing Tools / Support

- Debuggers
  - Manual code instrumentation
  - Inspect/trace variables
  - ...
- File comparators
  - E.g. for regression testing
- Test-stub/-driver generators
  - Simulate client or server components, which are not yet available



References

[Boehm 81] B.W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981. "Classical."

[BuRa 70] J.N. Buxton, B. Randell, *Proceedings of the 1969 NATO Conference on Software Engineering*, NATO Science Committee, 1970. "Historical."

[ESA 96] C. Mazza, J. Fairclough, B. Melton, D. de Pablo, A. Scheffer, R. Stevens, M. Jones, G. Alvisi, *Software Engineering Guides*, Prentice Hall, 1996. "Guide to ESA Standards."

[GoRu 95] A. Goldberg, K.S. Rubin, *Succeeding with Objects*, Addison-Wesley, 1995. Object-Oriented Software Engineering.

[Hump 95] W.S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, 1995. Main PSP textbook.

[Myers 79] G.J. Myers, *The Art of Software Testing*, Wiley, 1979. "Classical."

[Pfleeger 98] S.L. Pfleeger, *Software Engineering, Theory and Practice*, Prentice Hall, 1998. Course textbook.

[Schach 97] S.R. Schach, *Software Engineering with Java*, Irwin, 1997.

[Somm 96] I. Sommerville: *Software Engineering*, Addison-Wesley, 1996.

[Yourdon 92] E. Yourdon, *Decline and Fall of the American Programmer*, Prentice Hall, 1992. Critical Software Engineering textbook.