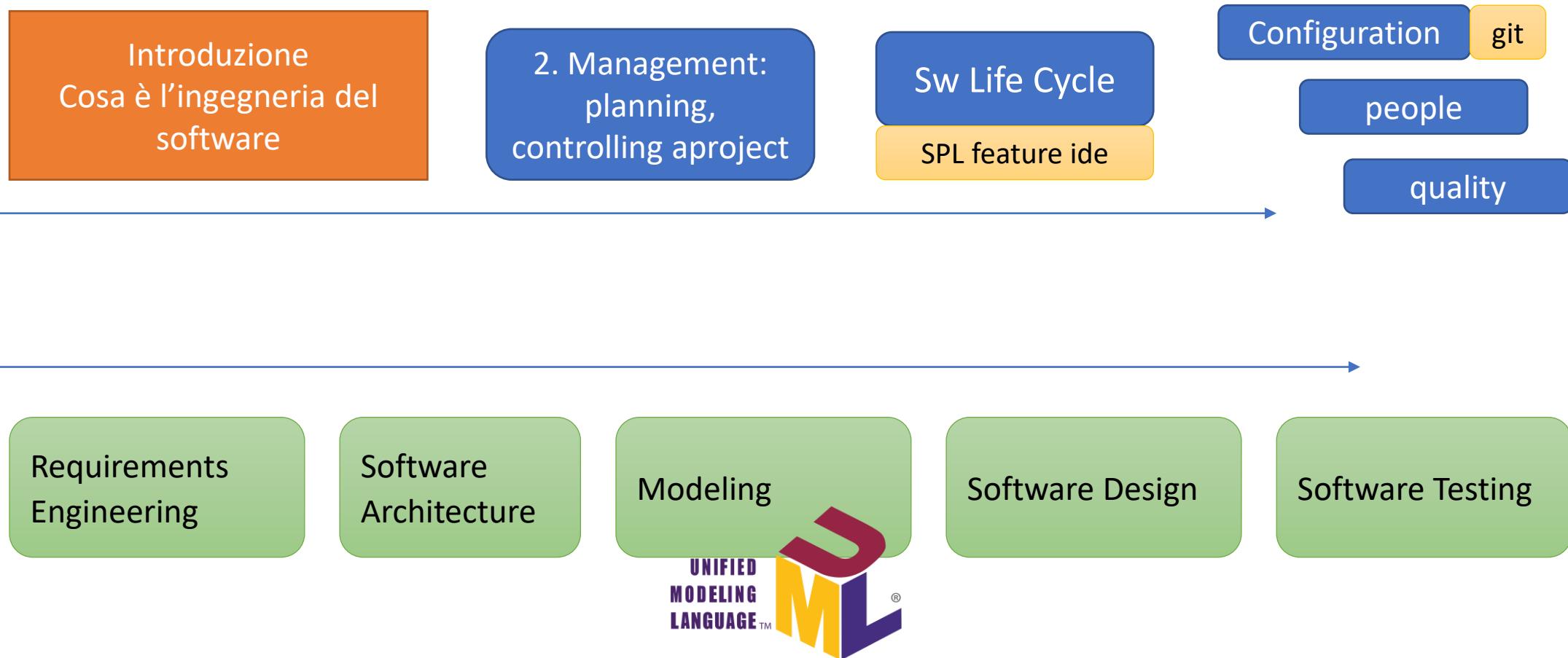


# Ingegneria del software

AA 2021/2022

Angelo Gargantini

# Contenuto del corso

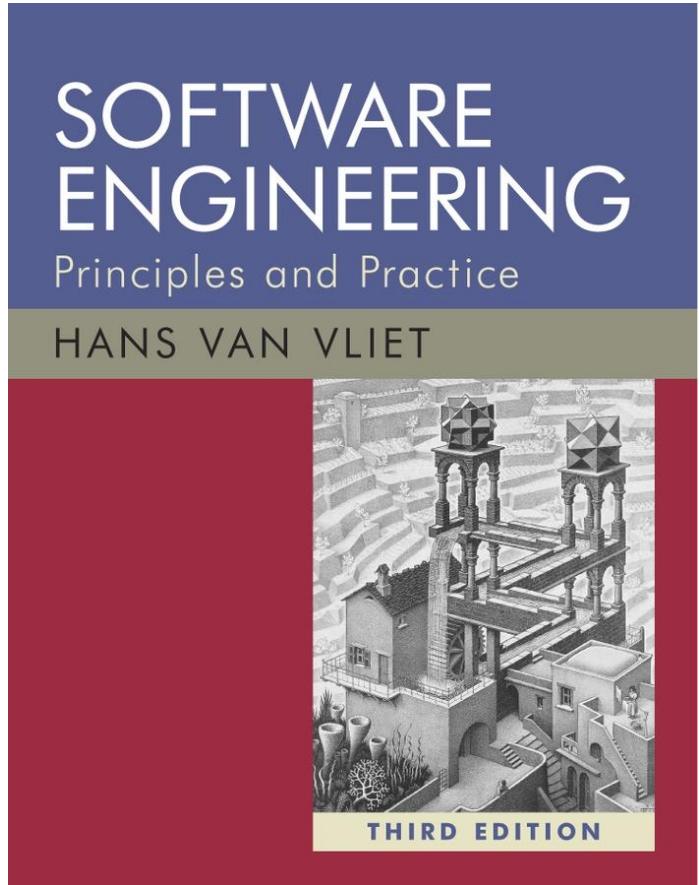


# Professori

- Angelo Gargantini
- Silvia Bonfanti (UML – 16 ore)
- Patrizia Scandurra (servizi 8 ore)

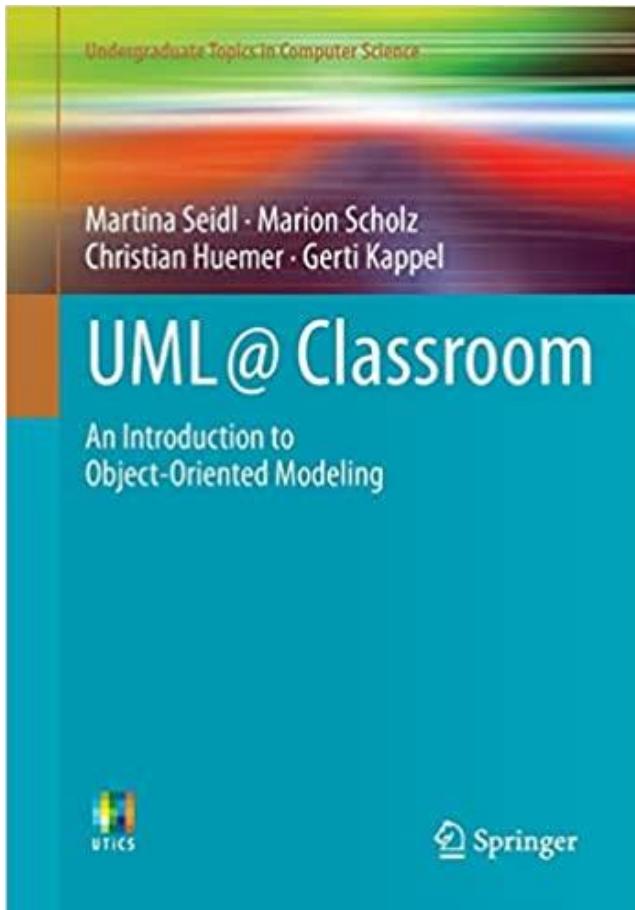


# Libro di testo



- **Software Engineering: Principles and Practice**
- **Hans Van Vliet**
- Peso articolo : 1.35 kg
-

# Per UML



- **UML @ Classroom: An Introduction to Object-Oriented Modeling**
- **Tool per UML: ancora da decider STARUML?**

# Lezioni e laboratorio

- Lunedì 8.30 -> 10.30 LEZIONE (8.45-> 10.15)
- Giovedì 14 -> 16 LEZIONE
- Venerdì 10.30 -> 12.30 : laboratorio (UML)
  
- Stiamo lavorando al kahoot

# Esame

1. Scritto (3h)
  - Domande teoriche alcune aperte altre chiuse
  - Piccoli esercizi
  - 28 punti max
2. Orale breve – nessun punteggio (più o meno 1)
3. Progetto da portare obbligatorio 4 punti
  - Si può fare in gruppo max 3
  - max 10 (1) 15(2) 20 (3) minuti per la presentazione
  - Dettagli a seguire

# **Software Engineering**

## **Capitolo 1: Introduzione**

**Angelo Gargantini 2021**

# From programming to ?

- **Beginning of computer science: small programs**
- **Present-day applications are rather different in many respects.**
  - very large and
  - are developed by teams that collaborate
  - over periods spanning several years.
- **programming art or craft? Arte o mestiere?**

# Software engineering

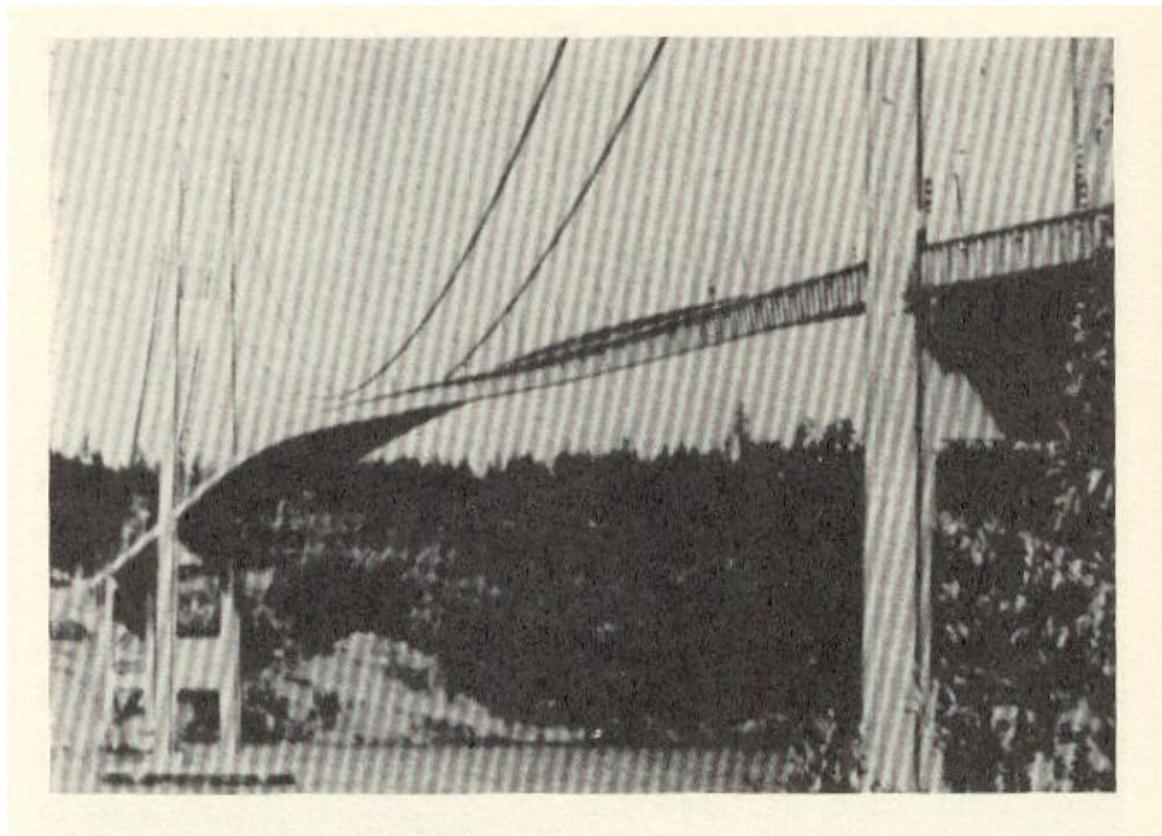
## The beginning

- **1968/69 NATO conferences: introduction of the term Software Engineering**
- **Idea: software development is not an art, or a bag of tricks**
- **Build software like we build bridges**
  - starting from a theoretical basis and using sound and proven design and construction techniques, as in other engineering fields?

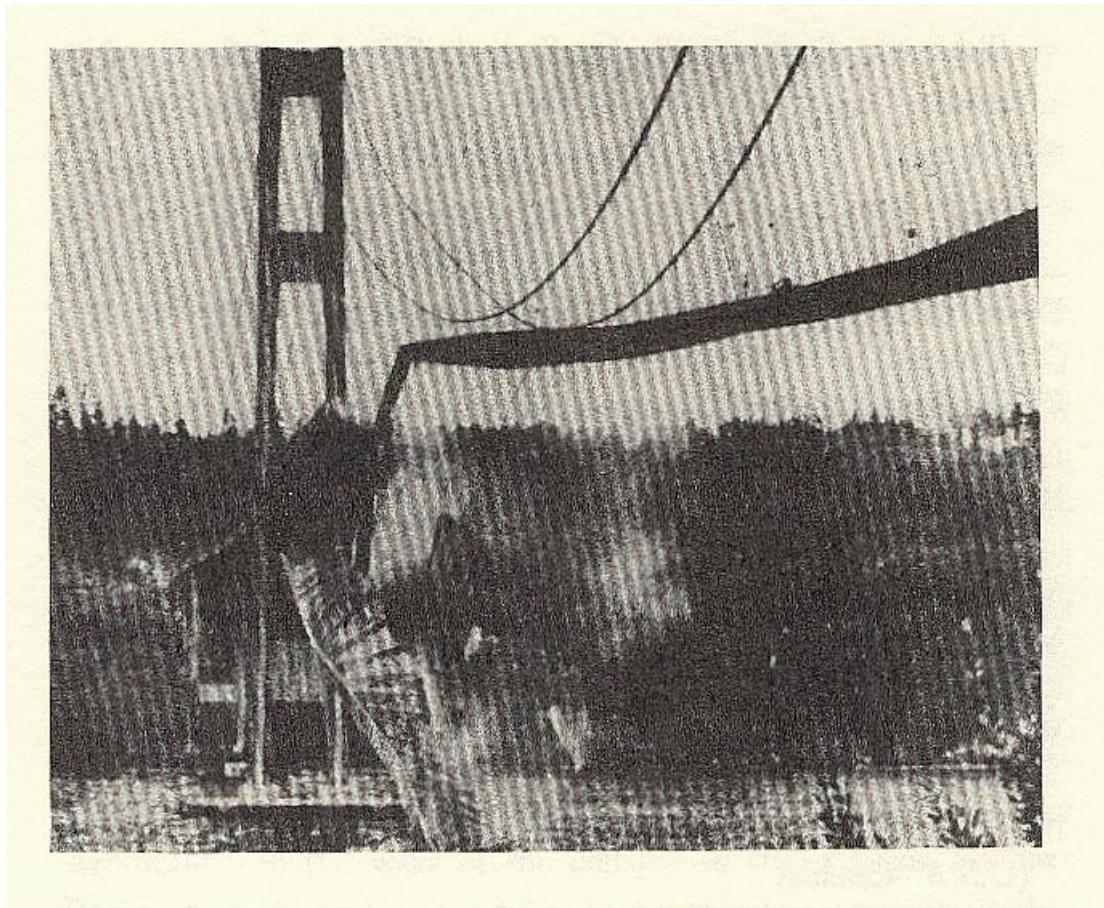
# **Current status**

- **a lot has been achieved**
- **but ...**
- **some of our bridges still collapse**

# Tacoma Narrows bridge



# Same bridge, a while later



# Further references

- **Henry Petroski, Design Paradigms: Case Histories of Error and Judgement in Engineering**
- **A. Spector & D. Gifford, A Computer Science Perspective of Bridge Design, Comm. Of the ACM 29, 4 (1986) p 267-283**

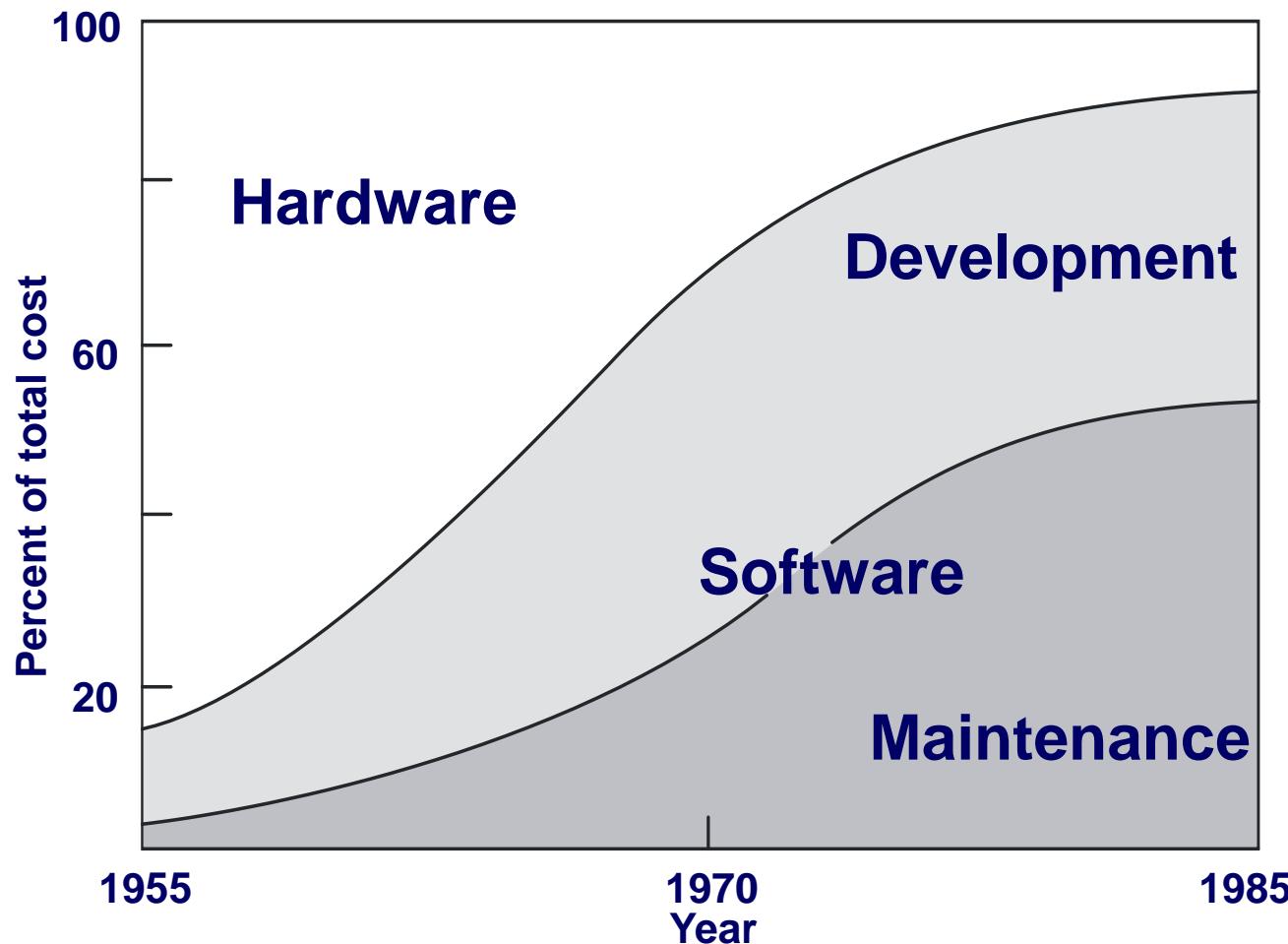
# **Errors of software**

- **errors in a software system may have serious financial consequence**
- **Quality and productivity are two central themes in the field of software engineering.**

# Cost of software

- **the cost of software is of crucial importance. This concerns not only the cost of developing the software, but also the cost of keeping the software operational once it has been delivered to the customer**

# Relative distribution of software/hardware costs



# Point to ponder #1



- Why does software maintenance cost so much?

# **1.1 WHAT IS SOFTWARE ENGINEERING?**

# Software engineering

- “*A discipline that deals with the building of software systems which are so large that they are built by a team or teams of engineers.*” [Ghezzi, Jazayeri, Mandrioli]
- “*Multi-person construction of multi-version software.*” [Parnas]

# Software engineering

- “*A discipline whose aim is the production of fault-free software, delivered on-time and within budget, that satisfies the user’s needs. Furthermore, the software must be easy to modify when the user’s needs change.*” [Schach]
- “*It’s where you get to design big stuff and be creative.*” [Taylor]

## Definition (IEEE)

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software

# Central themes

- SE is concerned with BIG programs
- complexity is an issue
- software evolves
- development must be efficient
- you're doing it together
- software must effectively support users
- involves different disciplines
- SE is a balancing act

# Programming versus software engineering

Small project	Large to huge project
You	Teams
Build what you want	Build what they want
One product	Family of products
Few sequential changes	Many parallel changes
Short-lived	Long-lived
Cheap	Costly
Small consequences	Large consequences

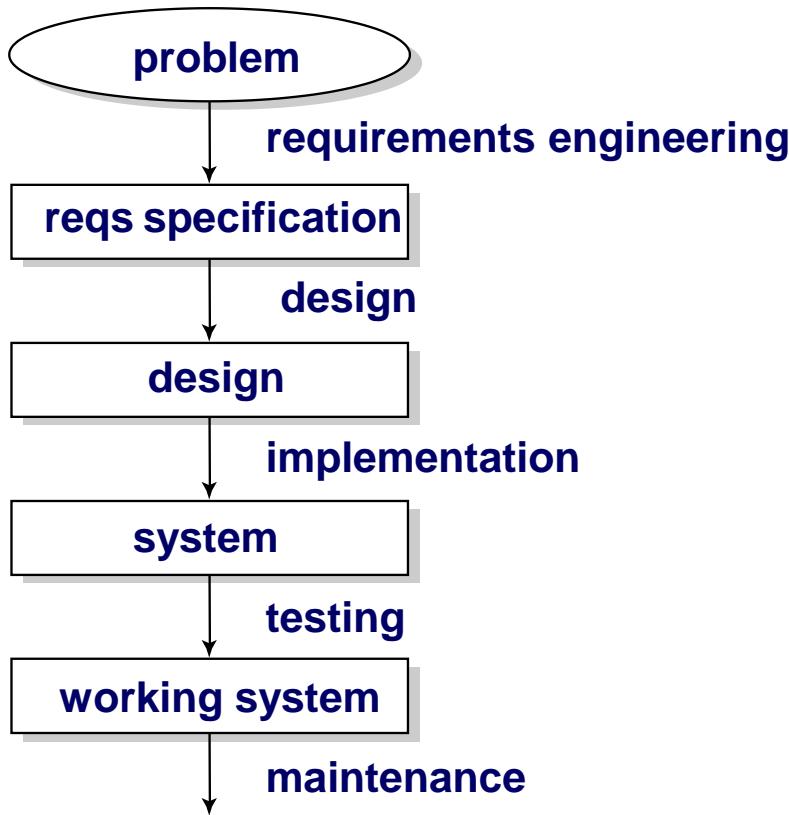


# **Building software ~ Building bridges?**

- **yes, and no**
- **software is logical, rather than physical**
- **progress is hard to see (speed ≠ progress)**
- **software is not continuous**

## **1.2 PHASES IN THE DEVELOPMENT OF SOFTWARE**

# Simple life cycle model



- Sequentially
- No backtracking
- Clear separation among activities

Questo però non avviene mai

# Point to ponder #2

Is this a good model

- of how we go about?
- of how we should go about?

# Requirements Engineering

- **yields a description of the desired system:**
  - which functions
  - possible extensions
  - required documentation
  - performance requirements
- **includes a feasibility study**
- **resulting document:** requirements specification

# Design

- earliest design decisions captured in *software architecture*
- decomposition into parts/components; what are the functions of, and interfaces between, those components?
- emphasis on *what* rather than *how*
- resulting document: *specification*

# Implementation

- **focus on individual components**
- **goal: a working, flexible, robust, ... piece of software**
- ***not a bag of tricks***
- **present-day languages have a module and/or class concept**

# Testing

- **does the software do what it is supposed to do?**
- **are we building the right system? (*validation*)**
- **are we building the system right? (*verification*)**
- **start testing activities in phase 1, on day 1**

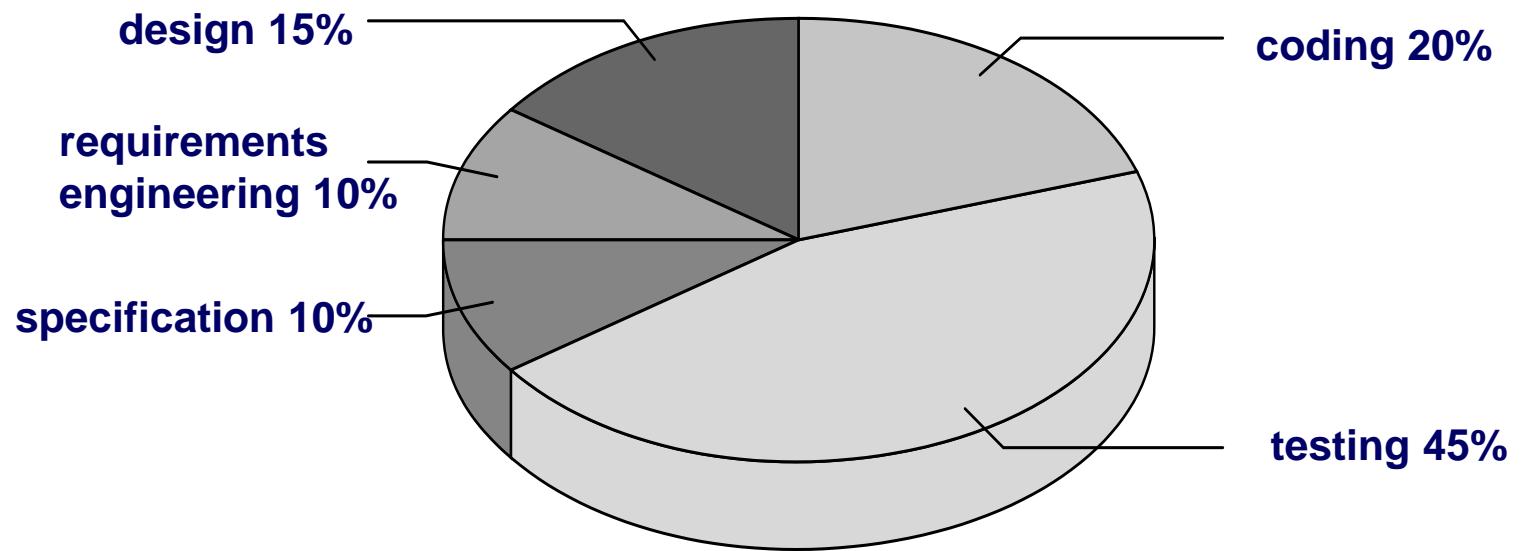
# Maintenance

- **correcting errors found after the software has been delivered**
- **adapting the software to changing requirements, changing environments, ...**

# **Project management**

- **is an activity that spans all phases**
- **An important activity not identified separately is documentation**

# Global distribution of effort



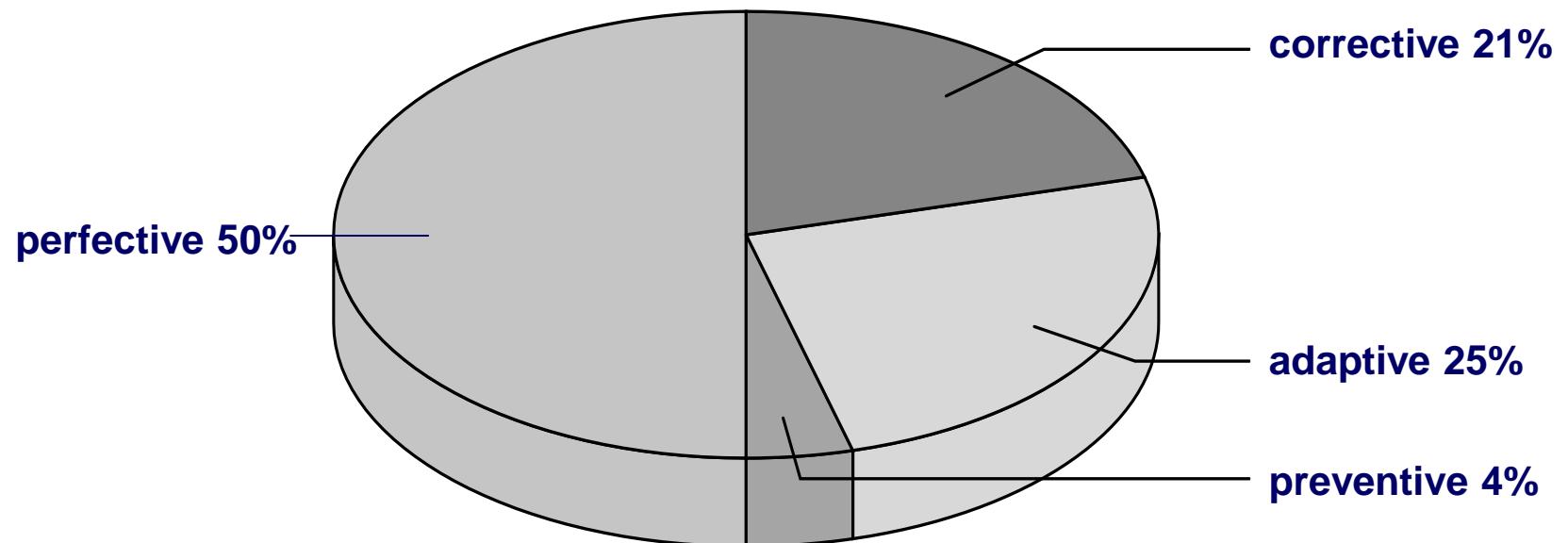
# **Global distribution of effort**

- **rule of thumb: 40-20-40 distribution of effort**
- **trend: enlarge requirements specification/design slots; reduce test slot**
- **beware: maintenance alone consumes 50-75% of total effort**

# Kinds of maintenance activities

- ***corrective maintenance***: correcting errors
- ***adaptive maintenance***: adapting to changes in the environment (both hardware and software)
- ***perfective maintenance***: adapting to changing user requirements
- ***preventive***: increasing the system's future maintainability

# Distribution of maintenance activities



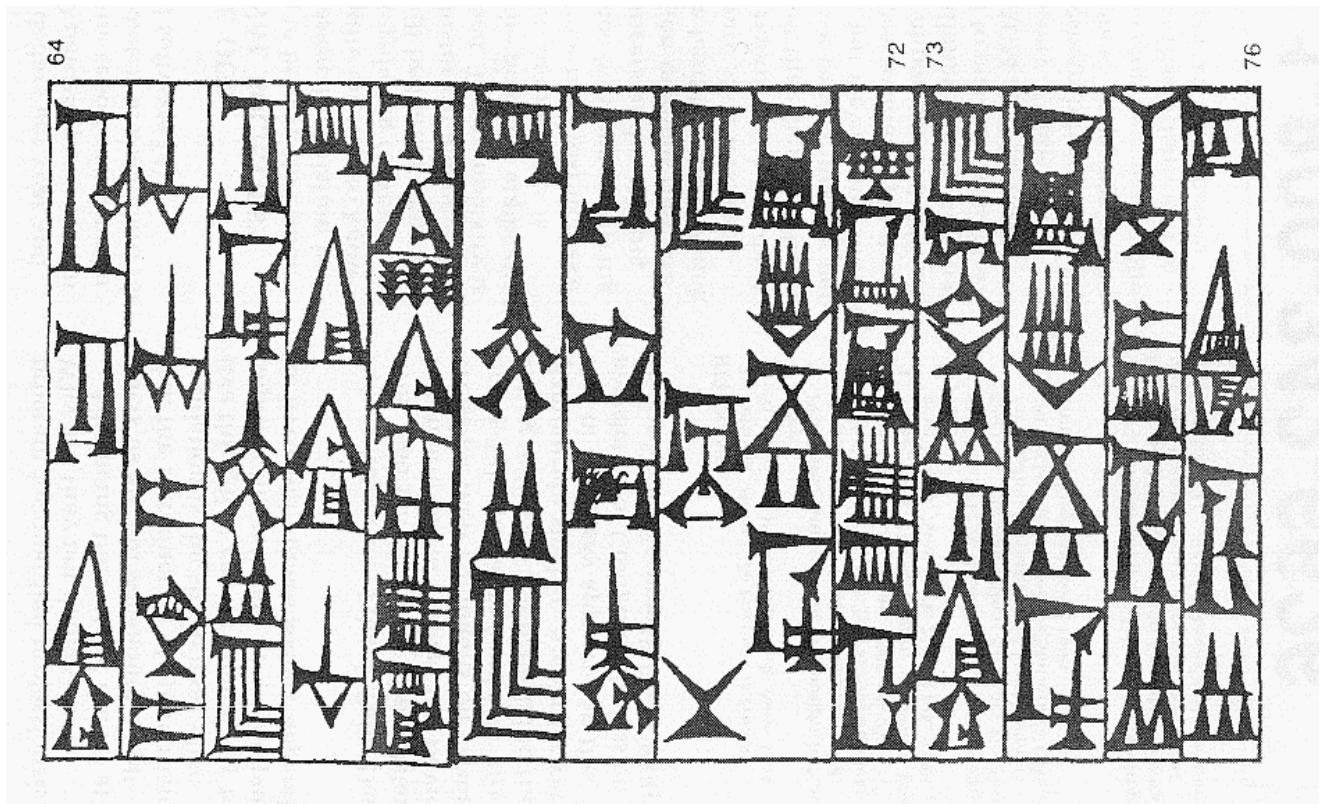
# **1.5 SOFTWARE ENGINEERING ETHICS**

## **Point to ponder #3**

**You are a tester, and the product you are testing does not yet meet the testing requirements agreed upon in writing. Your manager wants to ship the product now, continue testing so that the next release will meet the testing requirements. What do you do?**

- Discuss the issue with your manager?**
- Talk to the manager's boss?**
- Talk to the customer?**

# Hammurabi's Code



# Hammurabi's Code (translation)

**64: If a builder build a house for a man and do not make its construction firm, and the house which he has built collapse and cause the death of the owner of the house, that builder shall be put to death.**

**73: If it cause the death of a son of the owner of the house, they shall put to death a son of that builder.**

# Software Engineering Ethics - Principles

- **Act consistently with the public interest**
- **Act in a manner that is in the best interest of the client and employer**
- **Ensure that products meet the highest professional standards possible**
- **Maintain integrity in professional judgment**
- **Managers shall promote an ethical approach**
- **Advance the integrity and reputation of the profession**
- **Be fair to and supportive of colleagues**
- **Participate in lifelong learning and promote an ethical approach**

# Examples

- Approve software only if you have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life or privacy or harm the environment (clause 1.03<sup>1</sup>).
- Ensure adequate testing, debugging, and review of software and related documents on which you work (clause 3.10).
- As a manager, do not ask a software engineer to do anything inconsistent with this code of ethics (clause 5.11).

# Example of ethics

NEWS Home > Business > Business Operations > Business Management

## Facebook put "profit before public safety" with algorithm change

Ex-employee Frances Haugen reveals the social network's reasoning behind the 2018 news feed change

by: [Bobby Hellard](#) 4 Oct 2021

# Quo Vadis?

- It takes at least 15-20 years for a technology to become mature; this also holds for computer science: UNIX, relational databases, structured programming, ...
- Software engineering has made tremendous progression
- There is no silver bullet, though

# Recent developments

- Rise of agile methods
- Shift from producing software to using software
- Success of Open Source Software
- Software development becomes more heterogeneous

# The Agile Manifesto

- **Individuals and interactions over processes and tools**
- **Working software over comprehensive documentation**
- **Customer collaboration over contract negotiation**
- **Responding to change over following a plan**



# Producing software ⇒ Using software

- Builders build pieces, integrators integrate them
- Component-Based Development (CBSD)
- Software Product Lines (SPL)
- Commercial Off-The-Shelves (COTS)
- Service Orientation (SOA)

# Open Source: crowdsourcing

1. Go to LEGO site
  2. Use CAD tool to design your favorite castle
  3. Generate bill of materials
  4. Pieces are collected, packaged, and sent to you
  
  5. Leave your model in LEGO's gallery
  6. Most downloaded designs are prepackaged
- 
- No requirements engineers needed!
  - Gives rise to new business model

# Heterogeneity

- **Old days: software development department had everything under control**
- **Nowadays:**
  - Teams scattered around the globe
  - Components acquired from others
  - Includes open source parts
  - Services found on the Web

## Point to ponder #4

- **Which of the following do you consider as Software Engineering Principles?**
  - Build with and for reuse
  - Define software artifacts rigorously
  - Establish a software process that provides flexibility
  - Manage quality as formally as possible
  - Minimize software components interaction
  - Produce software in a stepwise fashion
  - Change is inherent, so plan for it and manage it
  - Tradeoffs are inherent, so make them explicit and document them
  - Uncertainty is unavoidable, so identify and manage it

# SUMMARY

- **software engineering is a balancing act, where trade-offs are difficult**
- **solutions are not right or wrong; at most they are better or worse**
- **most of maintenance is (*inevitable*) evolution**
- **there are many life cycle models, and they are all *models***

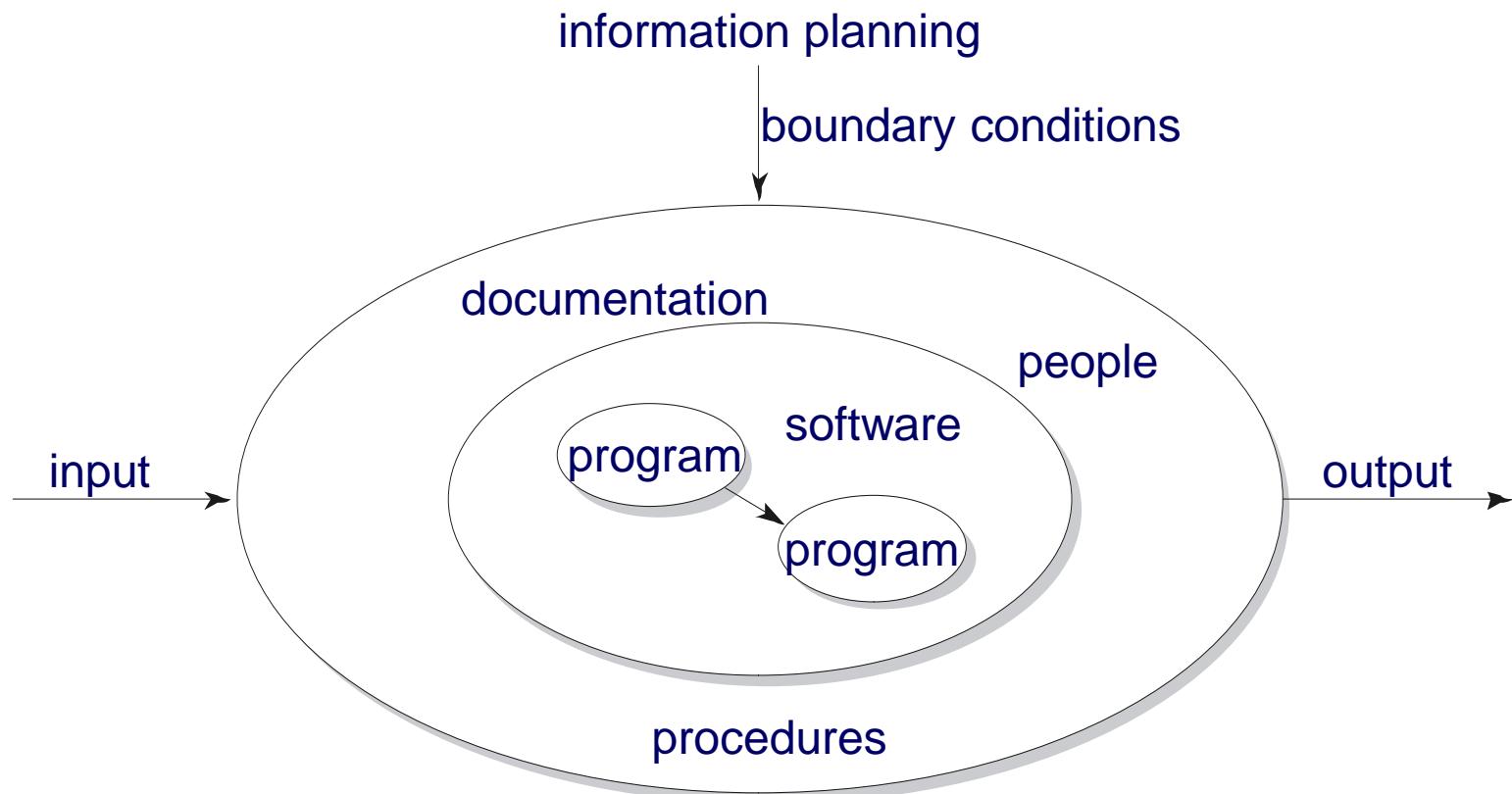
# Software Engineering Management

## CHAPTER 2

### **Main issues:**

- **Plan** - as much as possible, but not too much, up front
- **Control** - continuously

# A broader view on software development



# **Example: information plan of a university registration of student data**

- **Relations to other systems: personal data, courses, course results, alumni, ...**
- **Use both by central administration, at faculty level, and possibly by students themselves**
- **Requires training courses to administrative personnel**
- **Authorization/security procedures**
- **Auditing procedures**
- **External links, e.g. to scholarship funding agencies, ministry of education**

# **Contents of project plan**

- 1. Introduction:** project plan, the background and history, its aims, the project deliverables, the names of the persons responsible, and a summary
- 2. Process model**
- 3. Organization of the project** - The relationship of the project to other entities and the organization of the project itself are dealt with under this heading.
- 4. Standards, guidelines, procedures**
  - Example: coding standard .
- 5. Management activities**
  - Ex: management will have to submit regular reports on the status and progress of the project.
- 6. Risks**
- 7. Staffing**

# Contents of project plan

8. **Methods and techniques** - to be used during requirements engineering, design, implementation and testing are given.

EX: The necessary test environment and test equipment is described. During testing, considerable pressure will normally be put on the test equipment. Therefore, this activity has to be planned carefully.

9. **Quality assurance** - Which organization and procedures will be used to assure that the software being developed meets the quality requirements stated?

## 10. Work packages

# Contents of project plan

11. Resources

12. Budget and schedule

13. Changes

14. Delivery The procedures to be followed in handing over the system to the customer must be stated.

# Attenzione per il progetto

EXAM ALERT

## ■ Il sw development project plan deve:

- Contenere tutti i 12 punti
- Essere consegnato (condiviso) almeno 1 mese prima dell'esame
- Può essere modificato tenendo conto delle versioni

# Project control

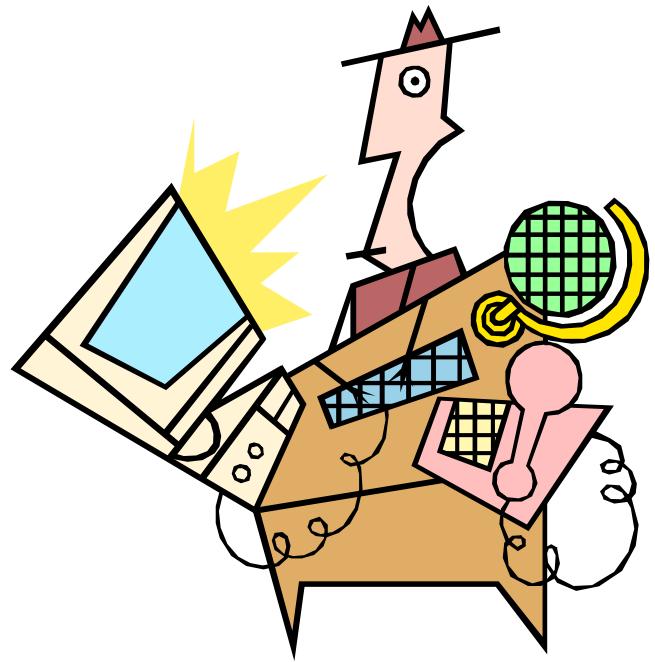
- **Time**, both the number of man-months and the schedule
- **Information**, mostly the documentation
- **Organization**, people and team aspects
- **Quality**, not an add-on feature; it has to be built in
- **Money**, largely personnel

# Managing time

- **Measuring progress is hard (“we spent half the money, so we must be halfway”)**
- **Development models serve to manage time**
- **More people ⇒ less time?**
  - Brooks’ law: adding people to a late project makes it later

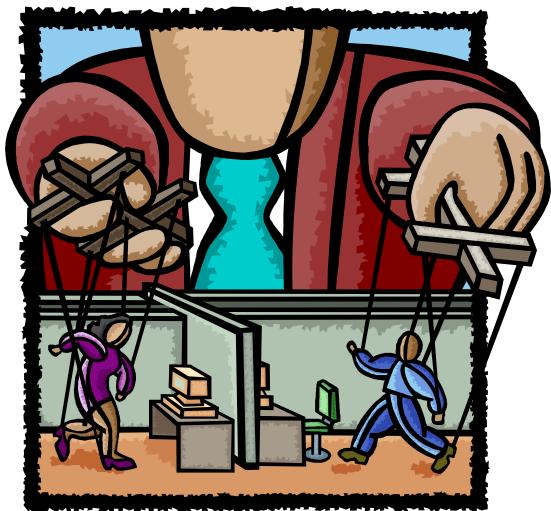
# Managing information

- **Documentation**
  - Technical documentation
  - Current state of projects
  - Changes agree upon
  - ...
- **Agile projects: less attention to explicit documentation, more on tacit knowledge held by people**



# Managing people

- Managing expectations
- Building a team
- Coordination of work



# Managing quality

- Quality has to be designed in
- Quality is not an afterthought
- Quality requirements often conflict with each other
- Requires frequent interaction with stakeholders



# Managing cost

- Which factors influence cost?
- What influences productivity?
- Relation between cost and schedule

# Summary

- **Project planning**
- **Project control concerns**
  - Time
  - Information
  - Organization
  - Quality
  - Money
- **Agile projects do less planning than document-driven projects**

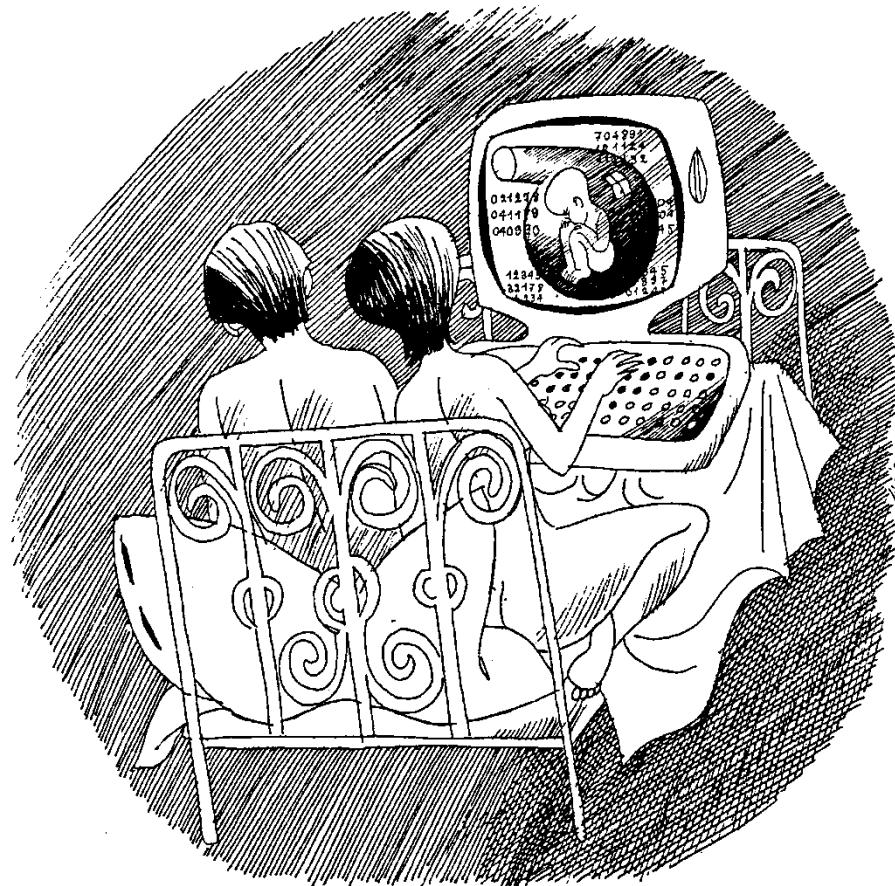
# **Software Life Cycle**

## **capitolo 3**

### **Main issues:**

- Discussion of different life cycle models**
- Maintenance or evolution**

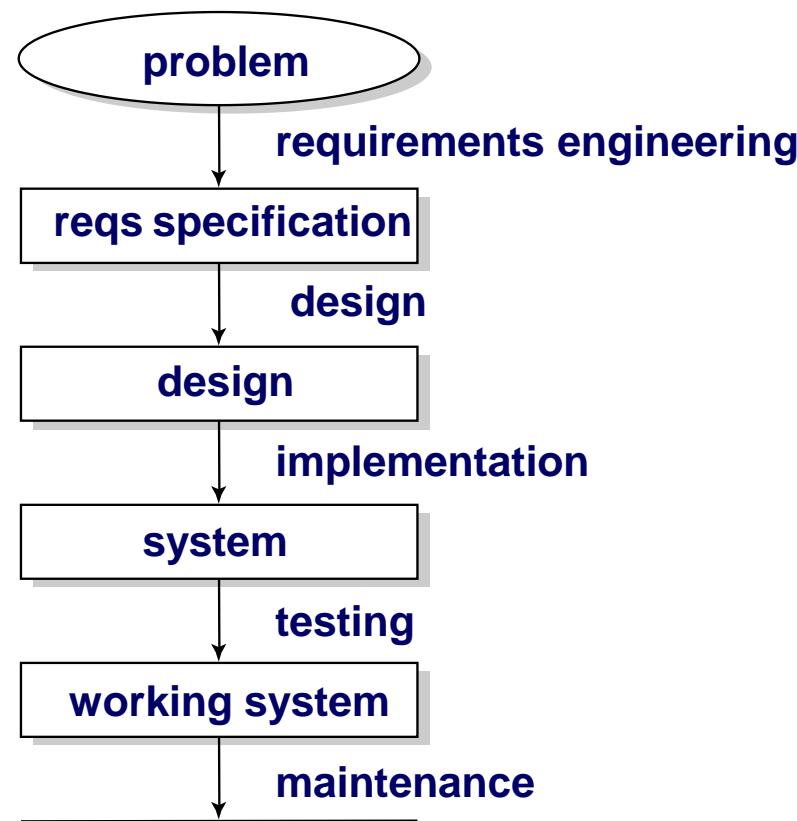
# Not this life cycle



# Introduction

- **software development projects are large and complex**
- **a phased approach to control it is necessary**
- **traditional models are document-driven: there is a new pile of paper after each phase is completed**
- **evolutionary models recognize that much of what is called maintenance is inevitable**
- **latest fashion: agile methods, eXtreme Programming**
- **life cycle models can be explicitly modeled, in a process modeling language**

# Simple life cycle model



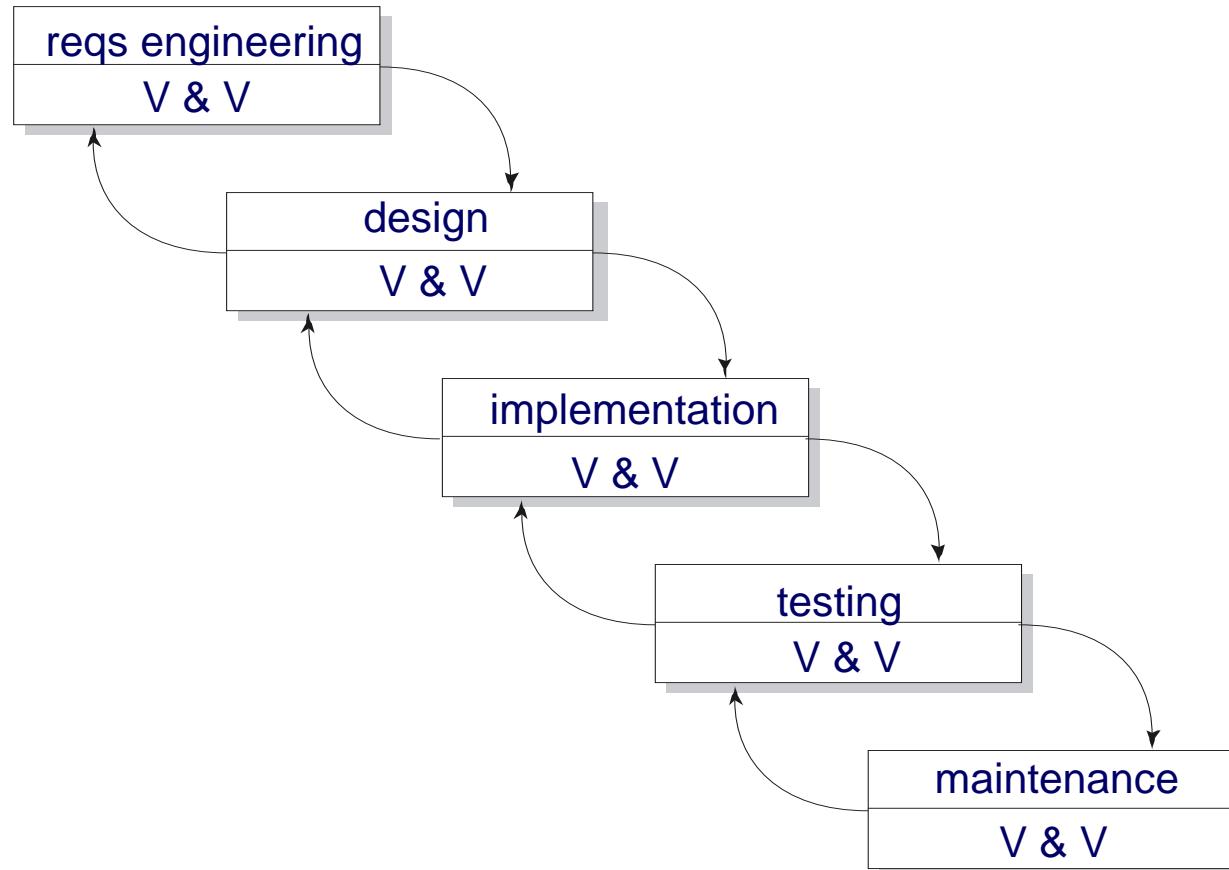
# Point to ponder #1

- **Why does the model look like this?**
- **Is this how we go about?**

# Simple Life Cycle Model

- **document driven, planning driven, heavyweight**
- **milestones are reached if the appropriate documentation is delivered (e.g., requirements specification, design specification, program, test document)**
- **much planning upfront, often heavy contracts are signed**
- **problems**
  - feedback is not taken into account
  - maintenance does not imply evolution

# Waterfall Model



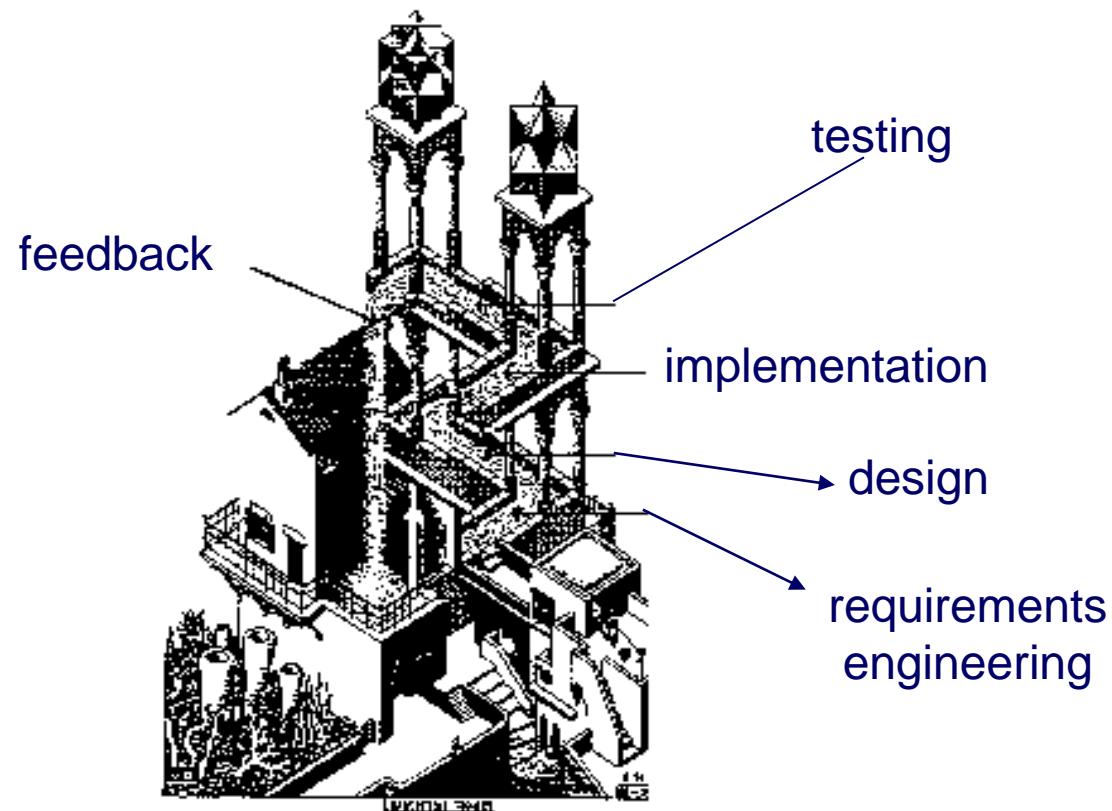
# V&V

- **V & V stands for Verification and Validation.**
- **Verification asks if the system meets its requirements (are we building the system right) and thus tries to assess the correctness of the transition to the next phase.**
- **Validation asks if the system meets the user's requirements (are we building the right system).**

# Waterfall model

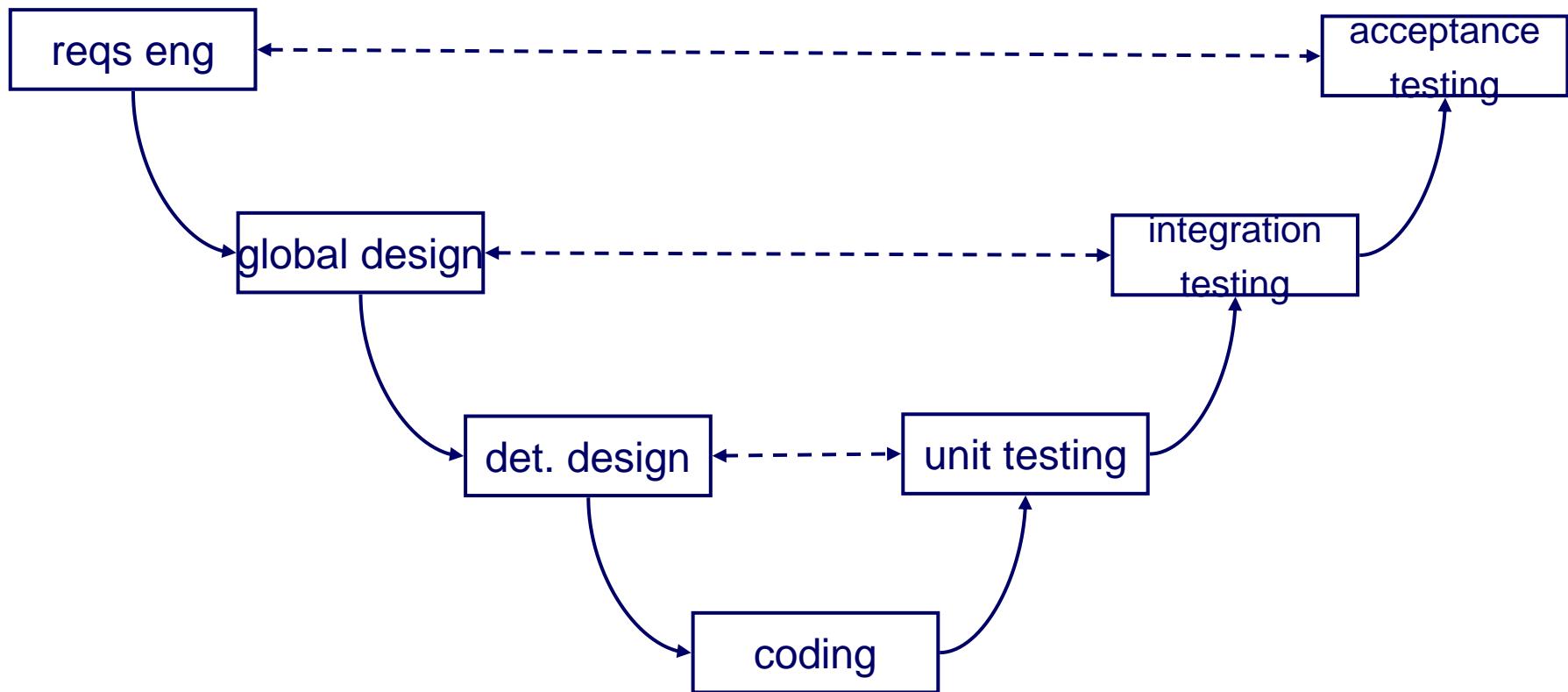
- **emphasis on a careful analysis before the system is actually built**
- **identify the user's requirements as early as possible.**
  - it is difficult in practice, if not impossible
- **a regular test should also be carried out with the prospective user.**

# Another waterfall model



# V-Model

different types of testing are related to the various development phases



# Waterfall Model (cntd)

- **includes iteration and feedback**
- **user requirements are fixed as early as possible**
- **problems**
  - too rigid
  - developers cannot move between various abstraction levels

# Activity versus phase

Activity \ Phase	Design	Implementation	Integration testing	Acceptance testing
Integration testing	4.7	43.4	26.1	25.8
Implementation (& unit testing) CODING	6.9	70.3	15.9	6.9
Design	49.2	34.1	10.3	6.4

Design during testing?

# Lightweight (agile) approaches

- prototyping
- incremental development
- RAD, DSDM
- XP



# The Agile Manifesto

## Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

**Individuals and interactions** over processes and tools

**Working software** over comprehensive documentation

**Customer collaboration** over contract negotiation

**Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

# **Individuals over processes**

- **Individuals and interactions over processes and tools**
- They emphasize the human element in software development. Team spirit is considered very important. Team relationships are close. Often, an agile team occupies one big room. The users are on site as well. Agile methods have short communication cycles between developers and users, and among developers.

# Sw vs documentation

- **Working software over comprehensive documentation**
- Why spend time on something that will soon be outdated? Rather, agile methods rely on the tacit knowledge of the people involved. If you have a question, ask one of your friends. Do not struggle with a large pile of paper that quite likely will not provide the answer anyway.

# **Customer collaboration**

- **Customer collaboration over contract negotiation**

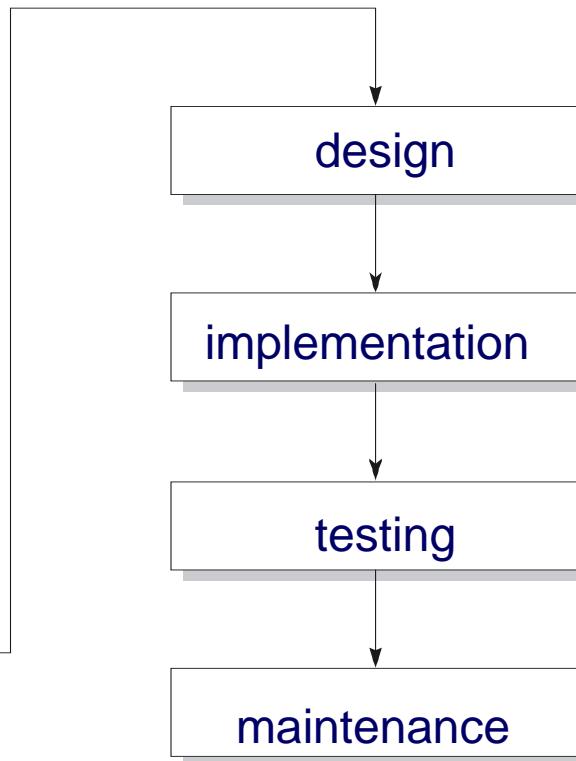
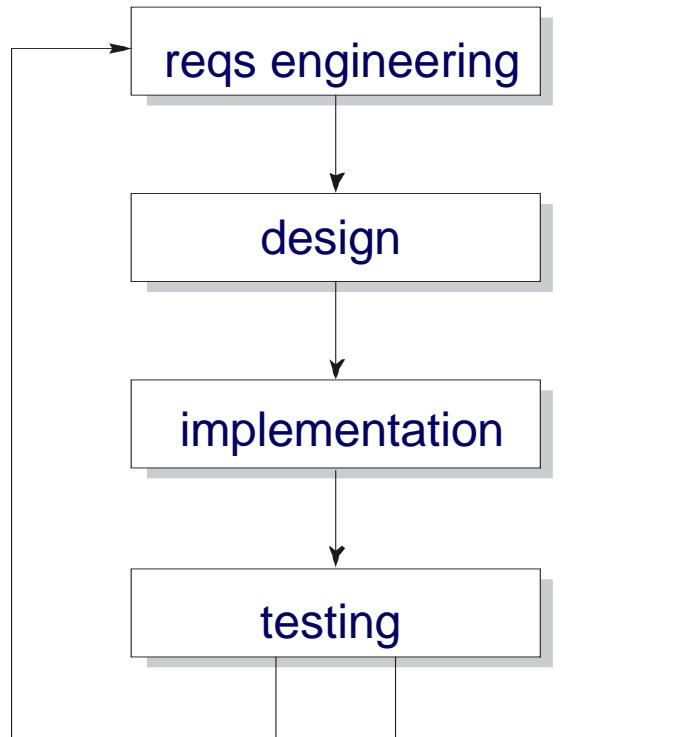
# Embracing the change

- **Responding to change over following a plan**
- Agile methods involve the users in every step taken. The development cycles are small and incremental. The series of development cycles is not extensively planned in advance, but the new situation is reviewed at the end of each cycle. This includes some, but not too much, planning for the next cycle.

# Prototyping

- **requirements elicitation is difficult**
  - software is developed because the present situation is unsatisfactory
  - however, the desirable new situation is as yet unknown
- **prototyping is used to obtain the requirements of some aspects of the system**
- **prototyping should be a relatively cheap process**
  - use rapid prototyping languages and tools
  - not all functionality needs to be implemented
  - production quality is not required

# Prototyping as a tool for requirements engineering



One of the main difficulties for users is to express their requirements precisely. It is natural to try to clarify these through prototyping.

# Prototyping (cntd)

- ***throwaway prototyping*: the n-th prototype is followed by a waterfall-like process (as depicted on previous slide)**
- ***evolutionary prototyping*: the nth prototype is delivered**

## **Point to ponder #2**

**What are the pros and cons of the two approaches?**

# Prototyping, advantages

- **The resulting system is easier to use**
- **User needs are better accommodated**
- **The resulting system has fewer features**
- **Problems are detected earlier**
- **The design is of higher quality**
- **The resulting system is easier to maintain**
- **The development incurs less effort**

# Prototyping, disadvantages

- **The resulting system has more features**
- **The performance of the resulting system is worse**
- **The design is of less quality**
- **The resulting system is harder to maintain**
- **The prototyping approach requires more experienced team members**

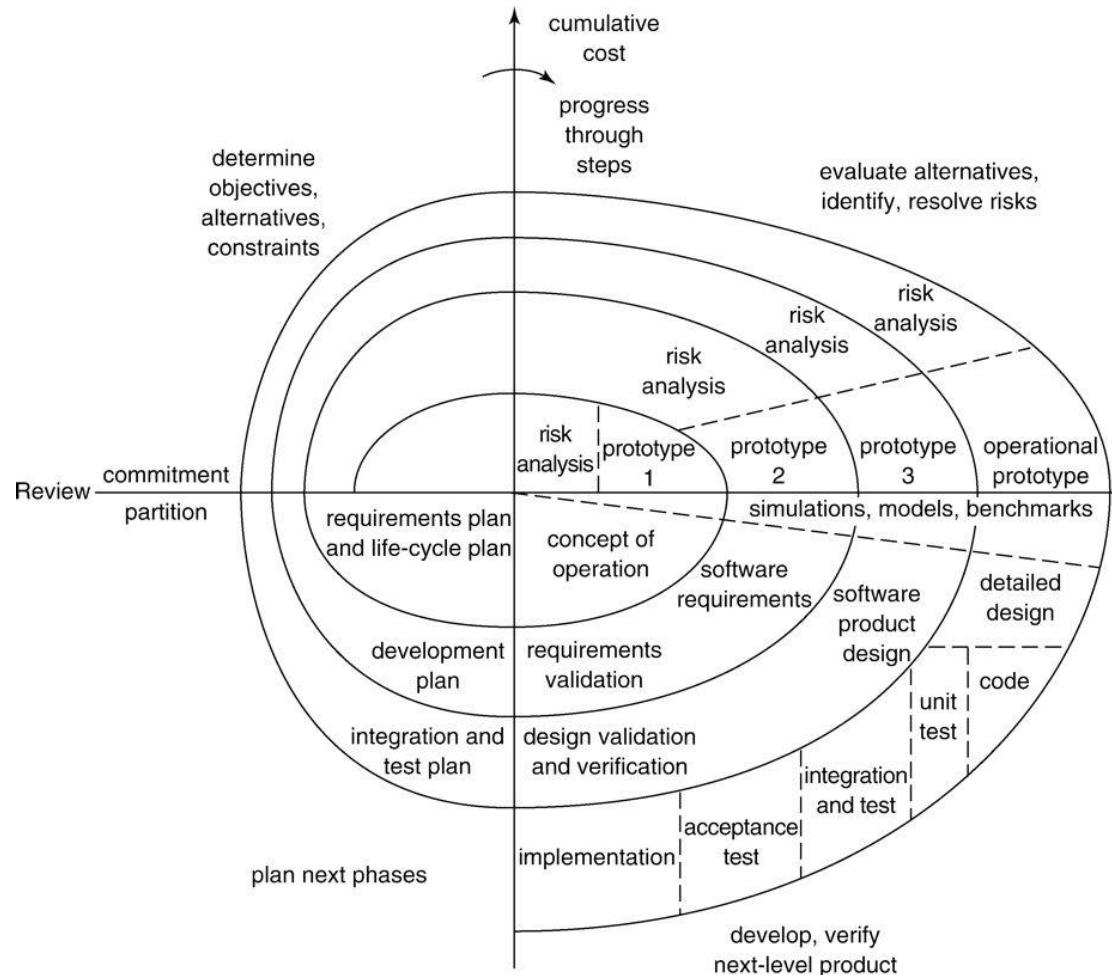
# Prototyping, recommendations

- **the users and the designers must be well aware of the issues and the pitfalls**
- **use prototyping when the requirements are unclear**
- **prototyping needs to be planned and controlled as well**

# Incremental Development

- **a software system is delivered in small increments, thereby avoiding the Big Bang effect**
- **the waterfall model is employed in each phase**
- **the user is closely involved in directing the next steps**
- **incremental development prevents overfunctionality**

# The spiral model



# RAD: Rapid Application Development

- **evolutionary development, with *time boxes*:** fixed time frames within which activities are done;
- **time frame is decided upon first, then one tries to realize as much as possible within that time frame;**
- **other elements:** Joint Requirements Planning (JRD) and Joint Application Design (JAD), workshops in which users participate;
- **requirements prioritization through a *triage*;**
- **development in a SWAT team: Skilled Workers with Advanced Tools**

# DSDM

- **Dynamic Systems Development Method, #1 RAD framework in UK**
- **Fundamental idea: fix time and resources (*timebox*), adjust functionality accordingly**
- **One needs to be a member of the DSDM consortium**

# DSDM phases

- **Feasibility:** delivers feasibility report and outline plan, optionally fast prototype (few weeks)
- **Business study:** analyze characteristics of business and technology (in workshops), delivers a.o. System Architecture Definition
- **Functional model iteration:** timeboxed iterative, incremental phase, yields requirements
- **Design and build iteration**
- **Implementation:** transfer to production environment

# DSDM practices

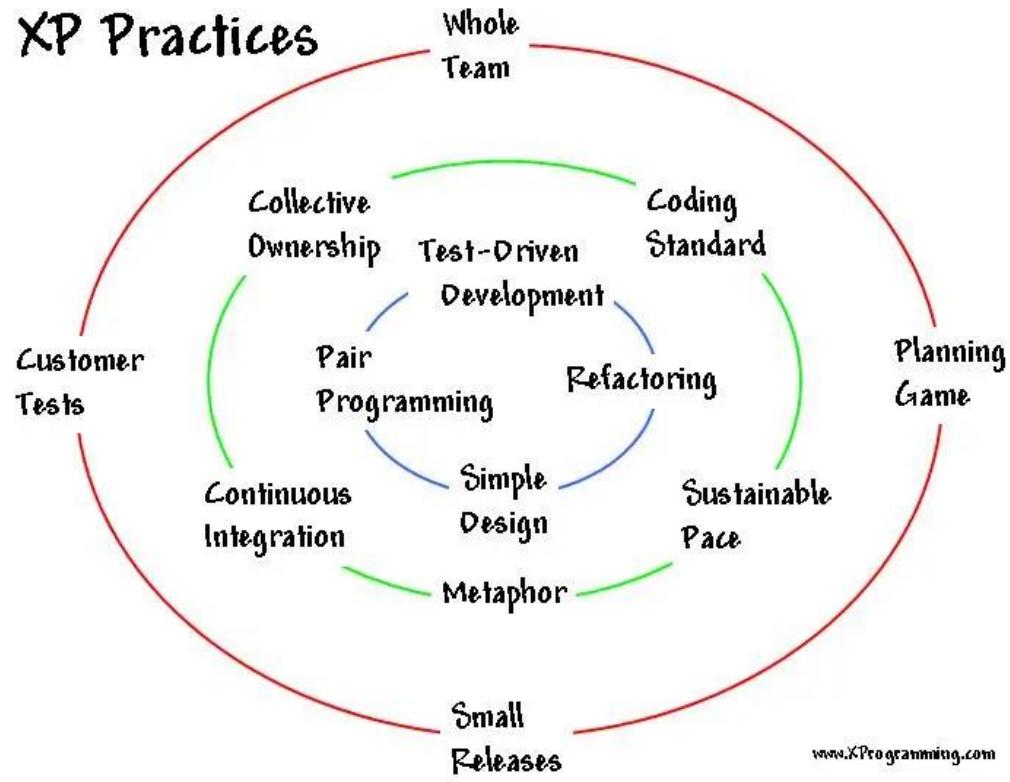
- **Active user involvement is imperative**
- **Empowered teams**
- **Frequent delivery of products**
- **Acceptance determined by fitness for business purpose**
- **Iterative, incremental development**
- **All changes are reversible**
- **Requirements baselined at high level**
- **Testing integrated in life cycle**
- **Collaborative, cooperative approach shared by all stakeholders is essential**

# XP – eXtreme Programming

- **Everything is done in small steps**
- **The system always compiles, always runs**
- **Client as the center of development team**
- **Developers have same responsibility w.r.t. software and methodology**

# 13 practices of XP

1. Whole team: client part of the team
2. Metaphor: common analogy for the system
3. The planning game, based on user stories
4. Simple design
5. Small releases (e.g. 2 weeks)
6. Customer tests
7. Pair programming
8. Test-driven development: tests developed first
9. Design improvement (refactoring)



10. Collective code ownership
11. Continuous integration: system always runs
12. Sustainable pace: no overtime
13. Coding standards

[www.XProgramming.com](http://www.XProgramming.com)

# XP practices

- **Whole team**

Il gruppo di lavoro include persone che, nell'insieme, hanno competenze di analisi, progettazione, programmazione e test. L'idea è che anche le singole persone abbiano più competenze specialistiche.

- **Planning game**

Lo sviluppo dell'applicazione è accompagnato dalla stesura di un piano di lavoro. Il piano è definito e, continuamente aggiornato, a intervalli brevi e regolari dai responsabili del progetto, secondo le priorità aziendali e le stime dei programmati.

- **Small releases**

Lo sviluppo dell'applicazione prevede rilasci di versioni del prodotto funzionanti. Ogni rilascio realizza un insieme di casi d'uso (storie di utilizzo che operano come specifica). Ogni rilascio è la conclusione di un'iterazione di sviluppo e l'inizio di una nuova pianificazione. Ogni iterazione dovrebbe durare non più di qualche settimana.

# XP practices

- **Customer test**

Il committente deve essere coinvolto nello sviluppo. Partecipa alla stesura dei test di sistema e verifica periodicamente che il sistema realizzato corrisponda effettivamente alle proprie esigenze.

- **Collective ownership**

Il codice dell'applicazione può essere liberamente manipolato da qualsiasi sviluppatore (che deve essere messo nelle migliori condizioni di comprenderlo).

# XP practices

- **Coding standard**

Il codice deve essere scritto sulla base di regole condivise. Tutti gli sviluppatori devono essere in grado di capire e modificare ogni linea di codice scritta da altri.

- **Sustainable pace**

Lo sviluppo di applicazioni con i metodi dell'eXtreme Programming è un'attività che richiede grande concentrazione. Lo straordinario è da evitare per non provocare un deterioramento della qualità dell'impegno.

# XP practices

- **Metaphor**

Ogni progetto è guidato da una metafora condivisa da responsabili e sviluppatori. La metafora è una descrizione sintetica del sistema nel suo complesso e fornisce un vocabolario comune a tutte le persone coinvolte, senza scendere nei dettagli implementativi.

- **Continuous integration**

Il codice sviluppato è frequentemente integrato in modo da avere una versione funzionante a disposizione di tutti i programmatore che cresce.

# XP practices

- **Test driven development**

L'applicazione è verificata sia a livello di sistema (test di sistema) sia a livello del singolo metodo (test di unità). I test di sistema sono basati su casi d'uso. I test di unità sono rieseguibili automaticamente. L'insieme di test per un modulo è rieseguito ad ogni modifica del modulo

- **Refactoring**

L'applicazione è periodicamente ristrutturata per eliminare parti superflue o semplificare la struttura.

# XP practices

- **Simple design**

L'architettura dell'applicazione deve essere la più semplice possibile. A fronte di nuove situazioni, saranno progettati nuovi componenti o riprogettati quelli esistenti. XP - Sintesi

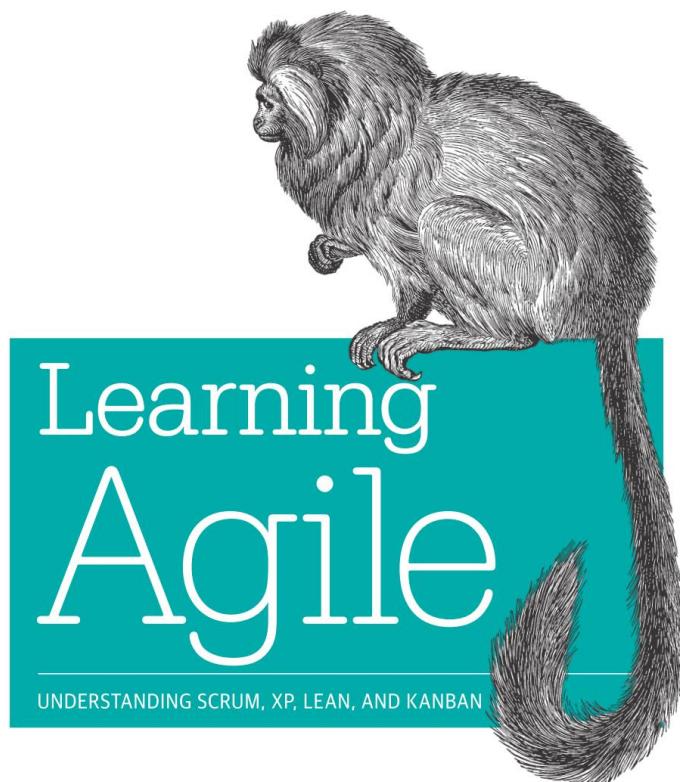
- **Pair programming**

La scrittura del codice è fatta da coppie di programmatore che lavorano al medesimo terminale. Le coppie non sono fisse, ma si compongono associando le migliori competenze per la risoluzione di uno specifico problema. Il lavoro in coppia permette, scambiandosi periodicamente i ruoli, di mantenere mediamente più alto il livello d'attenzione.

# Altri metodi agili

O'REILLY®

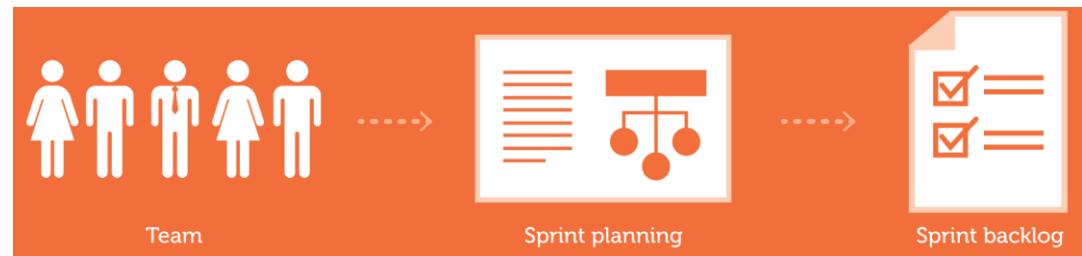
- XP
- SCRUM (esercitazione)
- Lean Kanban



Andrew Stellman & Jennifer Greene

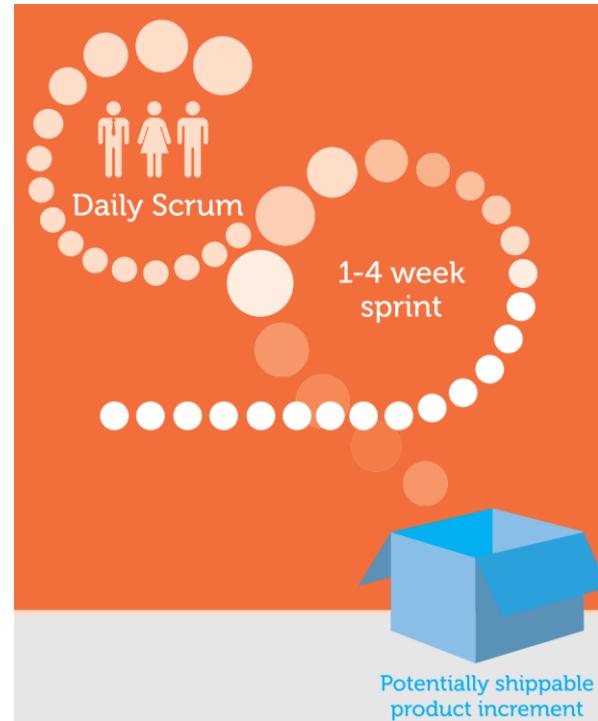
# SCRUM

- A **product owner** creates a prioritized wish list called a **product backlog**.
- During **sprint** planning, the team pulls a small chunk from the top of that wishlist, a **sprint backlog**, and decides how to implement those pieces.



# SCRUM (sprint)

- The team has a certain amount of time, a **sprint**, to complete its work – usually two to four weeks – but meets each day to assess its progress (daily scrum).
- At the end of the sprint, the work should be potentially shippable, as in ready to hand to a customer, put on a store shelf, or show to a stakeholder.



# SCRUM

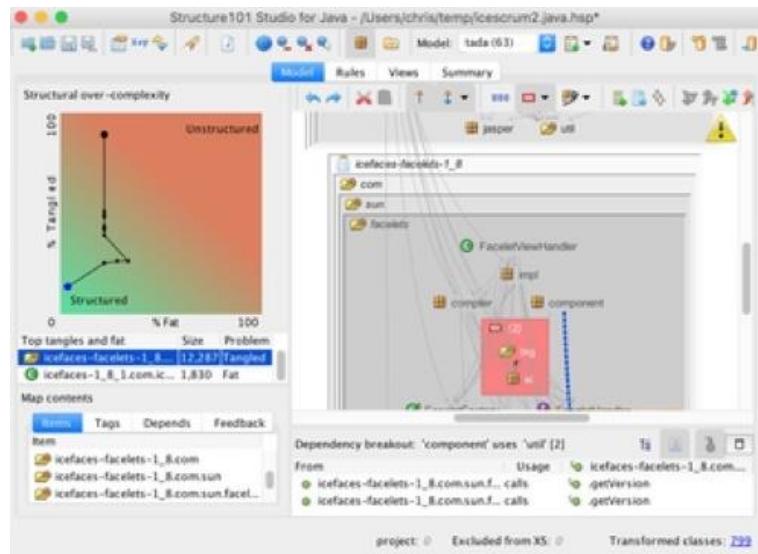
- Along the way, the **ScrumMaster** keeps the team focused on its goal.
- The **sprint ends** with a sprint review and retrospective.
- As the next sprint begins, the team chooses another chunk of the product backlog and begins working again.

The cycle repeats until enough items in the product backlog have been completed, the budget is depleted, or a deadline arrives. Which of these milestones marks the end of the work is entirely specific to the project. No matter which impetus stops work, Scrum ensures that the most valuable work has been completed when the project ends.

# Refactoring tools

- <https://structure101.com/>

- stanide



# RUP

- **Rational Unified Process**
- **Complement to UML (Unified Modeling Language)**
- **Iterative approach for object-oriented systems, strongly embraces use cases for modeling requirements**
- **Tool-supported (UML-tools, ClearCase)**

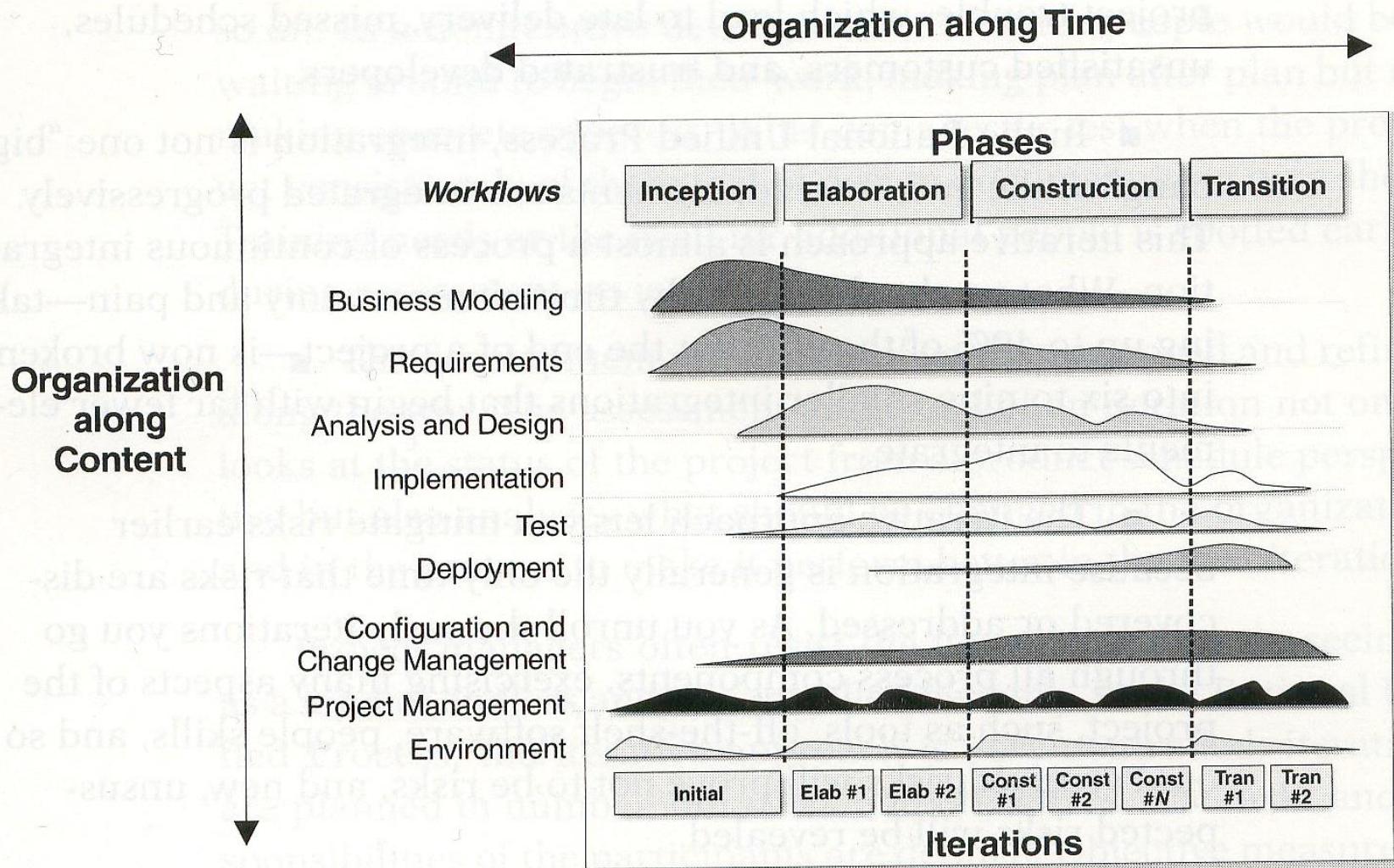
# RUP principles

- **It has a well-defined process and includes reasonably extensive upfront requirements-engineering activities, yet it emphasizes stakeholder involvement through its use-case-driven nature.**
- **RUP distinguishes four phases: inception, elaboration, construction, and transition. Within each phase, several iterations may occur.**

# RUP phases

- ***Inception:*** establish scope, boundaries, critical use cases, candidate architectures, schedule and cost estimates
- ***Elaboration:*** foundation of architecture, establish tool support, get all use cases
- ***Construction:*** manufacturing process, one or more releases
- ***Transition:*** release to user community, often several releases

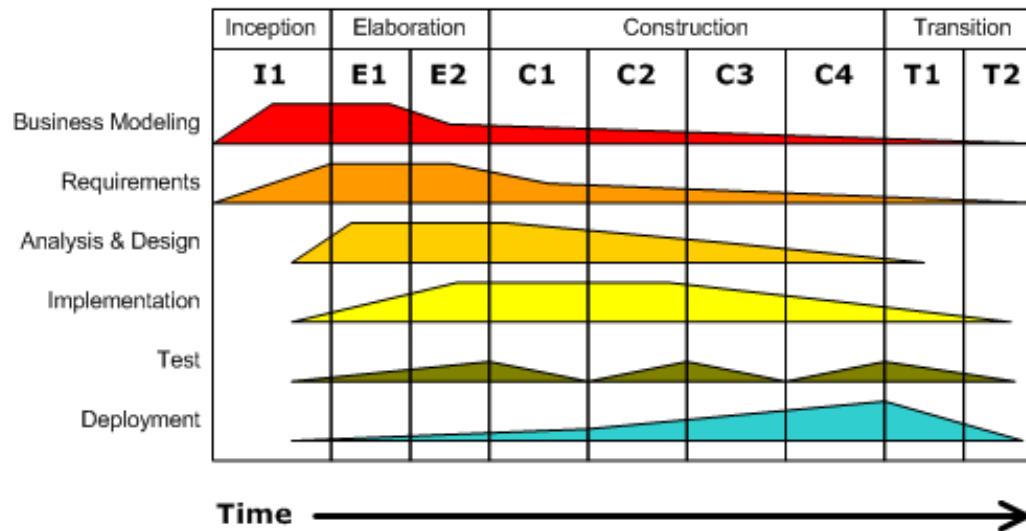
# Two-dimensional process structure of RUP



# Iterative development in RUP

## Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.



# RUP - practices

Practice	Description
Iterative development	Systems are developed in an iterative way. This is not an uncontrolled process. Iterations are planned and progress is measured carefully.
Requirements management	RUP has a systematic approach to eliciting, capturing and managing requirements, including possible changes to these requirements.
Architecture and use of components	The early phases of RUP result in an architecture. This architecture is used in the remainder of the project. It is described in different views. RUP supports the development of component-based systems, in which each component is a nontrivial piece of software with well-defined boundaries.
Modeling and UML	Much of RUP is about developing models, such as a use-case model, a test model, etc. These models are described in UML.
Quality of process and product	Quality is not an add-on, but the responsibility of everyone involved. The testing workflow is aimed at verifying that the expected level of quality is met.
Configuration and change management	Iterative development projects deliver a vast amount of products, many of which are frequently modified. This asks for sound procedures to do so, and appropriate tool support.
Use-case-driven development	Use cases describe the behaviour of the system. They play a major role in various workflows, especially the requirements, design, test and management workflow.
Process configuration	One size does not fit all. Though RUP can be used 'as-is', it can also be modified and tailored to better fit specific circumstances.
Tool support	To be effective, a software development methodology needs tool support. RUP is supported by a wide variety of tools, especially in the area of visual modeling and configuration management.

# Differences for developers

- ***Agile:* knowledgeable, collocated, collaborative**
- ***Heavyweight:* plan-driven, adequate skills, access to external knowledge**

# Differences for customers

- ***Agile:* dedicated, knowledgeable, collocated, collaborative, representative, empowered**
- ***Heavyweight:* access to knowledgeable, collaborative, representative, empowered customers**

# Differences for requirements

- *Agile*: largely emergent, rapid change
- *Heavyweight*: knowable early, largely stable

# Differences for architecture

- ***Agile***: designed for current requirements
- ***Heavyweight***: designed for current and foreseeable requirements

# Differences for size

- ***Agile*: smaller teams and products**
- ***Heavyweight*: larger teams and products**

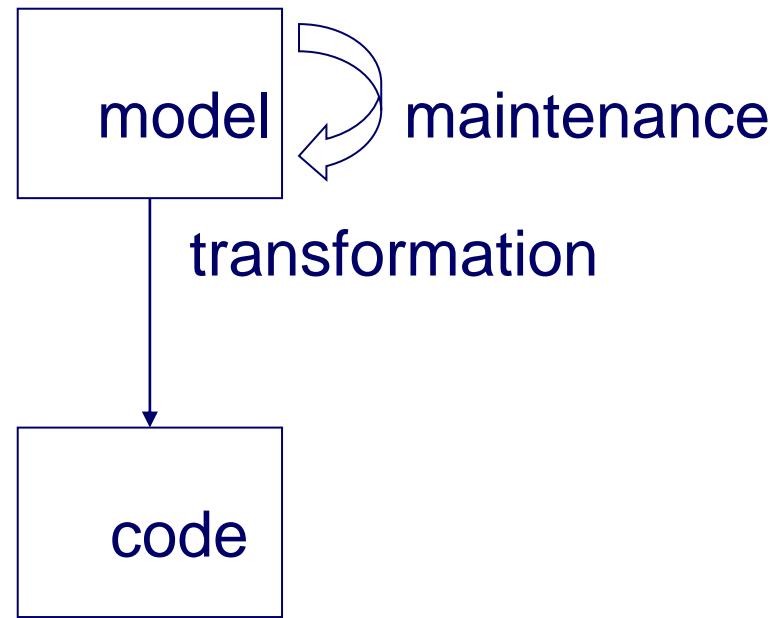
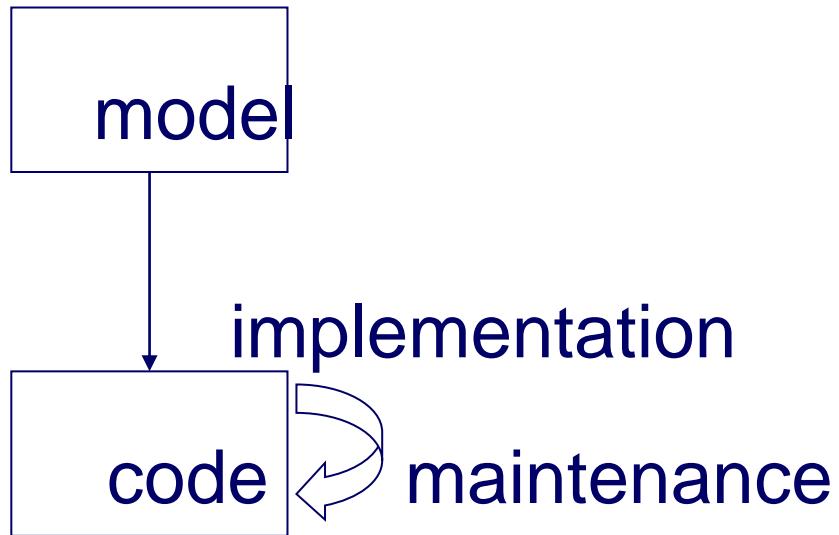
# Differences for primary objective

- *Agile*: rapid value
- *Heavyweight*: high assurance

# Model driven architecture/engeneering

- In traditional software development approaches, we build all kinds of models during requirements engineering and design.
  - From that all the source code is derived (manually)
  - Code = ultimate model
- Any evolution is accomplished by changing the last model in the chain, i.e. the source code
- requirements and design models are often not updated, so that they quickly become outdated.
- Is there a better bay to develop a system?

# MDA – Model Driven Architecture



# **Essence of MDA**

- computation-independent model (CIM)
- **Platform Independent Model (PIM)**
- **Model transformation and refinement**
- **Resulting in a Platform Specific Model (PSM)**

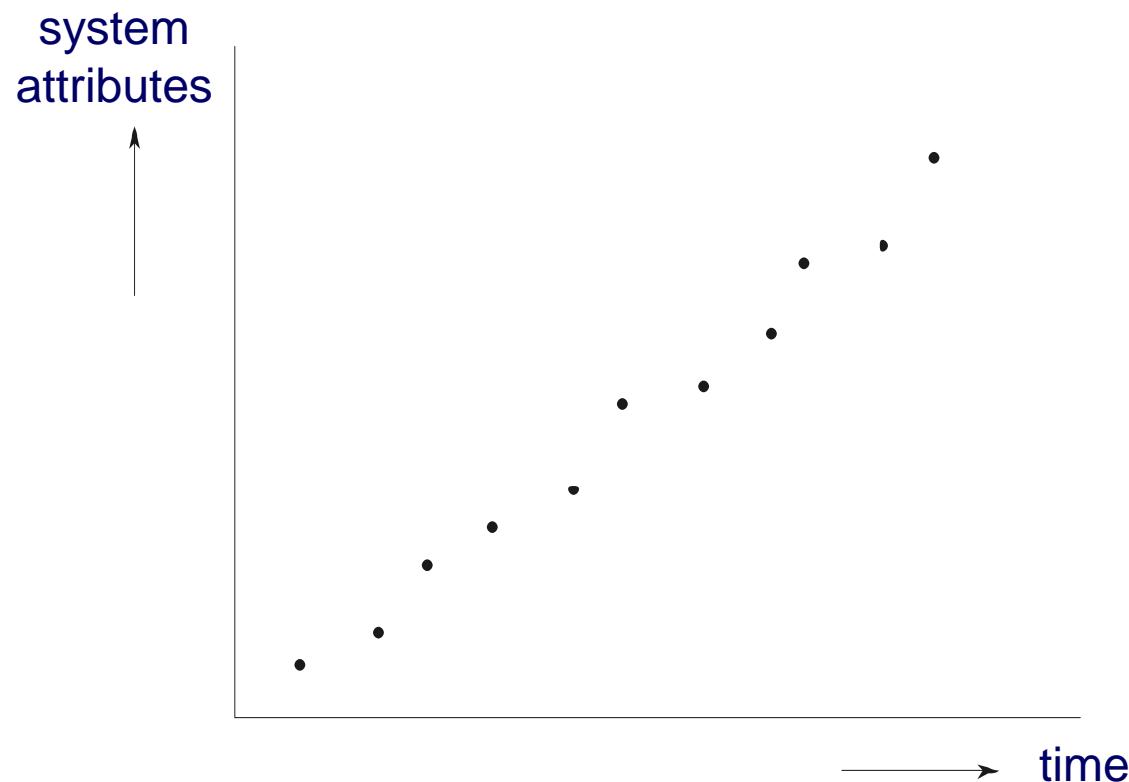
# Example YAKINDU SCT

- **Modeling complex systems with state machines**
  - Similar to the UML state machines
- **Simulating and testing your system's behavior with ease**
- **Generating high-quality source code for your target platform**

# Maintenance or Evolution

- **some observations**
  - systems are not built from scratch
  - there is time pressure on maintenance
- **the five laws of software evolution**
  - law of continuing change
  - law of increasingly complexity
  - law of program evolution
  - law of invariant work rate
  - law of incremental growth limit

# Illustration third law of Software Evolution



# Software Product Lines



- **developers are not inclined to make a maintainable and reusable product, it has additional costs**
- **this viewpoint is changed somewhat if the *product family* is the focus of attention rather than producing a single version of a product**
- **two processes result: *domain engineering*, and *application engineering***

# Domain engineering vs Application engineering

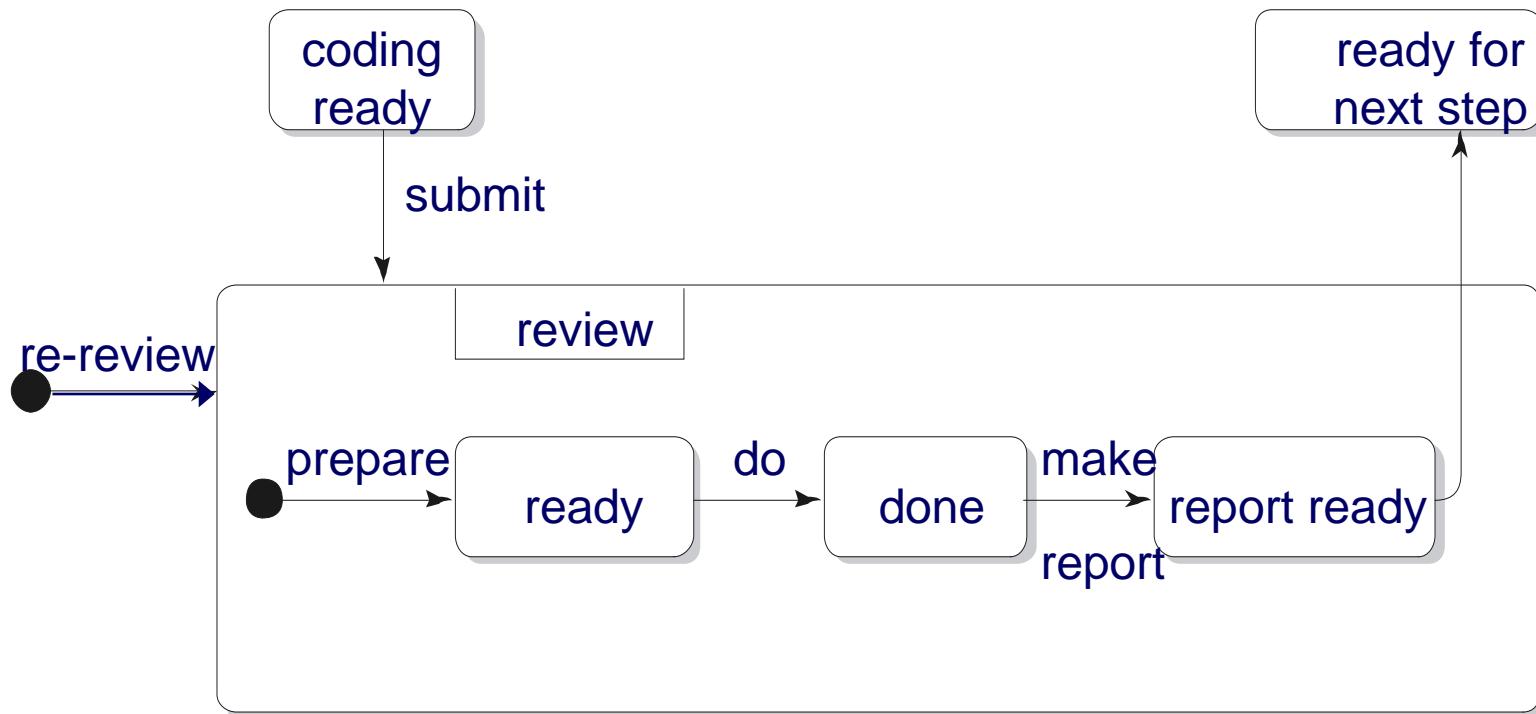
	Problem space	Solution space
In domain engineering, we analyze the domain for which we are going to develop. This process has a life cycle of its own	Domain analysis	set of reusable components - Reference architecture
Application engineering concerns the development of individual products  From customer needs	Requirement analysis	Product derivation

# Process modeling

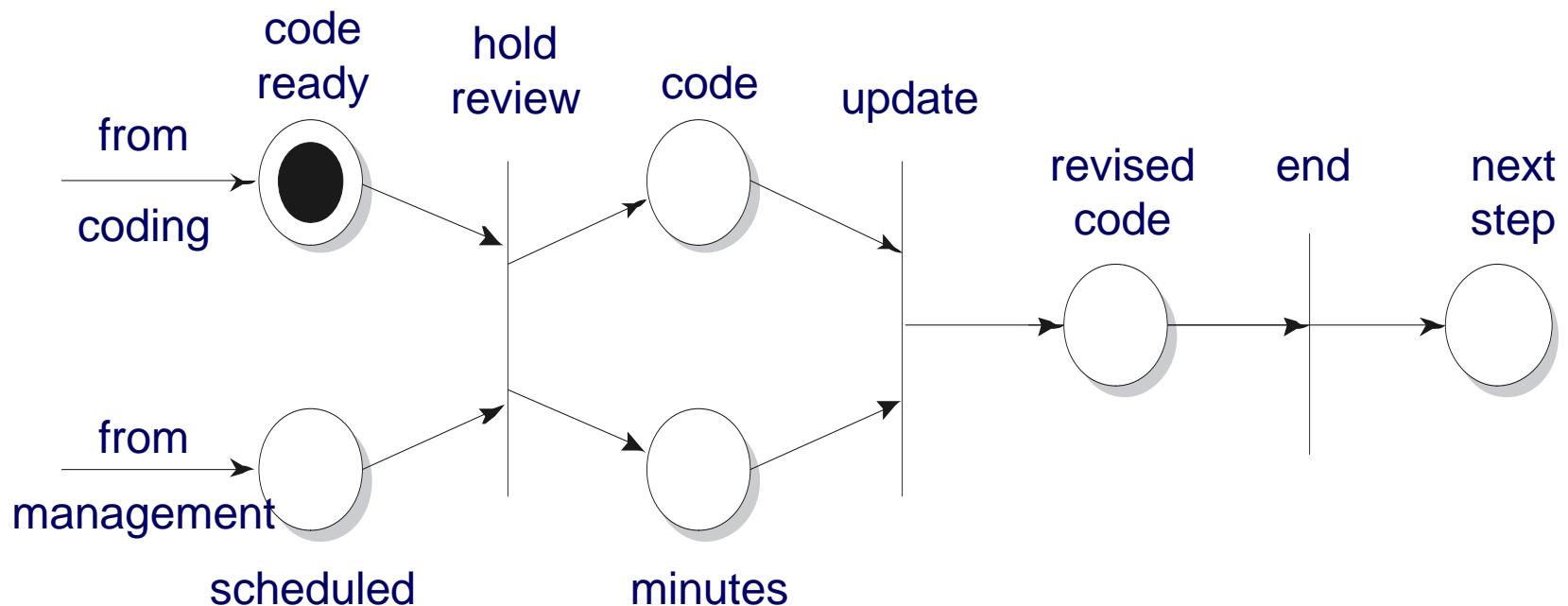
- **we may describe a software-development process, or parts thereof, in the form of a “program” too. E.G.:**

```
function review(document, threshold): boolean;  
begin prepare-review;  
    hold-review{document, no-of-problems);  
    make-report;  
    return no-of-problems < threshold  
end review;
```

# STD of review process



# Petri-net view of the review process



# Petri Nets

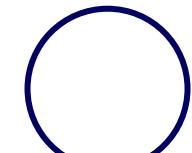
- Originate from C.A. Petri's PhD thesis (1962).
- They were originally conceived as a technique for the description and *analysis of concurrent behaviour in distributed systems*.
- Based on a few simple concepts, yet expressive.
- They have a simple graphical format and a precise operational semantics that makes them an attractive option for modeling the static and dynamic aspects of processes.
- Many analysis techniques exist.
- Many extensions and variants have been defined over the years.

## Applications

- Applications in many different areas, such as databases, software engineering, formal semantics, etc.
- There are two main uses of Petri nets for workflows:
  - Specifications of workflows.
  - Formal foundation for workflows (semantics, analysis of properties).

# Petri Nets: Basics

- ⑩ A Petri Net takes the form of a **directed bipartite graph** where the nodes are either *places* or *transitions*.
- ⑩ **Places** represent **intermediate states** that may exist during the operation of a process. Places are represented by circles.
- ⑩ Places can be input/output of **transitions**. Transitions correspond to the **activities** or **events** of which the process is made up. Transitions are represented by rectangles or thick bars.
- ⑩ **Arcs** connect places and transitions in a way that places can only be connected to transitions and vice-versa.



place



transition

or

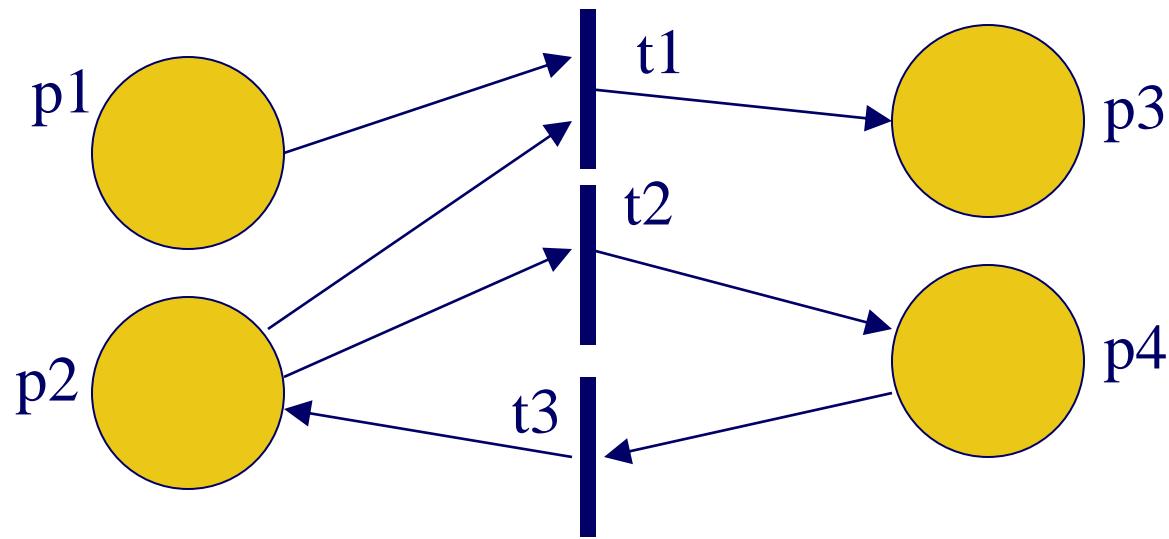


arc

# Petri Nets: Definition

- Formally a Petri net  $N$  is a triple  $(P, T, F)$  where
  - $P$  is a finite set of places
  - $T$  is a finite set of transitions where  $P \cap T = \emptyset$
  - $F \subseteq (P \times T \cup T \times P)$  is the set of arcs known as the **flow relation**
- A directed arc from a place  $p$  to a transition  $t$  indicates that  $p$  is an input place of  $t$ . Formally:
  - $\bullet t = \{p \in P \mid (p, t) \in F\}$
- A directed arc from a transition  $t$  to a place  $p$  indicates that  $p$  is an output place of  $t$ . Formally:
  - $t \bullet = \{p \in P \mid (t, p) \in F\}$
  - With an analogous meaning, we can define:
    - $p \bullet = \{t \in T \mid (p, t) \in F\}$  and  $\bullet p = \{t \in T \mid (t, p) \in F\}$

# Petri Net: Example



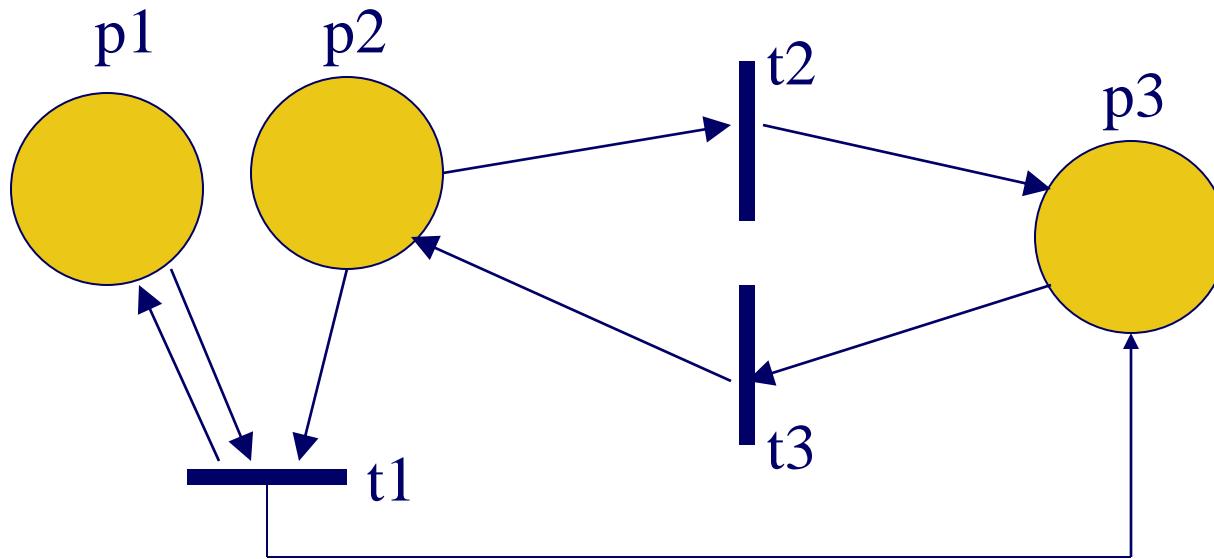
$$P = \{p_1, p_2, p_3, p_4\}$$

$$T = \{t_1, t_2, t_3\}$$

$$F = \{(p_1, t_1), (p_2, t_1), (t_1, p_3), (p_2, t_2), (t_2, p_4), (p_4, t_3), (t_3, p_2)\}$$

$$t_1 \bullet = \{p_3\}; \bullet t_1 = \{p_1, p_2\}; \bullet p_2 = \{t_3\}; \bullet p_1 = \emptyset; p_2 \bullet = \{t_1, t_2\}$$

# Petri Nets: Example



$P = \dots$

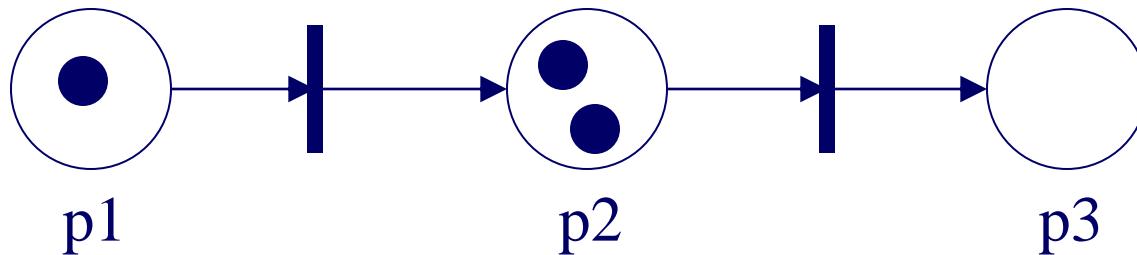
$T = \dots$

$F = \dots$

$t_1 \bullet = \dots ; \bullet t_1 = \dots ; \bullet p_2 = \dots ; p_2 \bullet =$   
 $\dots$

# Markings

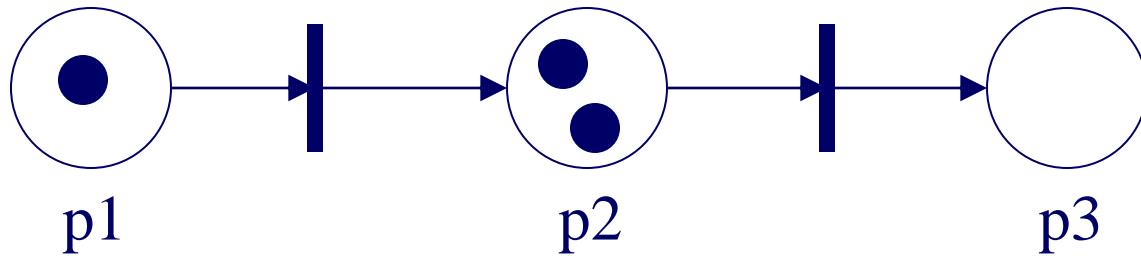
- The operational semantics of a Petri Net is described in terms of particular marks called *tokens* (graphically represented as black dots ●).
- Places in Petri Nets can contain any number of tokens. The distribution of tokens across all of the places in a net is called a marking. For a Petri net an *initial marking*  $M_0$  needs to be specified.
- Marking assigns tokens to places; formally, a marking  $M$  of a Petri net  $N = (P, T, F)$  is a function  $M: P \rightarrow \text{NAT}$ .



- The marking below is formally captured by the following marking  $M = \{(p_1, 1), (p_2, 2), (p_3, 0)\}$ .

# State of a Petri Net

- A state can be compactly described as shown in the following example:
  - $1p_1 + 2p_2 + 0p_3$  is the state with one token in place  $p_1$ , two tokens in  $p_2$  and no tokens in  $p_3$ .
  - We can also represent this state in the following (equivalent) way:  $p_1 + 2p_2$ .



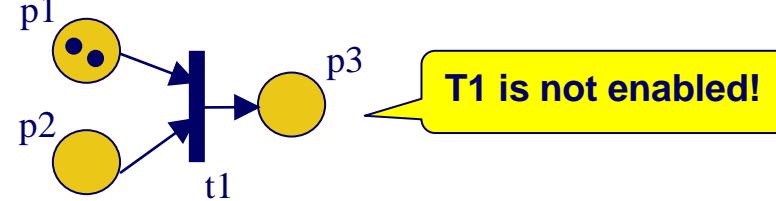
- We can also describe an ordering function  $\geq$  over the set of possible states such that, given a Petri net  $N = (P, T, F)$  and markings  $M$  and  $M'$ ,  $M \geq M'$  iff for all  $p$  in  $P$ :  $M(p) \geq M'(p)$ .  $M > M'$  iff  $M \geq M'$  and  $M \neq M'$ .

# Enabled Transitions

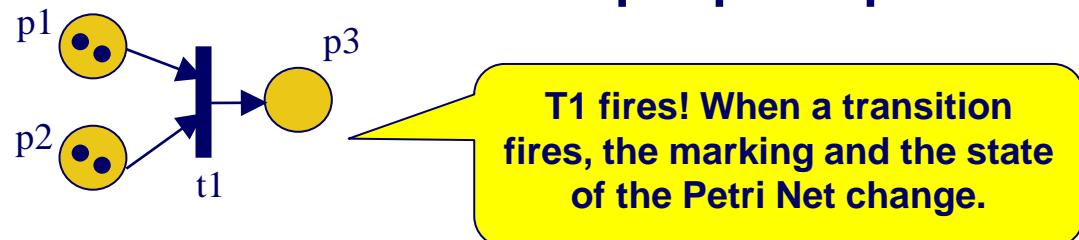
- The operational semantics of Petri nets are characterized by the notion of a transition executing or “firing”. A transition in a Petri net can “fire” whenever there are one or more tokens in each of its input places.
- The execution of a transition occurs in accordance with the following firing rules:

1. A transition  $t$  is said to be enabled if and only if each input place  $p$  of  $t$  contains at least one token. Only enabled transitions may fire.

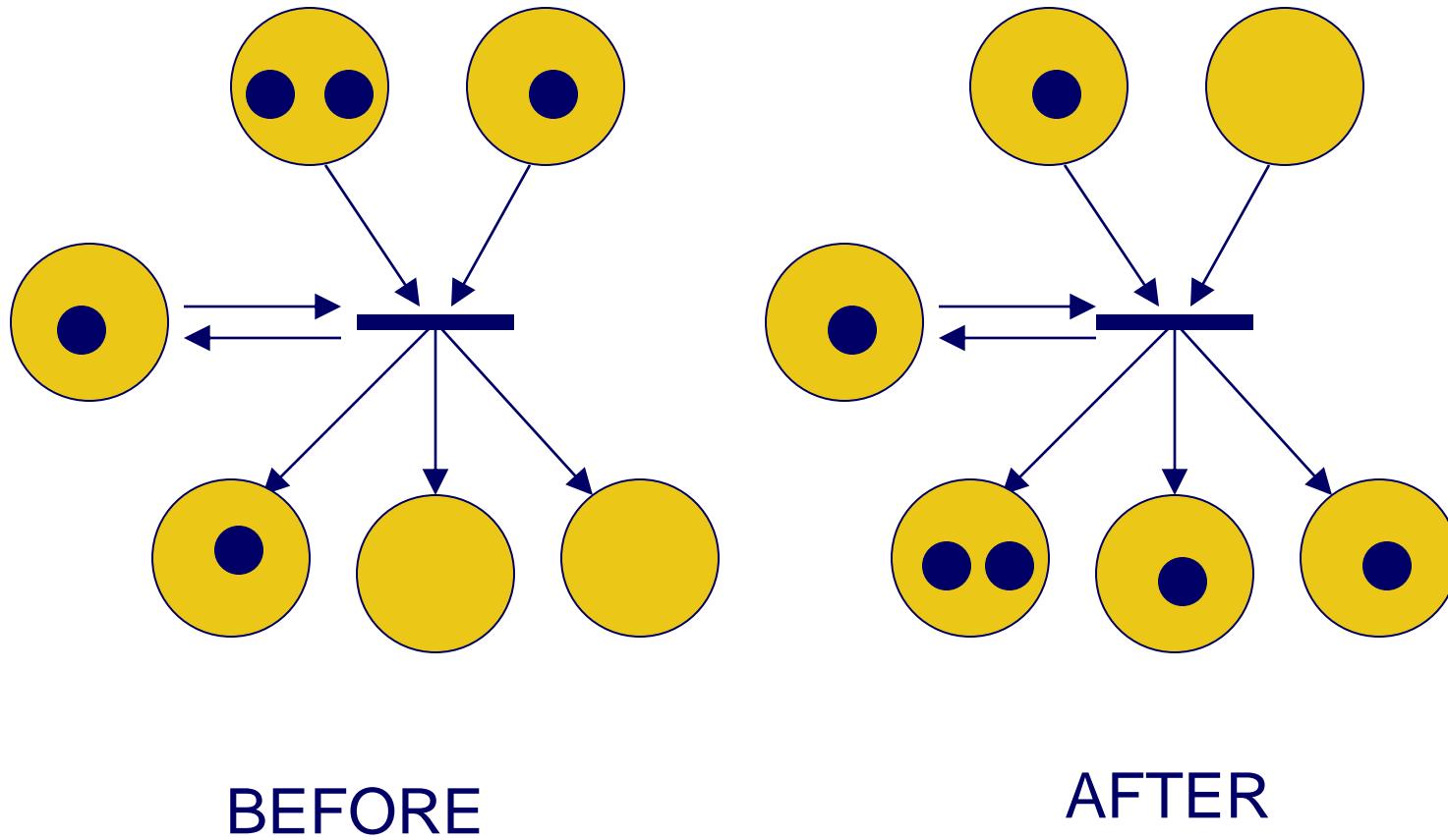
- Formally, a transition  $t$  is enabled in a marking  $M$  iff for each  $p$ , with  $p \in \bullet t$ ,  $M(p) > 0$ . (see definition 2.7 of [DE95])



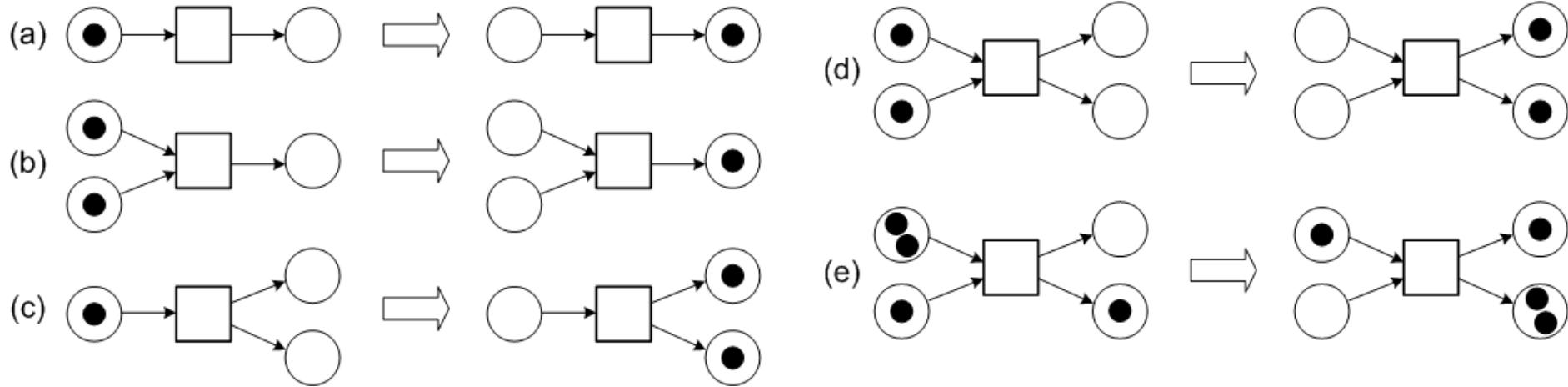
2. If transition  $t$  fires, then  $t$  consumes one token from each input place  $p$  of  $t$  and produces one token for each output place  $p$  of  $t$ .



# Firing a Transition: Example



# Firing Transitions: Further Examples



- ⑩ It is assumed that the firing of a transition is an atomic action that occurs instantaneously and cannot be interrupted.
- ⑩ If there are multiple enabled transitions, any one of them may fire; however, for execution purposes, it is assumed that **they cannot fire simultaneously**.
- ⑩ An enabled transition is not forced to fire immediately but can do so at a time of its choosing.
- ⑩ These features make Petri nets particularly suitable for modeling concurrent process executions.

# Firing Transitions

- Given a Petri Net ( $P, T, F$ ) and an initial state  $M$ , we have the following notations that characterize the firing of a given transition  $t$ :

- $M \rightarrow^t M'$  indicates that if transition  $t$  is enabled in state  $M$ , then firing  $t$  in  $M$  results in state  $M'$ . Formally, notation  $M \rightarrow^t M'$ , is defined by:

- $M'(p) = M(p)$  if  $p \notin t^\bullet \cup t^\circ$  or  $p \in t^\bullet \cap t^\circ$
- $M'(p) = M(p) - 1$  if  $p \in t^\bullet$  and  $p \notin t^\circ$
- $M'(p) = M(p) + 1$  if  $p \in t^\circ$  and  $p \notin t^\bullet$

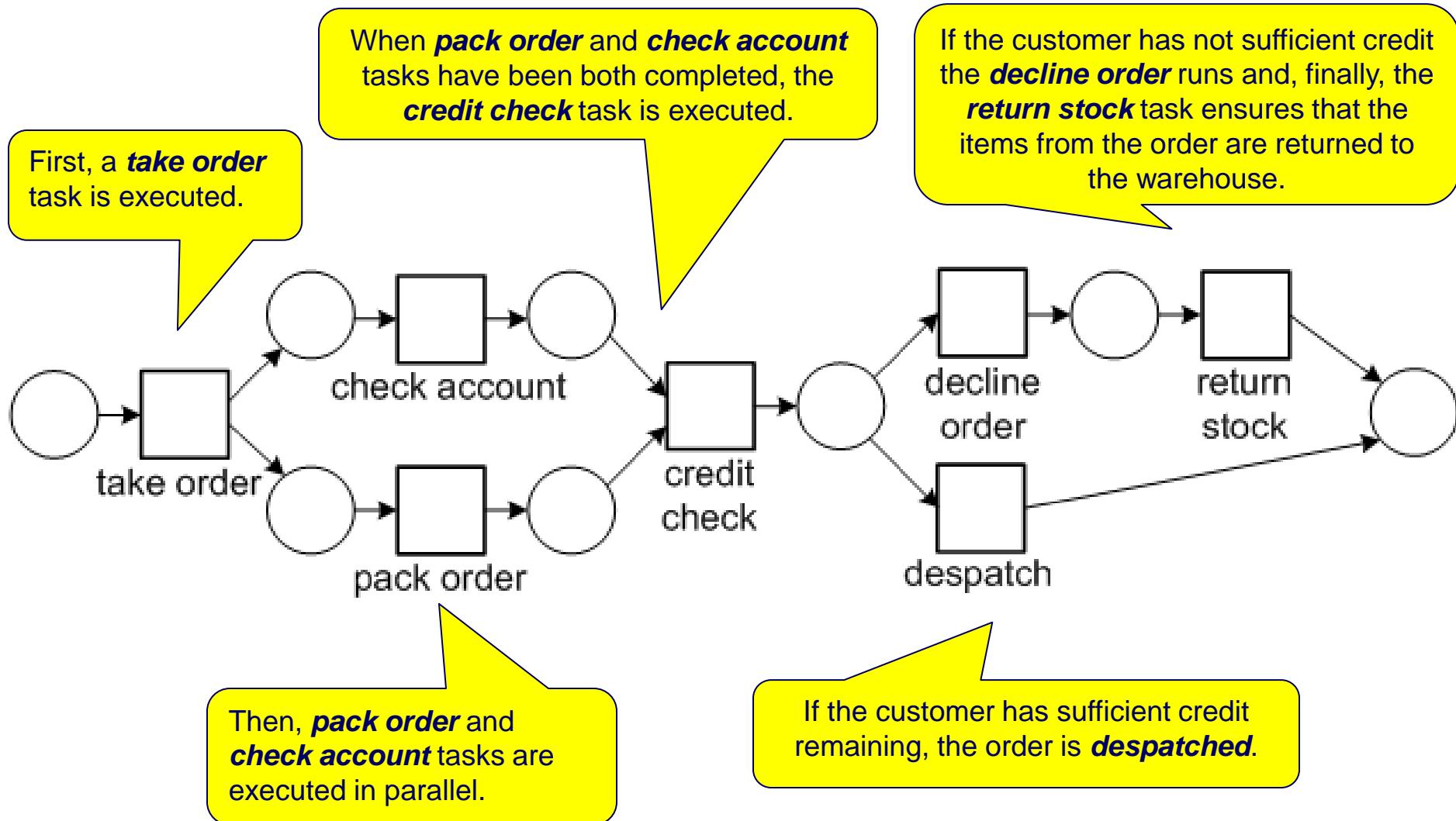
- $M \rightarrow M'$  indicates that there is a transition  $t$  such that  $M \rightarrow^t M'$ .
- $M \rightarrow^\sigma M'$  denotes the firing sequence  $\sigma = t_0 t_1 \dots t_{n-1}$  that leads from state  $M$  to state  $M'$ , such that

$$M = M_0 \rightarrow^{t_0} M_1 \rightarrow^{t_1} M_2 \dots M_{n-1} \rightarrow^{t_{n-1}} M_n = M'$$

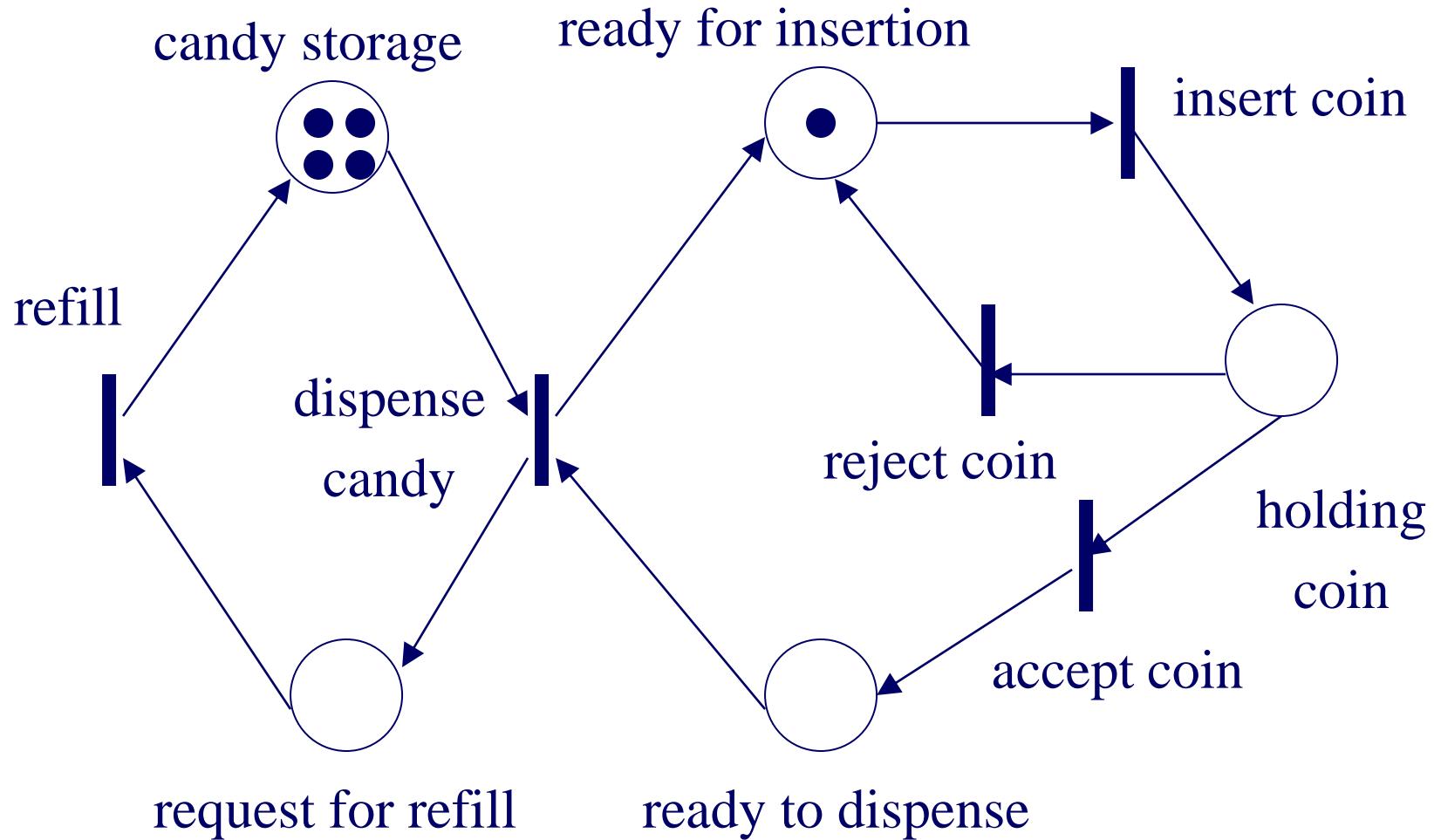
Note that the transitions do not have to be different!

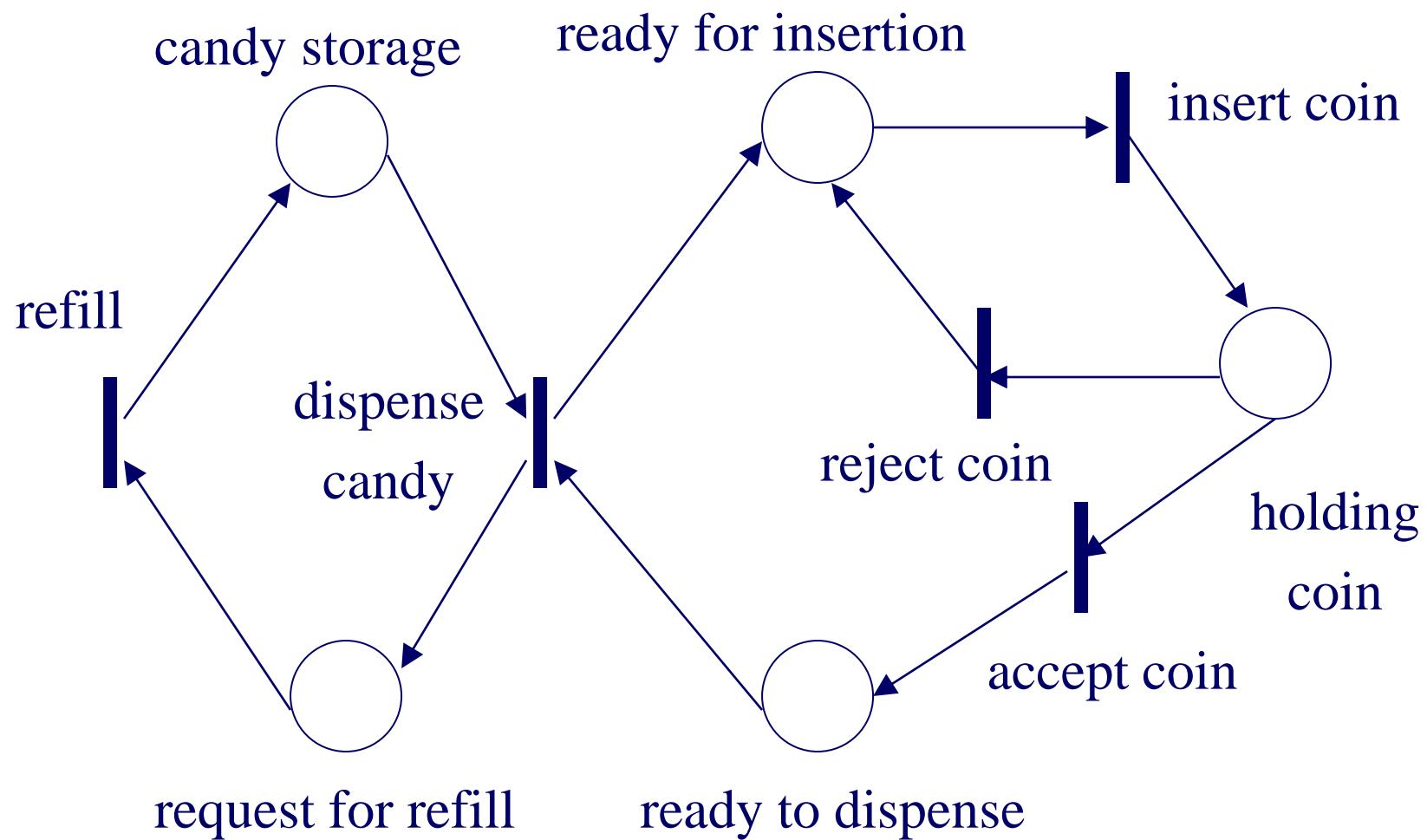
- A state  $M'$  is called **reachable** from state  $M$  (we write  $M \rightarrow^* M'$ ) iff there is a firing sequence  $\sigma$  that leads from state  $M$  to state  $M'$ .
  - Informally, a marking is reachable from another marking if there is a sequence of transitions that can fire from the first marking to arrive at the second marking.

# Petri nets: Order Fulfillment Example



# Petri nets: Example of a vending machine (source [DE95] p. 4)





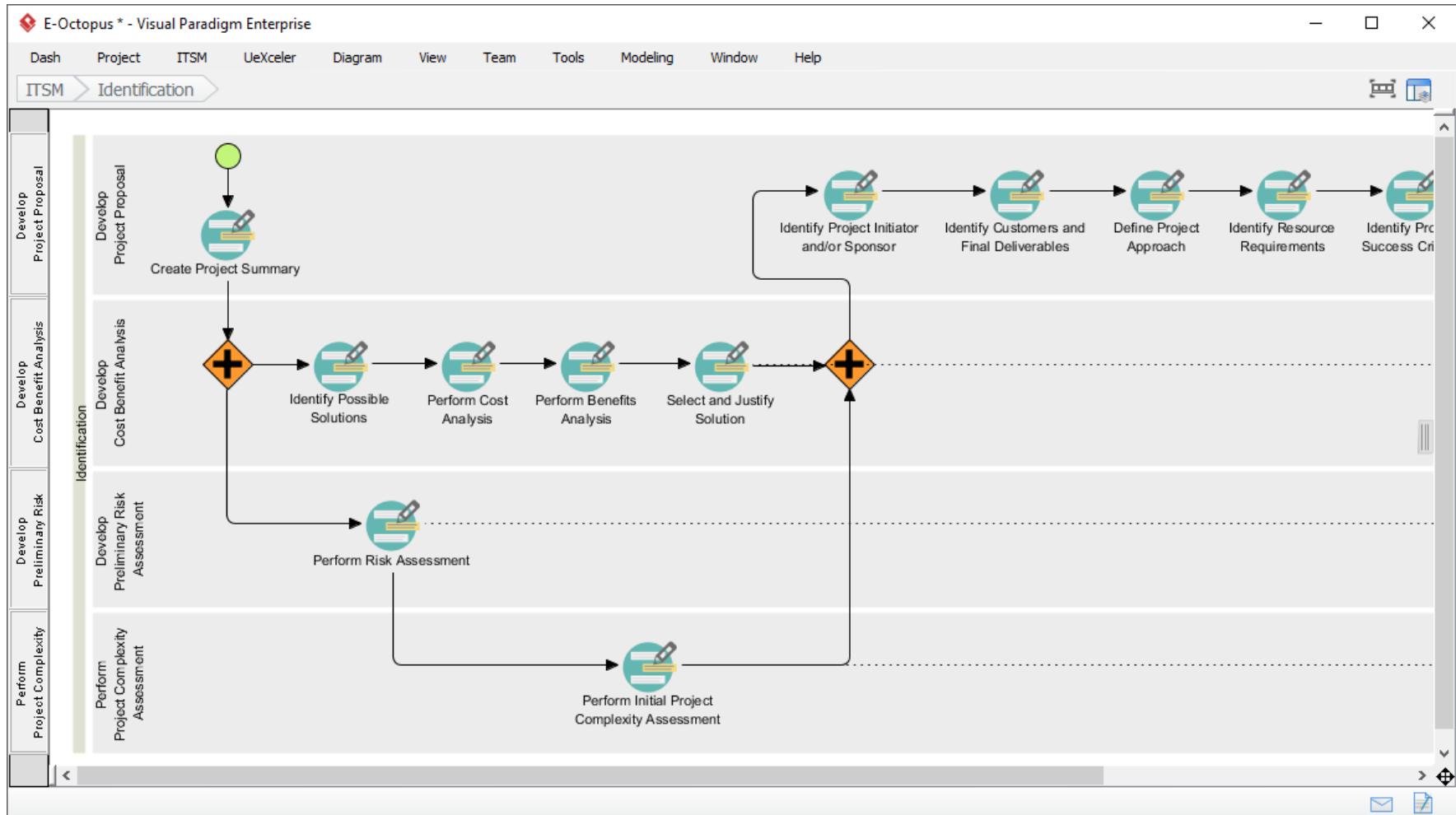
# Purposes of process modeling

- facilitates **understanding** and communication by providing a shared view of the process
- supports **management** and improvement; it can be used to assign tasks, track progress, and identify trouble spots
- serves as a basis for **automated** support (usually not fully automatic)

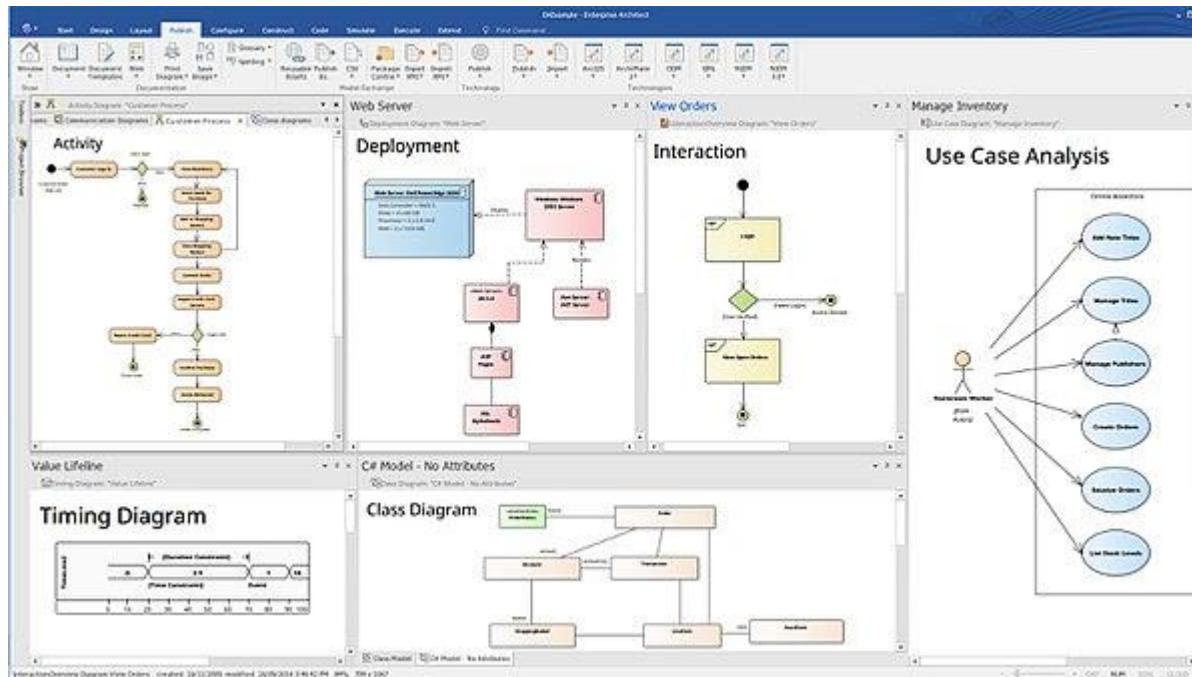
# Caveats of process modeling

- **not all aspects of software development can be caught in an algorithm**
- **a model is a model, thus a simplification of reality**
- **progression of stages differs from what is actually done**
- **some processes (e.g. learning the domain) tend to be ignored**
- **no support for transfer across projects**

# Tool example 1 – visual paradigm



# Sparx Systems Enterprise Architect



# Exam alert

- **Formalizza almeno una parte di processo in un formalismo di tua scelta**

# Summary

- **Traditional models focus on *control* of the process**
- **There is no one-size-fits-all model; each situation requires its own approach**
- **A pure project approach inhibits reuse and maintenance**
- **There has been quite some attention for process modeling, and tools based on such process models**

# Configuration Management

## CAPITOLO 4

### Main issues:

- manage items during software life cycle
- usually supported by powerful tools

# TASKS AND RESPONSIBILITIES

- **baseline:** one official version of the complete set of artifacts related to the project
- items contained in the baseline are the **configuration items**
  - source code components,
  - the requirements specification,
  - the design documentation,
  - the test plan, test cases, test results,
  - the user manual.
- **Initially the baseline will contain a requirements specification. As time goes on, elements will be added: design documents, source code components, test reports, etc.**

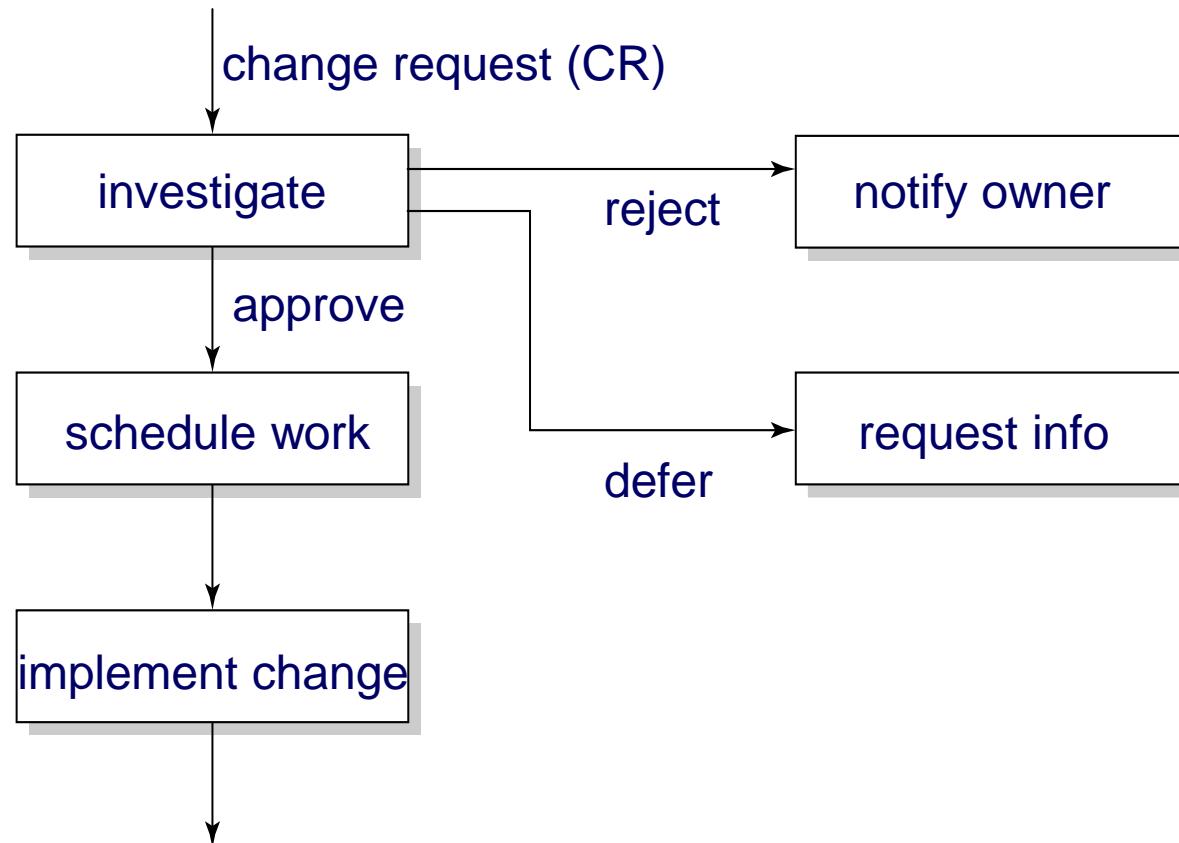
# Configuration management tasks

- Any proposed change to the baseline is called a change request.
- managing changes and making configuration items available during the software life cycle, usually through a **Configuration Control Board** (CCB)
- keeping track of the **status** of all items (including the change requests)
- **crucial for large projects**

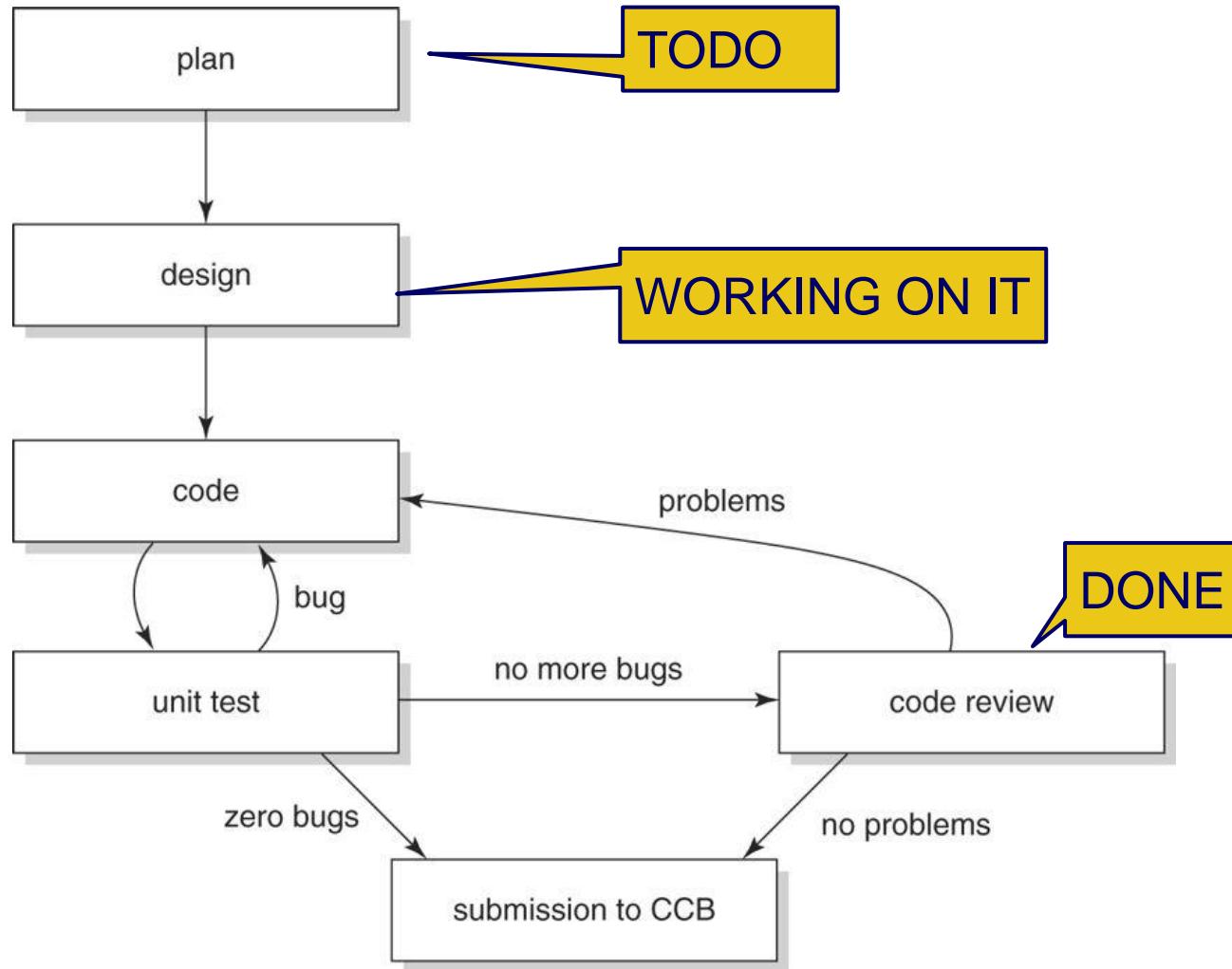
# Configuration Control Board

- ensures that every change to the baseline (change request - CR) is properly authorized and executed
- CCB needs certain information for every CR, such as who submits it, how much it will cost, urgency, etc
- CCB assesses the CR. If it is approved, it results in a work package which has to be scheduled.
- so, configuration management is not only about keeping track of changes, but also about workflow management

# Workflow of a change request



# development activities (example)

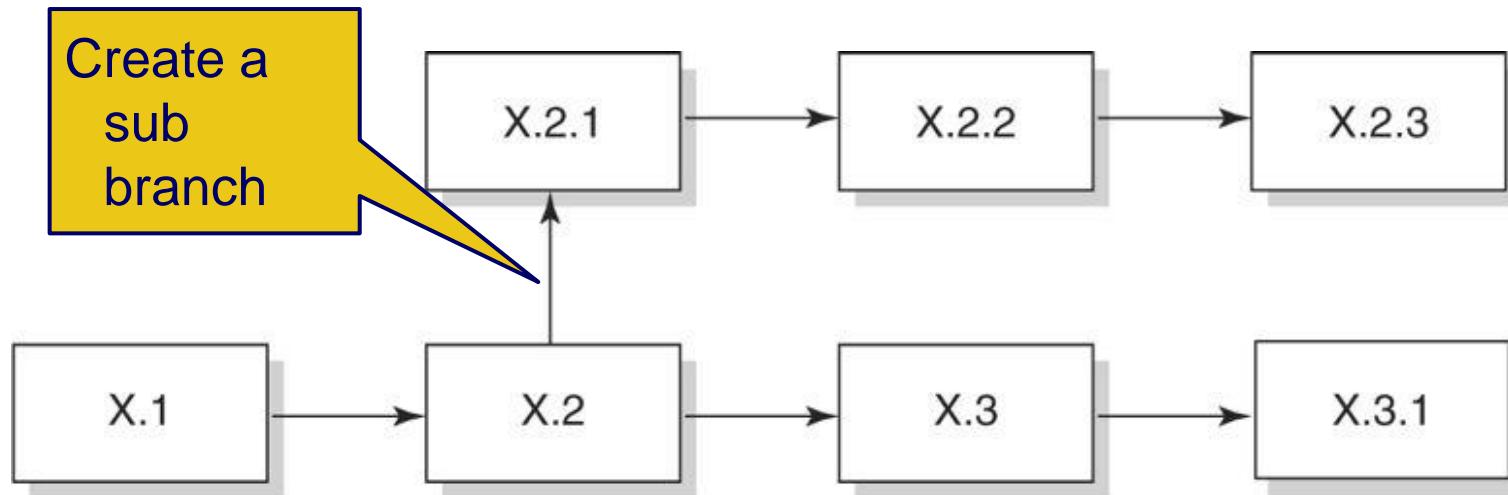


# Tool support for configuration management

- if an item has to be changed, one person gets a copy thereof, and meanwhile it is locked to all others
- new items can only be added to the baseline after thorough testing
- changes in the status of an item (e.g. code finished) trigger further activities (e.g. start unit testing)
- old versions of a component are kept as well, resulting in versions, like X.1, X.2, ...
- we may even create different branches of revisions: X.2.1, X.2.2, ... and X.3.1, X.3.2, ...

# Working in parallel

- When working in parallel



# Functionalities of SCM tools

- Components (storing, retrieving, accessing, ...)
- Structure (representation of system structure)
- Construction (build an executable)
- Auditing (follow trails, e.g. of changes)
- Accounting (gather statistics)
- Controlling (trace defects, impact analysis)
- Process (assign tasks)
- Team (support for collaboration)

# Models of configurations

- **version-oriented:** physical change results in a new version, so versions are characterized by their difference, i.e. **delta**
- **change-oriented:** basic unit in configuration management is a logical change
- **identification of configuration becomes different:** “baseline X plus fix table bug” instead of “X3.2.1 + Y2.7 + Z3.5 + ...”

# Evolution of SCM tools

- Early tools: emphasis on product-oriented tasks
- Nowadays: support for other functionalities too.  
They have become a (THE) major tool in large,  
multi-site projects
- Agile projects: emphasis on running system: *daily  
builds*

# Configuration Management Plan

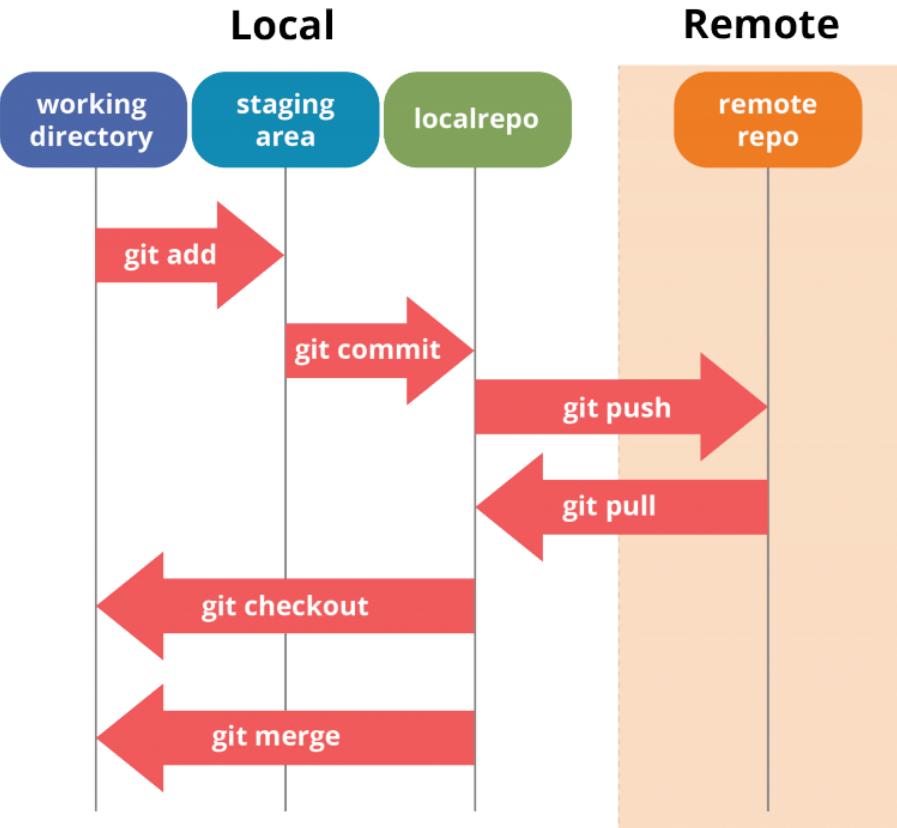
- **Management section: organization, responsibilities, standards to use, etc**
- **Activities: identification of items, keeping status, handling CRs**

# Example of configuration management plan

Part	Description
1. Introduction	Purpose, scope, main terms and references
2. SCM Management	Organization, responsibilities and authorities
3. SCM Activities	Identification of configuration items, control, status accounting, and auditing process
4. SCM Schedules	Coordination with other activities
5. SCM Resources	Tools, human and computer resources
6. SCM Plan Maintenance	How the plan is kept up to date

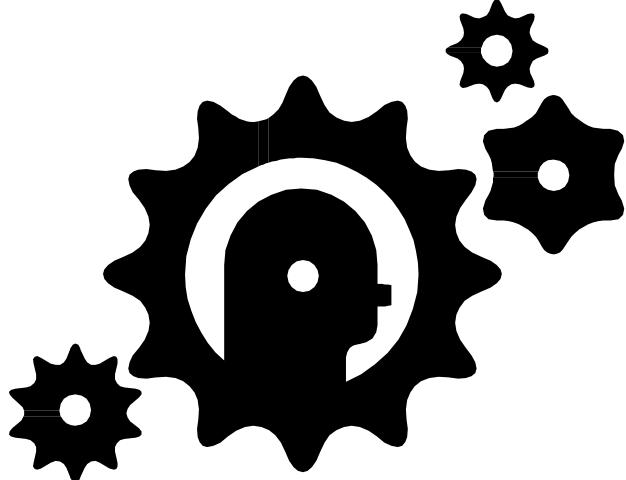
Exam alert

# Example: git (github) – prossimo venerdì



- **add**
- **commit**
- **push**
- **pull**
  
- **checkout**
- **merge**

# Summary



- CM is about managing all kinds of artifacts during software development
- Crucial for large projects
- Supported by powerful tools

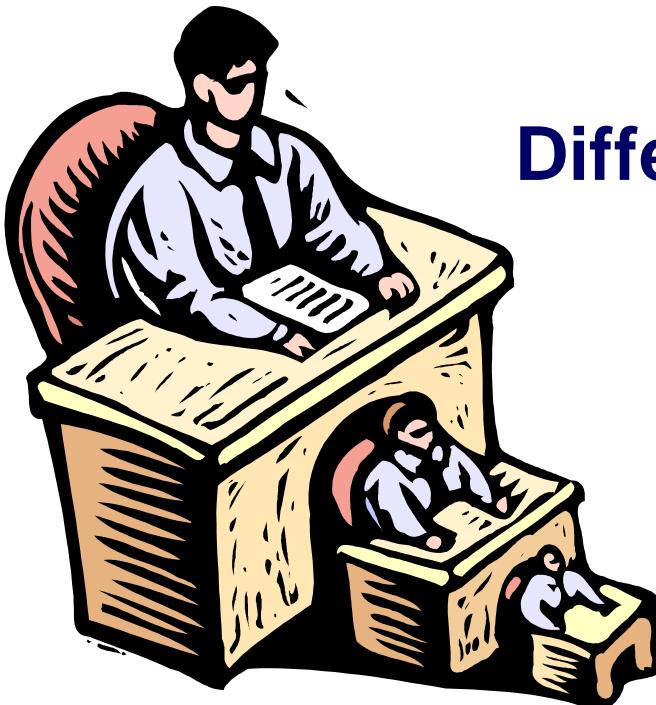
# People Management, People Organization

## CAPITOLO 5

### Main issues:

- **People** are key in software development
- Different ways to **organize** SD projects

# Different ways to organize people



# **People management**

- People have different goals**
- People and productivity**
- Group processes**
- Coordination of work**
- Importance of informal communication**

# Mintzberg's coordination mechanisms

- **Simple: direct supervision**
- **Machine bureaucracy: standardization of work processes**
- **Divisionalized form: standardization of work products**
- **Professional bureaucracy: standardization of worker skills**
- **Adhocracy: mutual adjustment**

# Mintzberg's

## ▪ Simple structure

- there may be one or a few managers and a core of people who do the work. The corresponding coordination mechanism is called *direct supervision*. This configuration is often found in new, relatively small organizations. There is little specialization, training and formalization. Coordination lies with separate people, who are responsible for the work of others.

## ▪ Machine bureaucracy

- When the content of the work is completely specified, it becomes possible to execute and assess tasks on the basis of precise instructions. Mass-production and assembly lines are typical examples of this configuration type. There is little training and much specialization and formalization. The coordination is achieved through *standardization of work processes*.

## ▪ Divisionalized form

- each division (or project) is granted considerable autonomy as to how the stated goals are to be reached. The operating details are left to the division itself. Coordination is achieved through *standardization of work outputs*. Control is executed by regularly measuring the performance of the division. This coordination mechanism is possible only when the end result is specified precisely.

# Mintzberg's

- **Professional bureaucracy**

- If it is not possible to specify either the end result or the work contents, coordination can be achieved through standardization of worker skills. In a professional bureaucracy, skilled professionals are given considerable freedom as to how they carry out their jobs. Hospitals are typical examples of this type of configuration.

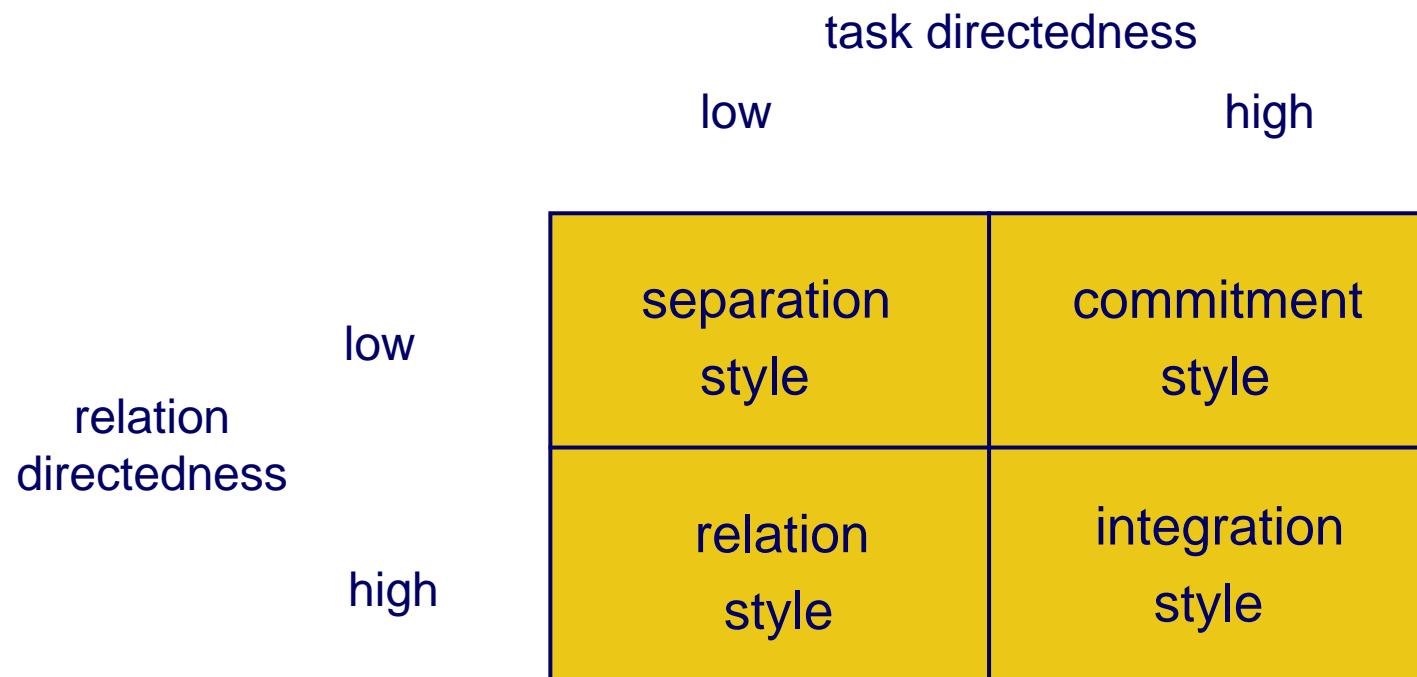
- **Adhocracy**

- In projects that are big or innovative in nature, work is divided amongst many specialists. We may not be able to tell exactly what each specialist should do, or how they should carry out the tasks allocated to them. The project's success depends on the ability of the group as a whole to reach a non-specified goal in a non-specified way. Coordination is achieved through mutual adjustment.

# External and Internal forces

- **Example context: a complex software development project in a new, not yet explored area, within a government agency**
- **External force: the bureaucratic context is likely to want to push a bureaucratic type of organization, with bosses, and hierarchical decision procedures**
- **Internal force: the project really requires a more democratic, consensus-based type of organization**

# Reddin's management styles



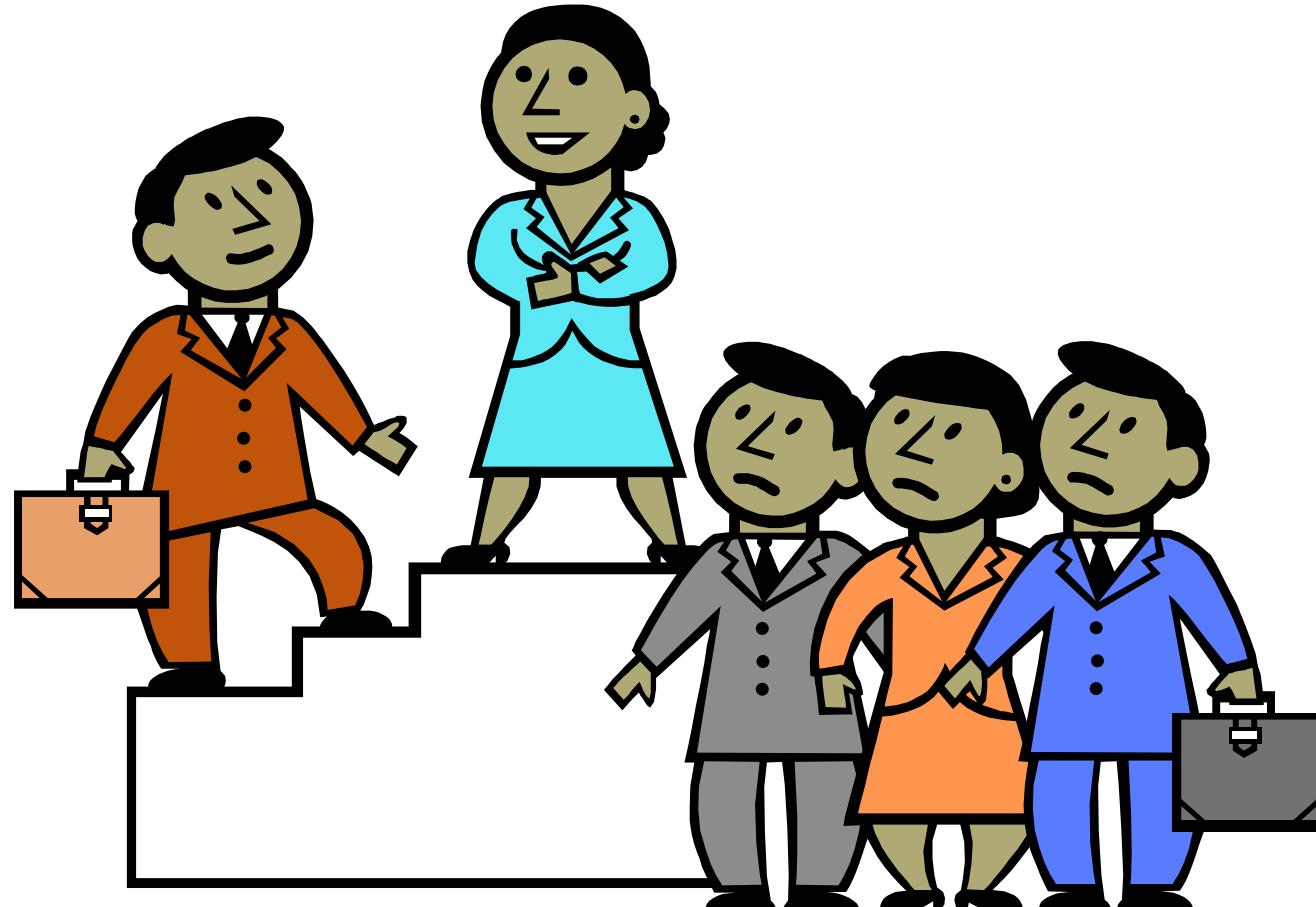
# Focus

- In both these schemes, we look from the manager to the team.
- We may also take the opposite position, and consider the relation and task *maturity* of individual team members.
- The manager should align his dealings with team members with their maturity.

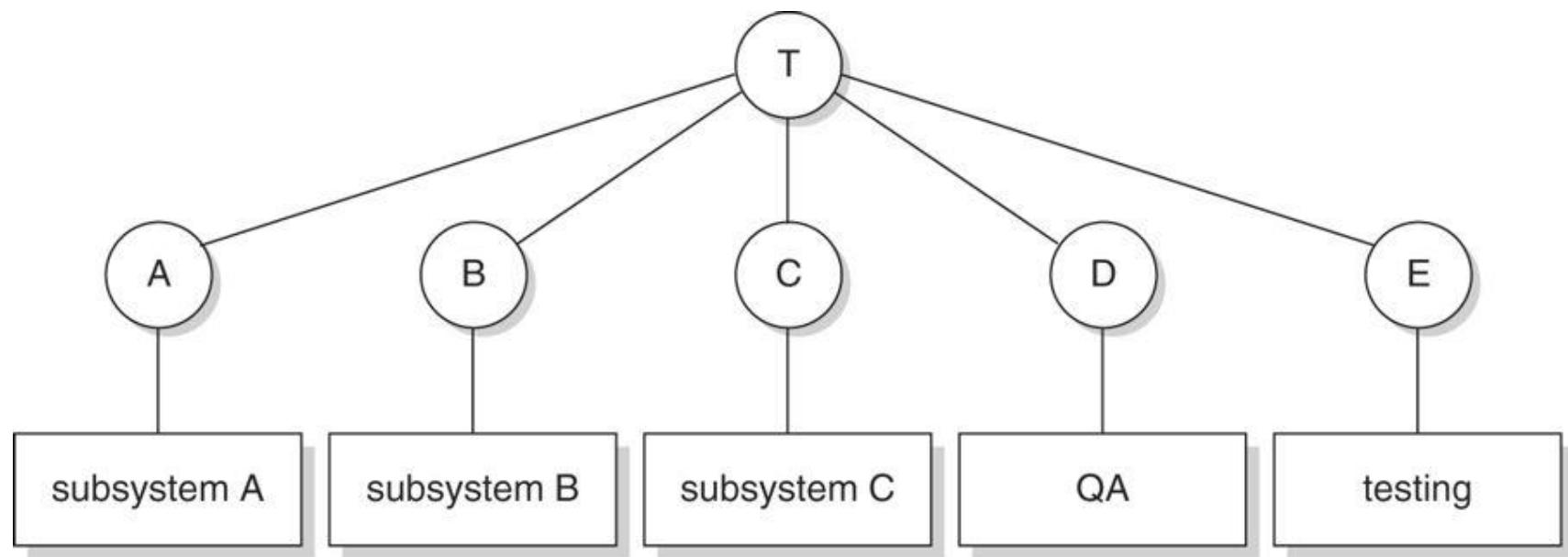
# Team Organization

- **Hierarchical organization**
- **Matrix organization**
- **Chief programmer team**
- **SWAT team**
- **Agile team/Extreme Programming (XP)**
- **Open Source Development**

# Hierarchical team



# Hierarchical team for SW



# Matrix organization



	Real-time program ming	Graphics	Database s	QA	Testing
Project C	X			X	X
Project B	X		X	X	X
Project A		X	X	X	X

# Chief programmer team



*Napoleon*



# Skilled worker with advanced tools (SWAT)



# **SWAT team**

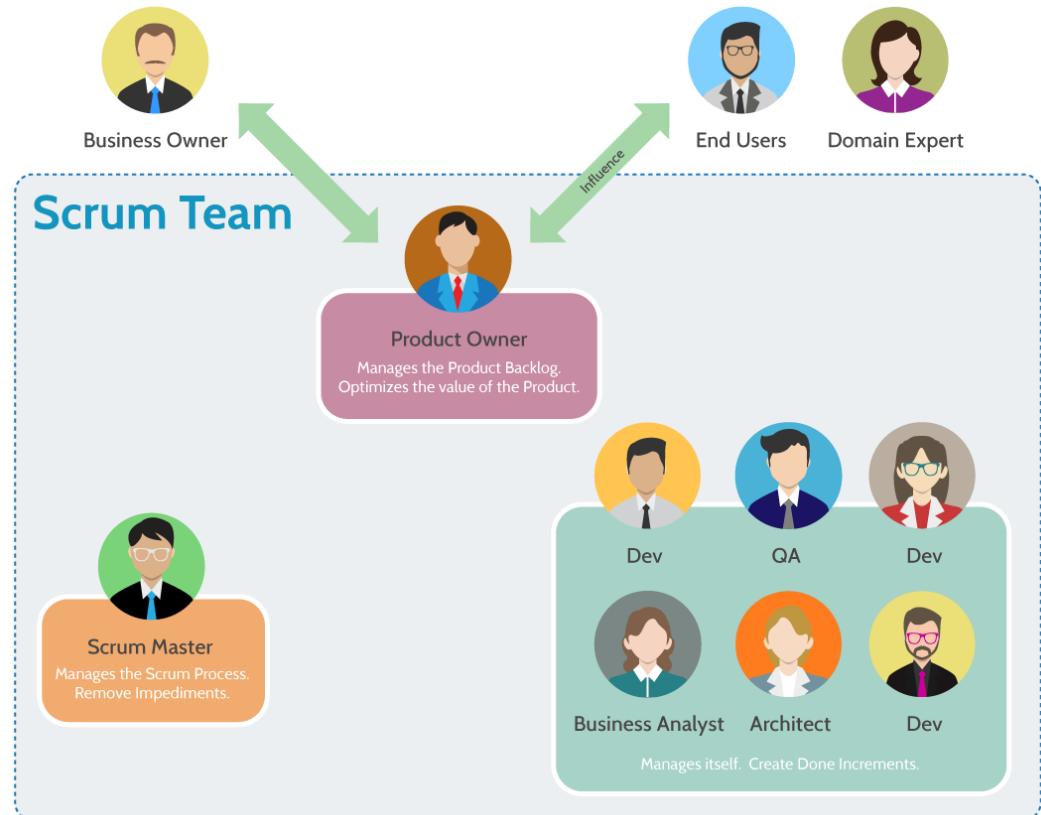
- A SWAT team is relatively small.
- It typically has four or five members.
- occupies one room.
- Communication channels are kept very short. The team does not have lengthy formal meetings with formal minutes. Rather, it uses workshops and brainstorming sessions of which little more than a snapshot of a white-board drawing is retained.

# Agile team



# SCRUM team

- **Product Owner**
- **Scrum Master, and**
- **Development team**



# A Scrum Team

- **typically between five and nine members**
- **working together to deliver the required product increments.**
- **The Scrum framework encourages a high level of communication among team members, so that the team can:**
  - Follow a common goal
  - adhere the same norms and rules
  - show respect to each other

# Product Owner

- **knows what the customer wants and the relative business**
- **He or she can then translate the customer's wants and values back to the Scrum team.**
- **must know the business case for the product and what features the customers' wants.**
- **He must be available to consult with the team to make sure they are correctly implementing the product vision.**
- **the authority to make all decisions necessary to complete the project**
- **is responsible for managing the Product Backlog which includes:**
  - Expressing Product Backlog items clearly
  - Ordering the Product Backlog items to best achieve goals and missions.
  - Optimizing the value of the work the Team performs.
  - Ensuring that the Product Backlog is visible, transparent, and clear to all, and shows what the Team will work on further.
  - Ensuring that the Team understands items in the Product Backlog to the level needed.

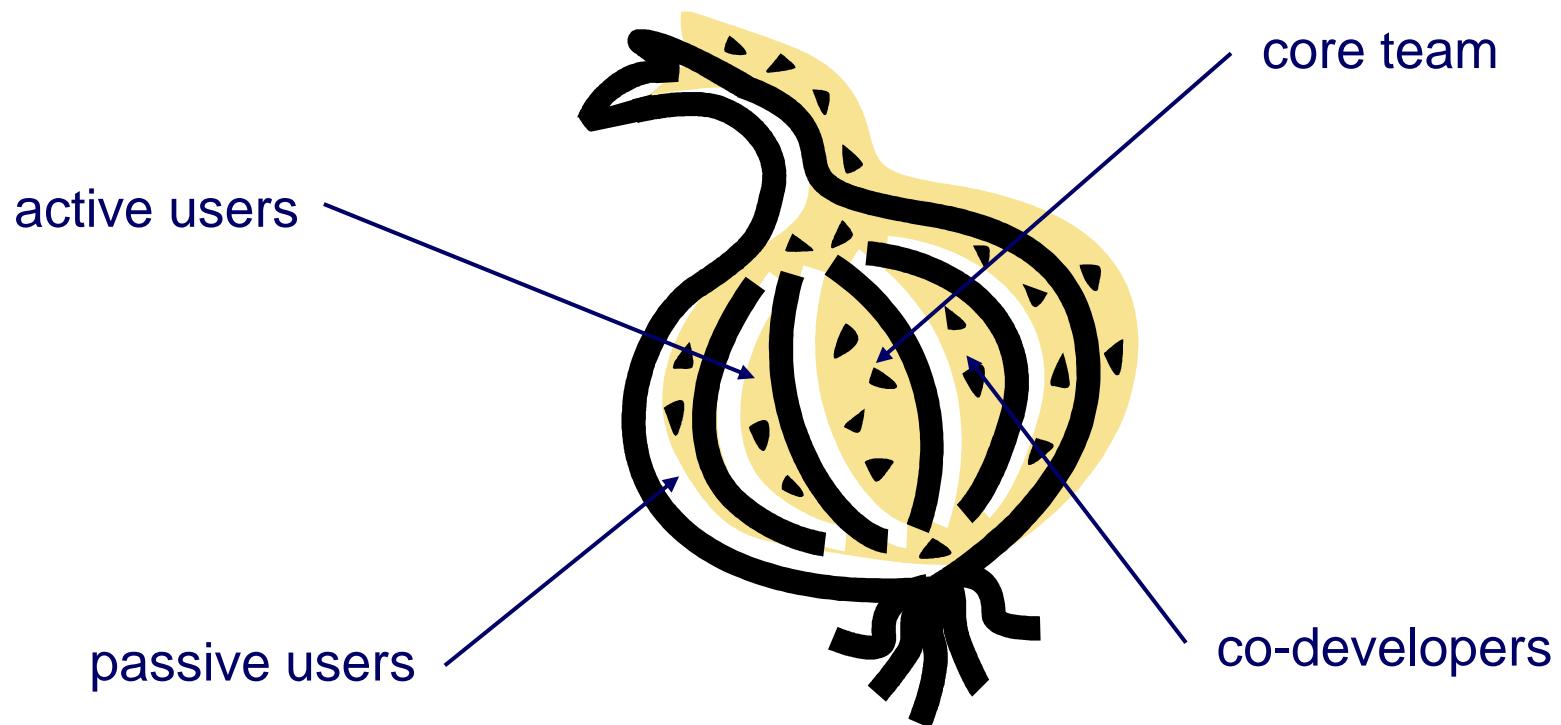
# Scrum Master

- **The scrum master help to keep the team accountable to their commitments to the business and also remove any roadblocks that might impede the team's productivity. They met with the team on a regular basis to review work and deliverables, most often in a weekly cadence. The role of a scrum master is to coach and motivate team member, not enforce rules to them.**  
**The role of a scrum master includes:**
  - **ensure the process run smoothly**
  - **remove obstacles that impact productivity**
  - **organize the critical events and meeting**

# Development Team

- **Development Teams are structured and empowered by the organization to organize and manage their own work. The resulting synergy optimizes the Development Team's overall efficiency and effectiveness.**
  - They are self-organizing. No one (not even the Scrum Master) tells the Development Team how to turn Product Backlog into Increments of potentially releasable functionality;
  - are cross-functional, with all the skills as a team necessary to create a product Increment;
  - no titles for Development Team members, regardless of the work being performed by the person;
  - no sub-teams in the Development Team, regardless of domains that need to be addressed like testing, architecture, operations or business analysis; and,
  - Individual Development Team members may have specialized skills and areas of focus, but accountability belongs to the Development Team as a whole.

# Open Source Software Development



# **Some general rules**

- **Use fewer, and better, people**
- **Fit tasks to people**
- **Help people to get the most out of themselves**
- **Look for a well-balanced team**
- **If someone doesn't fit the team: remove him**

# **Summary**

- Software is written by humans**
- Coordination issues/management styles**
- Common team organizations in software development:**
  - Hierarchical team
  - Matrix organization
  - Agile team
  - Open source development

# **Managing Software Quality**

## **Capitolo 6**

### **Main issues:**

- Quality cannot be added as an afterthought**
- To measure is to know**
- Product quality vs process quality**

# Commitment to quality pays off



# Approaches to quality

- **Quality of the product versus quality of the process**
- **Check whether (product or process) *conforms to* certain norms**
- **Improve quality by improving the product or process**

# Approaches to quality

	Conformance	Improvement
Product	ISO 9126	'best practices'
Process	ISO 9001 SQA	CMM SPICE Bootstrap

Not discussed in this chapter

## **ON MEASURES AND NUMBERS**

# What is quality?



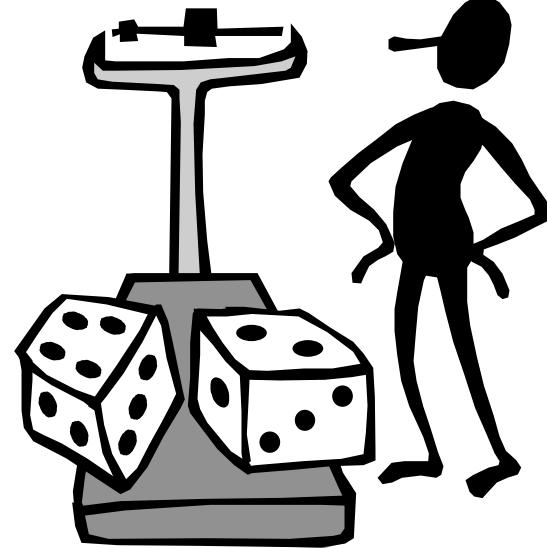
software

+



measures

# How to measure “complexity”?



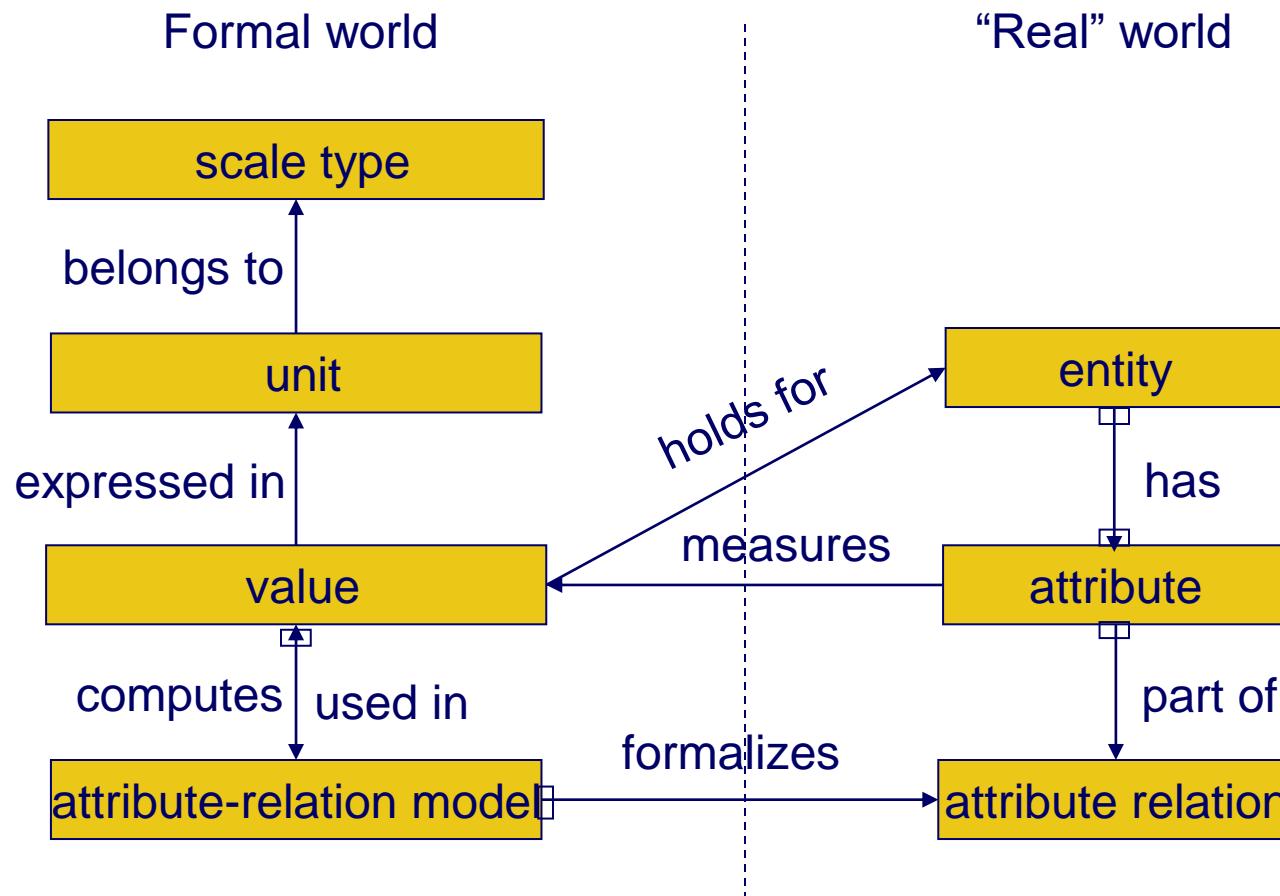
- The length of the program?
- The number of goto's?
- The number of if-statements?
- The sum of these numbers?
- Yet something else?

# Which is better?

```
procedure bubble
  (var a: array [1..n] of integer; n: integer);
  var i, j, temp: integer;
  begin
    for i:= 2 to n do
      j:= i; (a)
    while j > 1 and a[j] < a[j-1] do
      temp:= a[j];
      a[j]:= a[j-1];
      a[j-1]:= temp;
      j:= j-1;
    enddo
  enddo
end;
```

```
procedure bubble
  (var a: array [1..n] of integer; n: integer);
  var i, j, temp: integer;
  begin
    for i:= 2 to n do
      if a[i] >= a[i-1] then goto next endif;
      j:= i;
      loop: if j <= 1 then goto next endif;
      if a[j] >= a[j-1] then goto next endif;
      temp:= a[j];
      a[j]:= a[j-1];
      a[j-1]:= temp;
      j:= j-1;
      goto loop;
    next: skip;
  enddo
end;
```

# A measurement framework



# Scale types

- **Nominal:** just classification
- **Ordinal:** linear ordering ( $>$ )
- **Interval:** like ordinal, but interval between values is the same (so average has a meaning)
- **Ratio:** like interval, but there is a 0 (zero) (so A can be twice B)
- **Absolute:** counting number of occurrences



# Representation condition

- A measure  $M$  is *valid* if it satisfies the representation condition, i.e. if  $A > B$  in the real world, then  $M(A) > M(B)$
- E.g. if we measure complexity as the number of if-statements, then:
  - Two programs with the same number of if-statements are equally complex
  - If program A has more if-statements than program B, then A is more complex than B

# More on measures

- ***Direct versus indirect measures***
- ***Internal versus external attributes***
  - External attributes can only be measured indirectly
  - Most quality attributes are external
- **Scale type of a combined measure is the ‘weakest’ of the scale types of its constituents**
  - This is often violated; see cost estimation models

## **6.2 A TAXONOMY OF QUALITY ATTRIBUTES**

# Quality attributes (McCall)

## Product operation: to the use of the software after it has become operational

<b>Correctness</b>	The extent to which a program satisfies its specifications and fulfills the user's mission objectives.
<b>Reliability</b>	The extent to which a program can be expected to perform its intended function with required precision.
<b>Efficiency</b>	The amount of computing resources and code required by a program to perform a function.
<b>Integrity</b>	The extent to which access to software or data by unauthorized persons can be controlled.
<b>Usability</b>	The effort required to learn, operate, prepare input, and interpret output of a program.

# Quality attributes (McCall) (2)

## Product revision: the maintainability of the system

Maintainability	The effort required to locate and fix an error in an operational program.
Testability	The effort required to test a program to ensure that it performs its intended function.
Flexibility	The effort required to modify an operational program.

# Quality attributes (McCall) (3)

**ease with which a transition to a new environment can be made.**

Portability	The effort required to transfer a program from one hardware or software environment to another.
Reusability	The extent to which a program (or parts thereof) can be reused in other applications.
Interoperability	The effort required to couple one system with another.

# Quality attributes (McCall) – 3 levels

## ▪ Product operation

- Correctness does it do what I want?
- Reliability does it do it accurately all of the time?
- Efficiency will it run on my hardware as well as it can?
- Integrity is it secure?
- Usability can I use it?

## ▪ Product revision

- Maintainability can I fix it?
- Testability can I test it?
- Flexibility can I change it?

## ▪ Product transition

- Portability will I be able to use it on another machine?
- Reusability will I be able to reuse some of the software?
- Interoperability will I be able to interface it with another system?

# Taxonomy of quality attributes (ISO 9126)

- **Functionality**
- **Reliability**
- **Usability**
- **Efficiency**
- **Maintainability**
- **Portability**

The ISO quality characteristics strictly refer to a software *product*. Their definitions do not capture *process* quality issues.

# ISO 9126

Characteristic	Sub-characteristics
<b>Functionality</b>	Suitability, Accuracy, Interoperability, Security, Functionality compliance
<b>Reliability</b>	Maturity, Fault tolerance, Recoverability, Reliability compliance
<b>Usability</b>	Understandability, Learnability, Operability, Attractiveness, Usability compliance
<b>Efficiency</b>	Time behavior, Resource utilization, Efficiency compliance
<b>Maintainability</b>	Analyzability, Changeability, Stability, Testability, Maintainability compliance
<b>Portability</b>	Adaptability, Installability, Co-existence, Replaceability, Portability compliance

## Table 6.7

## ISO 9126 (cnt'd)

- ISO 9126 measures ‘quality in use’: the extent to which users can achieve their goal
- Quality in use is modeled in four characteristics:
  - Effectiveness
  - Productivity
  - Safety
  - Satisfaction

# Perspectives on quality (6.3 – SKIP)



- **Transcendent (“I really like this program”)**
- **User-based (“fitness for use”)**
- **Product-based (based on attributes of the software)**
- **Manufacturing-based (conformance to specs)**
- **Value-based (balancing time and cost vs profits)**

# **6.4 THE QUALITY SYSTEM**

**ISO 9000**

# ISO 9001

- ISO 9001 is a generic standard that can be applied to any product.
- **Model for quality assurance in design, development, production, installation and servicing**
- **Basic premise: confidence in product conformance can be obtained by adequate demonstration of supplier's capabilities in processes (design, development, ...)**
- **ISO registration by an officially accredited body, re-registration every three years**

# **6.5 SOFTWARE QUALITY ASSURANCE**

# SQA and IEEE standard 730

- IEEE Standard 730 offers a framework for the contents of a Quality Assurance Plan for software development

# **6.6 CMM**

# Capability Maturity Model (CMM)

- **software development process can be controlled, measured, and improved**
- **Can we measure the capability?**
- one of five maturity levels, evolutionary levels toward achieving a mature software process.

# Capability Maturity Model (CMM)



- **Initial level:** software development is ad-hoc
- **Repeatable level:** basic processes are in place
- **Defined level:** there are *standard* processes
- **Quantitatively managed level:** data is gathered and analyzed routinely
- **Optimizing level:** stable base, data is gathered to improve the process

# Initial ⇒ repeatable level

- At this level, the organization operates without formalized procedures, project plans, or cost estimates
- Performance can be improved by instituting basic project management controls:
  - Requirements management
  - Project planning
  - Project monitoring and control
  - Supplier agreement management
  - Measurement and analysis
  - Process and product quality assurance
  - Configuration management

# **Repeatable ⇒ defined level**

- **Repeatable** The main difference between the initial process level and the repeatable process level is that there is control over the way plans and commitments are established.
- the development of a new type of product, and major organizational changes however still represent major risks at this level
- **Advances in:**
  - Requirements development, Technical solution
  - Product integration
  - Verification and Validation
  - Organization process focus
  - Organization process definition
  - Organizational training
  - Integrated project management
  - Risk management
  - Decision analysis and resolution

# **Defined => Quantitatively managed**

- **At the defined process level, a set of standard processes for the development and maintenance of software is in place**
- **Organizational process performance**
- **Quantitative project management**

# Quantitatively managed => Optimizing

- At the quantitatively managed process level, quantitative data is gathered and analyzed on a routine basis. Everything is under control and attention may therefore shift from being reactive (what happens to the present project?) to being proactive (what can we do to improve future projects?). The focus shifts to opportunities for continuous improvement:

# CMM: critical notes



- **Most appropriate for big companies**
- **Pure CMM approach may stifle creativity**
- **Crude 5-point scale (now: CMMI)**

# Get started on Software Process Improvement (SPI)

- **Formulate hypotheses**
- **Carefully select metrics**
- **Collect data**
- **Interpret data**
- **Initiate improvement actions**
  
- **Iterate**

# Lessons w.r.t. data collection



- **Closed loop principle: result of data analysis must be useful to supplier of data**
- **Do not use data collected for other purposes**
- **Focus on continuous improvement**
- **Only collect data you really need**

# Summary



- **Product quality versus process quality**
- **Quality conformance versus quality improvement**
- **Quality has to be actively pursued**
- **There are different notions of quality**
- **Quality has many aspects**
- **Quality is hard to measure**

# Requirements Engineering

## capitolo 9

### Main issues:

- **What do we want to build**
- **How do we write this down**

# Requirement (IEEE610)

- “**a condition or capability needed by a user to solve a problem or achieve an objective**”
- **it has to be met?**
  - Most requirements are negotiable
- **RE vs requirement analysis**

# Requirements Engineering

- **the first step in finding a solution for a data processing problem**
  - **the results of requirements engineering is a requirements specification**
  - **requirements specification**
    - contract for the customer
    - starting point for design
  - **the phase in which the user's requirements are analyzed and documented is also sometimes called the 'specification' phase.**
- AVOID**

# Requirements engineering and design

- Requirements engineering and design generally cannot be strictly separated in time.
- Often, a preliminary design is done after an initial set of requirements has been determined.
- Based on the result of this design effort, the requirements specification may be changed and refined.

# Natural language specs are dangerous

**“All users have the same control field”**

- **the same value in the control field?**
- **the same format of the control field?**
- **there is one (1) control field for all users?**

# What put in the requirements specification?

- from '**essential** requirements' to 'nice features'?
- **future** requirements?
- Only **functions** of the software
- hardware?
- Performances?

# **market-driven vs customer-driven**

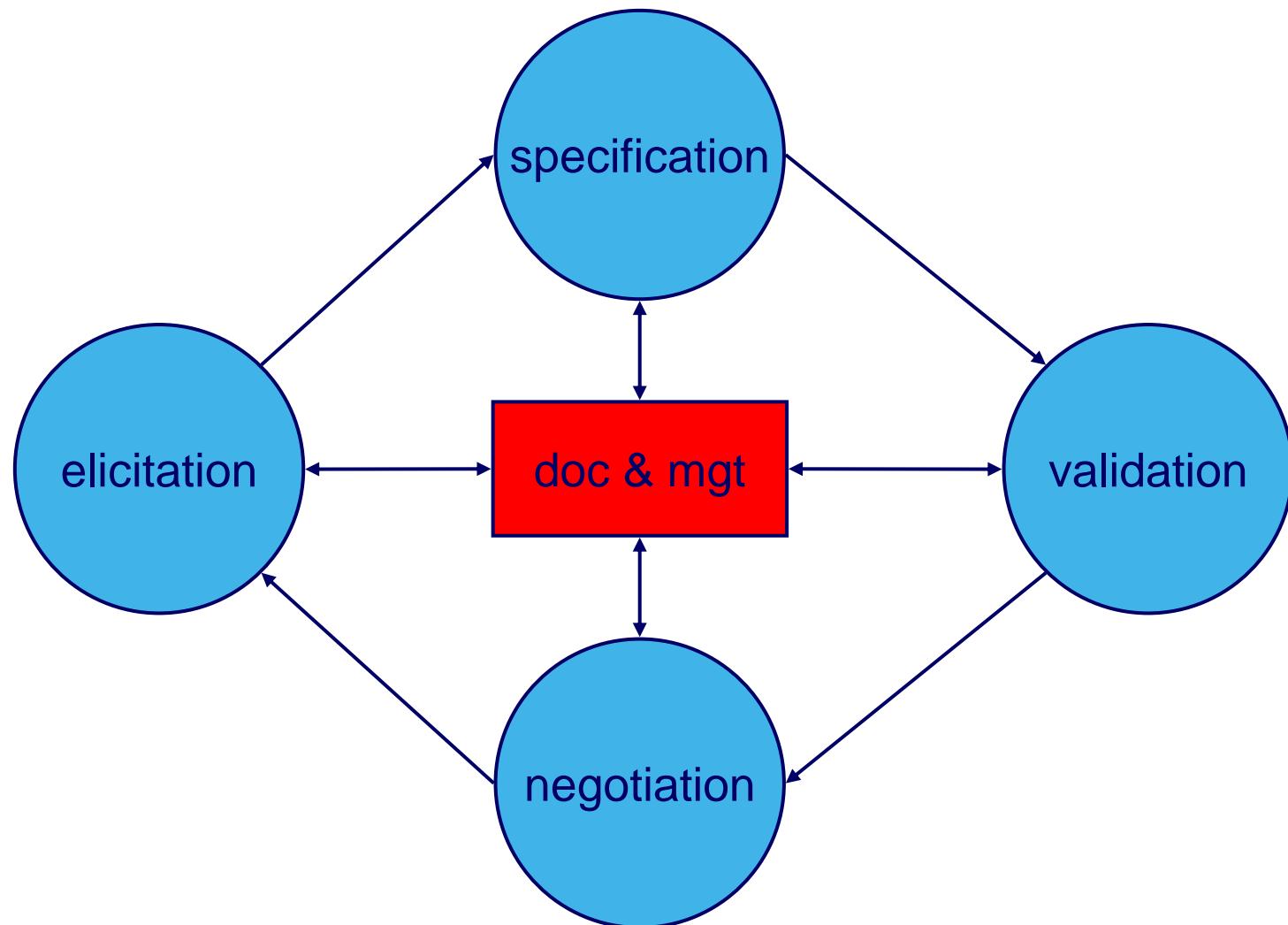
- Much software developed today is market-driven rather than customer-driven.
- No real user can be questioned to build the requirements

# **Requirements engineering, main steps**

- 1. understanding the problem: elicitation**
- 2. describing the problem: specification**
- 3. agreeing upon the nature of the problem: validation**
- 4. agreeing upon the boundaries of the problem: negotiation**

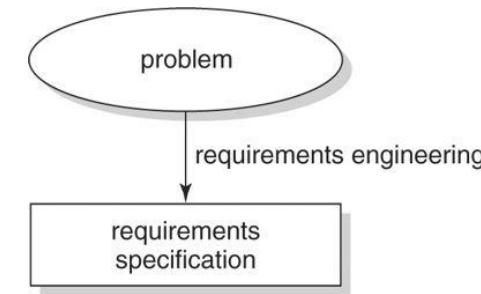
**This is an iterative process**

# Framework for RE process



# 1. ELICITATION

## Conceptual modeling



- you model part of reality: the **Universe of Discourse (UoD)**
- this model is an explicit **conceptual model**
- people in the UoD have an implicit **conceptual model of that UoD**
- making this implicit model explicit poses **problems**:
  - analysis problems
  - negotiation problems

# **Requirements engineering is difficult**

**Success depends on the degree with which we manage to properly describe the system desired**

Software is not continuous!

**Tsjechow vs Chekhov vs Чехов**

# Beware of subtle mismatches



- a library employee may also be a client
- there is a difference between `a book` and `a copy of a book`
- status info `present` / `not present` is not sufficient; a (copy of a) book may be lost, stolen, in repair, ...

# Humans as information sources



- different backgrounds
  - short-term vs long-term memory
  - human prejudices
  - limited capability for rational thinking
- 
- People involved in a UoD have an implicit conceptual model of that UoD.

# Requirements Engineering Paradigms

- **existing methods are “Taylorian”**
  - which tasks are recursively decomposed into simpler tasks and each task has one ‘best way’ to accomplish it.
  - By careful observations and experiments this one best way can be found and formalized into procedures and rules.
- **they may work in a “technical” environment, but many UoDs contain people as well, and their models may be irrational, incomplete, inconsistent, contradictory**
- **as an analyst, you cannot neglect these aspects; you participate in shaping the UoD**

## Point to ponder #1



- how do you handle conflicts during requirements engineering?

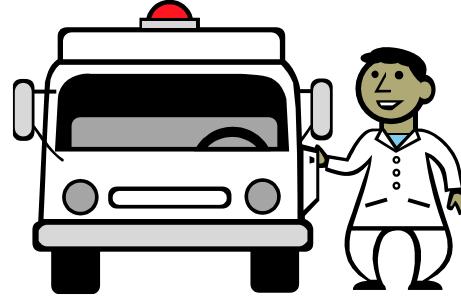
# How we study the world around us

- **people have a set of assumptions about a topic they study (paradigm)**
- **this set of assumptions concerns:**
  - how knowledge is gathered
  - how the world is organized
- **this in turn results in two dimensions:**
  - subjective-objective (wrt knowledge)
  - conflict-order (wrt the world)
- **which results in 4 archetypical approaches to requirements engineering**

## Four approaches to RE (skip 9.1.1)

- functional (**objective+order**): the analyst is the expert who empirically seeks the truth
- social-relativism (**subjective+order**): the analyst is a `change agent'. RE is a learning process guided by the analyst
- radical-structuralism (**objective+ conflict**): there is a struggle between classes; the analyst chooses for either party
- neohumanism (**subjective+conflict**): the analyst is kind of a social therapist, bringing parties together

## Point to ponder #2



- **how does the London Ambulance System example from chapter 1 relate to the different possible approaches to requirements engineering?**

# Elicitation techniques

- **Asking**
  - interview
  - Delphi technique
  - brainstorming session
- **task analysis**
- **scenario analysis**
- **ethnography**
- **form analysis**
- **analysis of natural language descriptions**
- **synthesis from existing system**
- **domain analysis**
- **Business Process Redesign (BPR)**
- **prototyping**

# Asking

- We may simply ask the users what they expect from the system.
- A presupposition then is that the user is able to bypass his own limitations and prejudices.
  - interviews
  - The Delphi technique is an iterative technique in which information is exchanged in written form until a consensus is reached.

# Task Analysis

- **Task analysis is the process of analyzing the way people perform their jobs: the things they do, the things they act on and the things they need to know.**
- **The relation between tasks and goals: a task is performed in order to achieve a goal.**
- **Task analysis has a broad scope.**

## Task Analysis (cntd)

- **Task analysis concentrates on the current situation. However, it can be used as a starting point for a new system:**
  - users will refer to new elements of a system and its functionality
  - scenario-based analysis can be used to exploit new possibilities
- **See also the role of task analysis as discussed in the context of user interface design (chapter 16)**

# Scenario-Based Analysis

- **Provides a more user-oriented view perspective on the design and development of an interactive system.**
- **The defining property of a scenario is that it projects a concrete description of an activity that the user engages in when performing a specific task, a description sufficiently detailed so that the design implications can be inferred and reasoned about.**
  - Scenarios and use cases are the elicitation methods most often used.
  - (vedremo poi con UML)

# Scenario-Based Analysis (example)

- **first shot:**
  - check due back date
  - if overdue, collect fine
  - record book as being available again
  - put book back
- **as a result of discussion with library employee:**
  - what if person returning the book is not registered as a client?
  - what if the book is damaged?
  - how to handle in case the client has other books that are overdue, and/or an outstanding reservation?

# Scenario-Based Analysis (cntd)

## The scenario view

- **concrete descriptions**
- **focus on particular instances**
- **work-driven**
- **open-ended, fragmentary**
- **informal, rough, colloquial**
- **envisioned outcomes**

## The standard view

- **abstract descriptions**
- **focus on generic types**
- **technology-driven**
- **complete, exhaustive**
- **formal, rigorous**
- **specified outcomes**

# Scenario-Based Analysis (cntd)

- **Application areas:**
  - requirements analysis
  - user-designer communication
  - design rationale
  - software architecture (& its analysis)
  - software design
  - implementation
  - verification & validation
  - documentation and training
  - evaluation
  - team building
- **Scenarios must be structured and managed**

Ethnography skip

# Form analysis (example)

A lot of information about the domain being modeled can often be found in various forms being used.

## Proceedings request form:

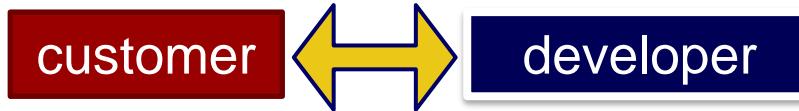
<b>Client name</b>	.....
<b>Title</b>	.....
<b>Editor</b>	.....
<b>Place</b>	.....
<b>Publisher</b>	.....
<b>Year</b>	.....

Natural language descriptions

## Other techniques:

- **Derivation from an existing system**
    - Or many: domain analysis.
  - **Business process redesign (BPR)**
  - **Prototyping**
- 
- As the uncertainty decreases, the beneficial effects of user participation in requirements engineering diminish. With greater uncertainty, however, greater user participation does have a positive effect on the quality of requirements engineering.

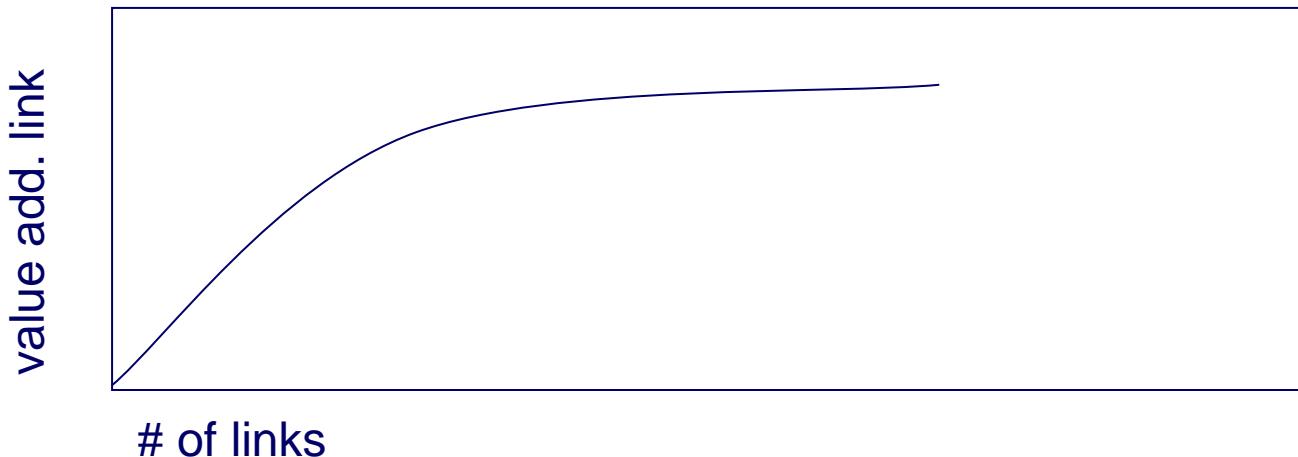
# Types of links between customer and developer



- facilitated teams
- intermediary
- support line/help desk
- survey
- user interface prototyping
- requirements prototyping
- interview
- usability lab
- observational study
- user group
- trade show
- marketing & sales

# Direct versus indirect links

- **lesson 1: don't rely too much on indirect links (intermediaries, surrogate users)**
- **lesson 2: the more links, the better - up to a point**

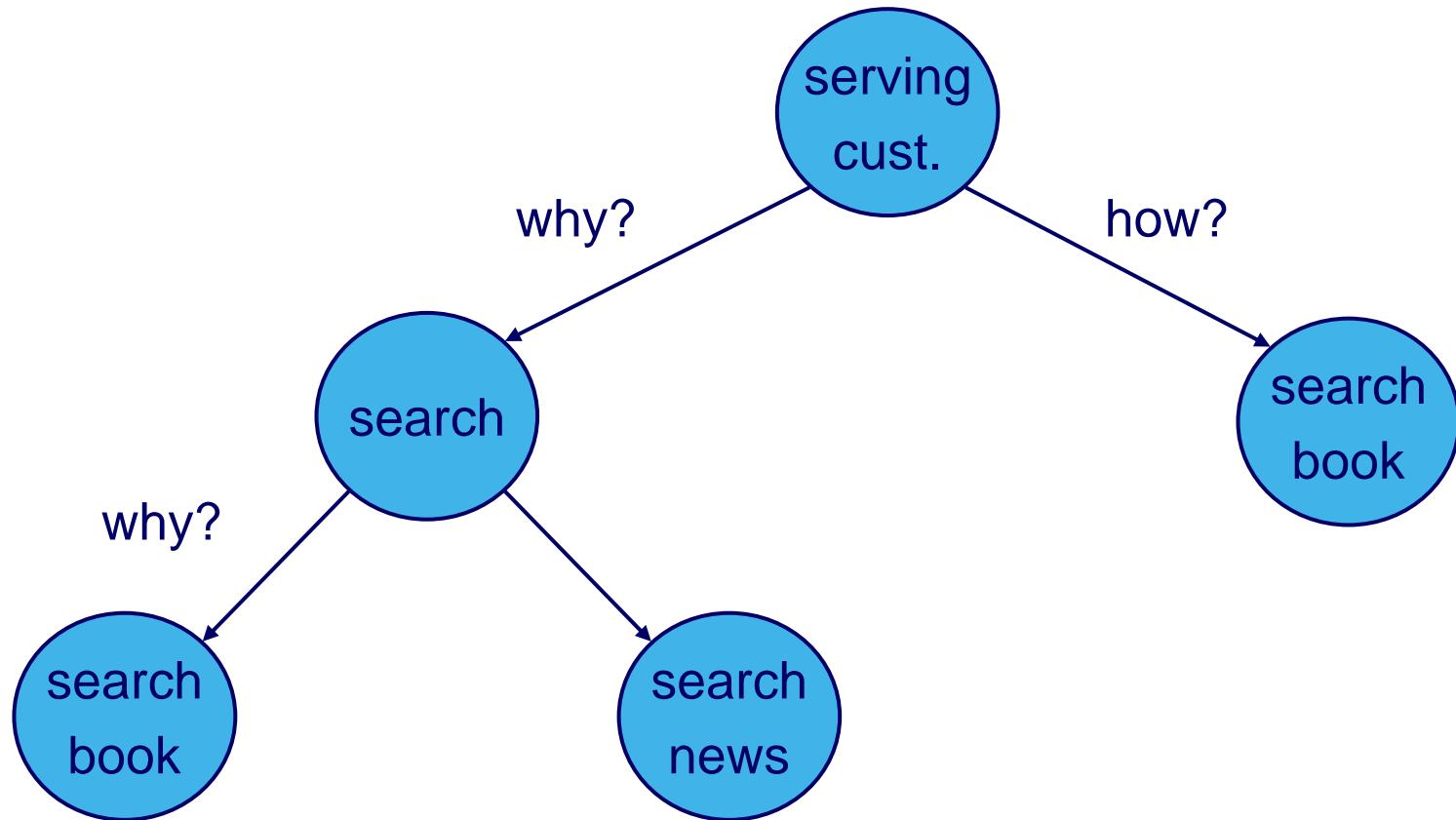


## **Structuring a set of requirements - Goals and Viewpoints**

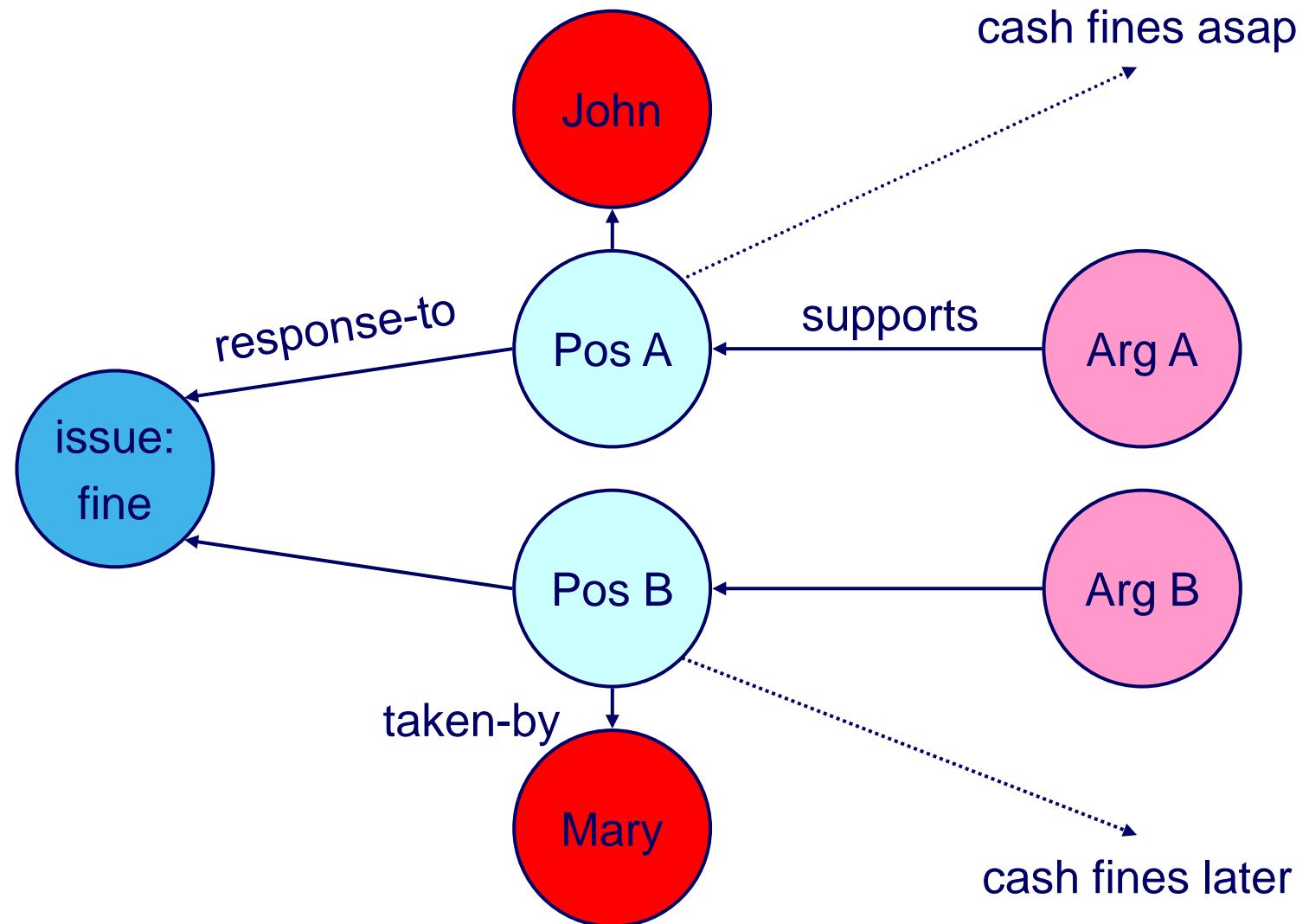
- 1. Hierarchical structure: higher-level reqs are decomposed into lower-level reqs**
- 2. Link requirements to specific stakeholders (e.g. management and end users each have their own set)**

**In both cases, elicitation and structuring go hand in hand**

# Goal-driven requirements engineering



# Conflicting viewpoints



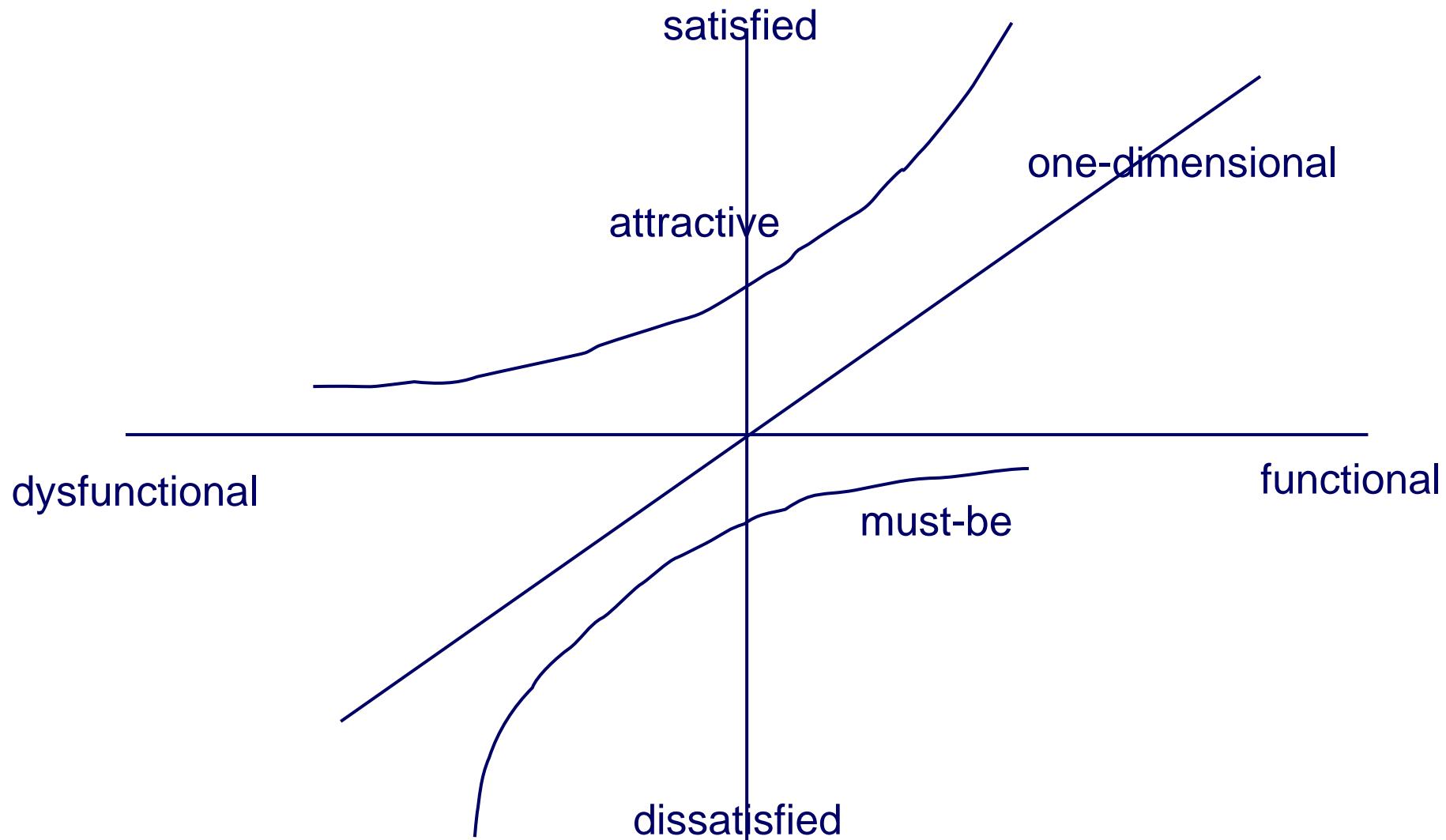
# Prioritizing requirements (MoSCoW)

- **Must haves:** top priority requirements
- **Should haves:** highly desirable
- **Could haves:** if time allows
- **Won't haves:** not today

## Prioritizing requirements (Kano model)

- **Attractive**: more satisfied if +, not less satisfied if –
- **Must-be**: dissatisfied when -, at most neutral
- **One-dimensional**: satisfaction proportional to number
- **Indifferent**: don't care
- **Reverse**: opposite of what analyst thought
- **Questionable**: preferences not clear

# Kano diagram



# COTS selection



- **COTS: Commercial-Off-The-Shelf**
  - the customer has to choose from what is available
- **Iterative process:**
  - Define requirements
  - Select components
  - Rank components
  - Select most appropriate component, or iterate
- **Simple ranking: weight \* score (WSM – Weighted Scoring Method)**

# Crowdsourcing

**the requirements engineering process is outsourced to a large community of volunteers**

- 1. Go to LEGO site**
  - 2. Use CAD tool to design your favorite castle**
  - 3. Generate bill of materials**
  - 4. Pieces are collected, packaged, and sent to you**
  
  - 5. Leave your model in LEGO's gallery**
  - 6. Most downloaded designs are prepackaged**
- 
- No requirements engineers needed!**
  - Gives rise to new business model**

# **REQUIREMENTS DOCUMENTATION AND MANAGEMENT**

# Requirements specification

- **readable**
- **understandable**
- **non-ambiguous**
- **complete**
- **verifiable**
- **consistent**
- **modifiable**
- **traceable**
- **usable**
- ...
- ...

# Requirements Specification IEEE830

- **correct.**
- **unambiguous**
- **complete**
- **(internally) consistent**
- **it should rank requirements for importance or stability**
- **verifiable**
- **modifiable**
- **traceable**

# **IEEE Standard 830**

## **Global structure of the requirements specification**

### **1. Introduction**

#### **1.1. Purpose**

#### **1.2. Scope**

#### **1.3. Definitions, acronyms and abbreviations**

#### **1.4. References**

#### **1.5. Overview**

### **2. General description**

#### **2.1. Product perspective**

#### **2.2. Product functions**

#### **2.3. User characteristics**

#### **2.4. Constraints**

#### **2.5. Assumptions and dependencies**

### **3. Specific requirements**

The IEEE framework for the requirements specification is especially appropriate in document-driven models for the software development process: the waterfall model and its variants.

# IEEE Standard 830 (cntd)

## 3. Specific requirements

### 3.1. External interface requirements

#### 3.1.1. User interfaces

#### 3.1.2. Hardware interfaces

#### 3.1.3. Software interfaces

#### 3.1.4. Comm. interfaces

### 3.2. Functional requirements

#### 3.2.1. User class 1

##### 3.2.1.1. Functional req. 1.1

##### 3.2.1.2. Functional req. 1.2

...

#### 3.2.2. User class 2

...

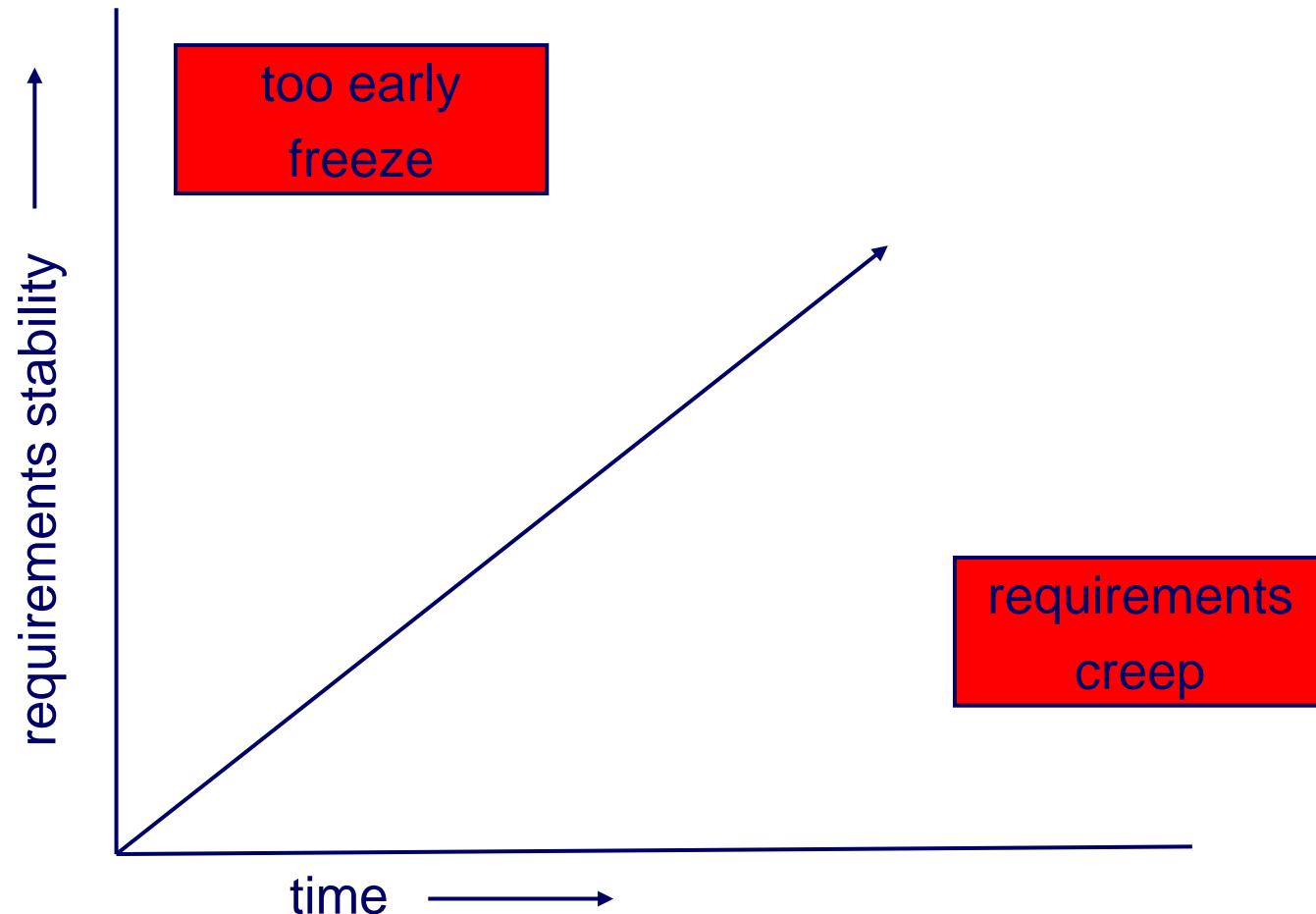
### 3.3. Performance requirements

### 3.4. Design constraints

### 3.5. Software system attributes

### 3.6. Other requirements

# Requirements management



# Requirements management

- Requirements identification (number, goal-hierarchy numbering, version information, attributes)
- Requirements change management (CM)
- Requirements traceability:
  - Where is requirement implemented?
  - Do we need this requirement?
  - Are all requirements linked to solution elements?
  - What is the impact of this requirement?
  - Which requirement does this test case cover?
- Related to Design Space Analysis

# **REQUIREMENTS SPECIFICATION TECHNIQUES**

# Which notation? using natural language?

- **noise**
- **silence**
- **overspecification**
- **contradictions**
- **ambiguity**
- **forward references**
- **wishful thinking (pia illusion)**



alternative given by Meyer is to first describe and analyze the problem using some formal notation and then translate it back into natural language

# Functional vs. Non-Functional Requirements

- **functional requirements:** the system services which are expected by the users of the system.
- **non-functional (quality) requirements:** the set of constraints the system must satisfy and the standards which must be met by the delivered system.
  - speed
  - size
  - ease-of-use
  - reliability
  - robustness
  - portability

# **VERIFICATION AND VALIDATION**

# Validation of requirements

- **inspection of the requirement specification w.r.t. correctness, completeness, consistency, accuracy, readability, and testability.**
- **some aids:**
  - structured walkthroughs
  - prototypes
  - develop a test plan
  - tool support for formal specifications

# Summary

- **goal: a maximally clear, and maximally complete, description of WHAT is wanted**
- **RE involves elicitation, specification, validation and negotiation**
- **modeling the UoD poses both analysis and negotiation problems**
- **you must realize that, as an analyst, you are more than an outside observer**
- **a lot is still done in natural language, with all its inherent problems**

# One final lesson

Walking on water

and

developing software from a specification

are easy

if they are frozen

**(E.V. Berard, Essays on object-oriented software engineering)**

# Modeling

## capitol 10

Main issues:

- What do we want to build
- How do we write this down

# Specifiche dei requisiti (1)

- La descrizione informale di un sistema SW è data in termini di **requisiti**
- La **specifiche dei requisiti** è una descrizione completa e non ambigua dei requisiti
  - Describe **COSA** un sistema software deve fare e non **COME** deve farlo
- I requisiti si suddividono in
  - Requisiti funzionali
  - Requisiti non funzionali
  - Requisiti del processo e manutenzione

# Specifiche dei requisiti (2)

- Una specifica va intesa come l' accordo tra il produttore di un servizio e il suo committente
  - Il 73% dei progetti SW vengono abbandonati o non rispondono alle aspettative a causa di requisiti errati [Standish 95]
  - Le cause del fallimento sono dovute a:
    - difetti iniziali dei requisiti
    - mancato coinvolgimento dell'utenza
    - incapacità di gestire le variazioni in corso d'opera dei requisiti stessi

# Specifiche dei requisiti (3)

- Lo sviluppo di una specifica richiede:
  - un'analisi iterativa e cooperativa del problema
    - Individuazione e raffinamento dei requisiti attraverso il colloquio con chi ha interesse al sistema (stakeholders)
  - l'impiego di linguaggi formali o semiformali

# Qualità delle specifiche (1)

- Chiarezza
- Non ambiguità
- Consistenza
- Completezza (interna, esterna)
- Incrementalità
- Comprensibilità

# Qualità delle specifiche (2)

- **Chiarezza:**
  - la specifica deve descrivere quanto più chiaramente possibile i termini e le operazioni coinvolte
- Esempio:
  - “*La selezione è il processo di designazione di **aree** del documento su cui si vuole operare. La maggior parte delle operazioni di modifica e formattazione richiede due fasi: è necessario prima selezionare ciò su cui si vuole operare; poi si può iniziare l'azione appropriata.*”
    - Cosa si intende per **area** su cui si vuole operare ?

# Qualità delle specifiche <sup>(3)</sup>

- Non ambiguità:
  - la specifica non deve generare interpretazioni ambigue
- Esempio 1:
  - *“Il messaggio deve essere triplicato. Le tre copie devono essere inviate su tre diversi canali ad uno stesso soggetto ricevente che accetterà il messaggio utilizzando una politica di selezione del tipo 2 su 3.”*
    - Il requisito è **ambiguo**: il ricevente deve aspettare la ricezione di tutti i messaggi prima di applicare la politica di decisione ?

# Qualità delle specifiche (4)

- Non ambiguità: (continua)
- Esempio 2:
  - *“Si richiede di scrivere un programma che prende in input una sequenza di interi e da in output la versione ordinata di tale sequenza”*
    - Il requisito è ambiguo: la sequenza di output deve essere ordinata in modo ascendente o discendente?
      - Il comportamento del programma **sort** dipenderà dalla decisione presa dal programmatore

# Qualità delle specifiche (5)

- **Consistenza:**
- la specifica non deve contenere contraddizioni
- Esempio:
  - “*Il testo deve essere mantenuto su linee di uguale lunghezza, specificata dall’utente*
  - *A meno che l’utente non lo specifichi esplicitamente, una parola non può essere interrotta da un comando di invio*
    - I 2 requisiti sono **in contraddizione**: cosa accade se una parola è più lunga del limite definito dall’utente ?

# Qualità delle specifiche (7)

- **Completezza interna:**

- la specifica deve definire ogni concetto nuovo e ogni terminologia usata (definire un **glossario**)

- “*In assenza di altre richieste, l’ascensore passa in uno stato di attesa-per-richiesta*”
  - Occorre definire cosa si intende per stato = “attesa-per-richiesta”
  - Es.: *area* nel programma di videoscrittura

- **Completezza esterna:**

- la specifica deve essere completa rispetto ai requisiti
  - Deve documentare tutti i requisiti richiesti

# Qualità delle specifiche <sup>(9)</sup>

- **Comprensibilità:**
- la specifica, in quanto contratto tra committente e produttore, deve essere intuitiva e comprensibile per il cliente
  - Possibilmente *visuale*
  - Il più possibile *succinta*

# Linguaggi per la specifica

- **Informali**

- Linguaggio naturale

- **Semi-formali** (es. UML)

- Possibilmente grafici

- **Formali**

- Formalismi operazionali
  - Formalismi dichiarativi
    - Formalismo è una notazione rigorosa

# Diversi modi di specifica

- **Informale:**

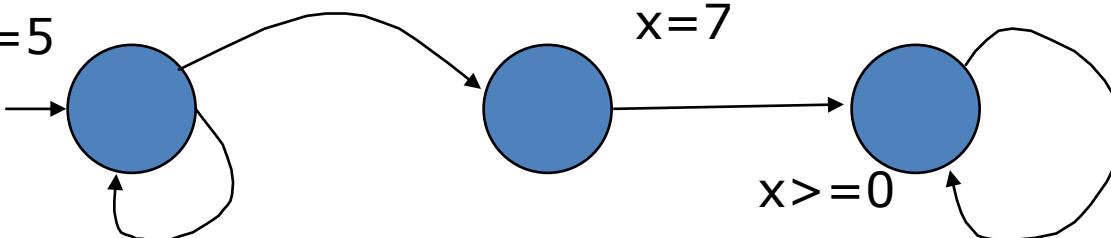
Il valore di  $x$  sarà tra 1 e 5, finché ad un certo momento diventa 7. In ogni caso non sarà mai negativo.

- **Formale:** (logica temporale)

$$(1 \leq x \leq 5 \cup x=7) \wedge [] x \geq 0$$

- **Grafico:**

$$1 \leq x \leq 5$$



# Diversi formalismi <sup>(1)</sup>

- **Formalismi operazionali:**

- definiscono il sistema descrivendone il comportamento (normalmente mediante un modello) come se eseguito da una macchina astratta

- **Formalismi dichiarativi:**

- definiscono il sistema dichiarando le proprietà che esso deve avere

# Diversi formalismi (2)

- Esempio: definizione di ellisse

- **Definizione operazionale**

E' l'insieme dei punti del piano che si ottiene muovendosi in modo che la somma delle distanze tra il punto e due punti fissi p1 e p2 rimanga invariata

- **Definizione dichiarativa**

E' l'insieme dei punti del piano che soddisfano l'equazione

$$ax^2 + by^2 + c = 0$$

# Diversi formalismi (3)

- Esempio: definizione di array ordinato

- Definizione operazionale

Sia  $a$  un array di  $n$  elementi. Il risultato del suo ordinamento è un array  $b$  di  $n$  elementi tali che il primo elemento di  $b$  è il minimo di  $a$  (se diversi elementi hanno lo stesso valore, uno di essi è selezionato); il secondo elemento di  $b$  è il minimo dell' array di  $n-1$  elementi ottenuto da  $a$  rimuovendo il suo elemento minimale; e così via fino a che gli  $n$  elementi di  $a$  sono stati rimossi

- Definizione dichiarativa

Il risultato dell'ordinamento di un array  $a$  è un array  $b$  che è una permutazione di  $a$  ed è ordinato

# Formalismi operazionali

- L' **approccio operazionale** fornisce una rappresentazione più intuitiva poiché più simile al modo di ragionare della mente umana
  - Permette facilità di realizzazione
  - Permette facoltà di validazione
- Esempi di formalismi operazioni:
  - Abstract State Machines
  - B method
  - Z method
  - SCR (Software Cost Reduction)
  - Reti di Petri

# Formalismi dichiarativi

- L' **approccio dichiarativo** fornisce una rappresentazione che non si presta ad ambiguità, ma è più difficile da comprendere e sviluppare
  - Permette facilità di verifica
- Esempi di formalismi dichiarativi:
  - Logica temporale
  - Trio
  - Algebre dei processi

# Choosing a modeling notation (9.3.1)

- The user → a document which speaks his language
  - Natural language + terms from the domain
- software engineer → often use some formal/informal language.
  - A requirements specification phrased in such a formal language may be checked using formal techniques, for instance with regard to consistency and completeness.

# System Modeling Techniques

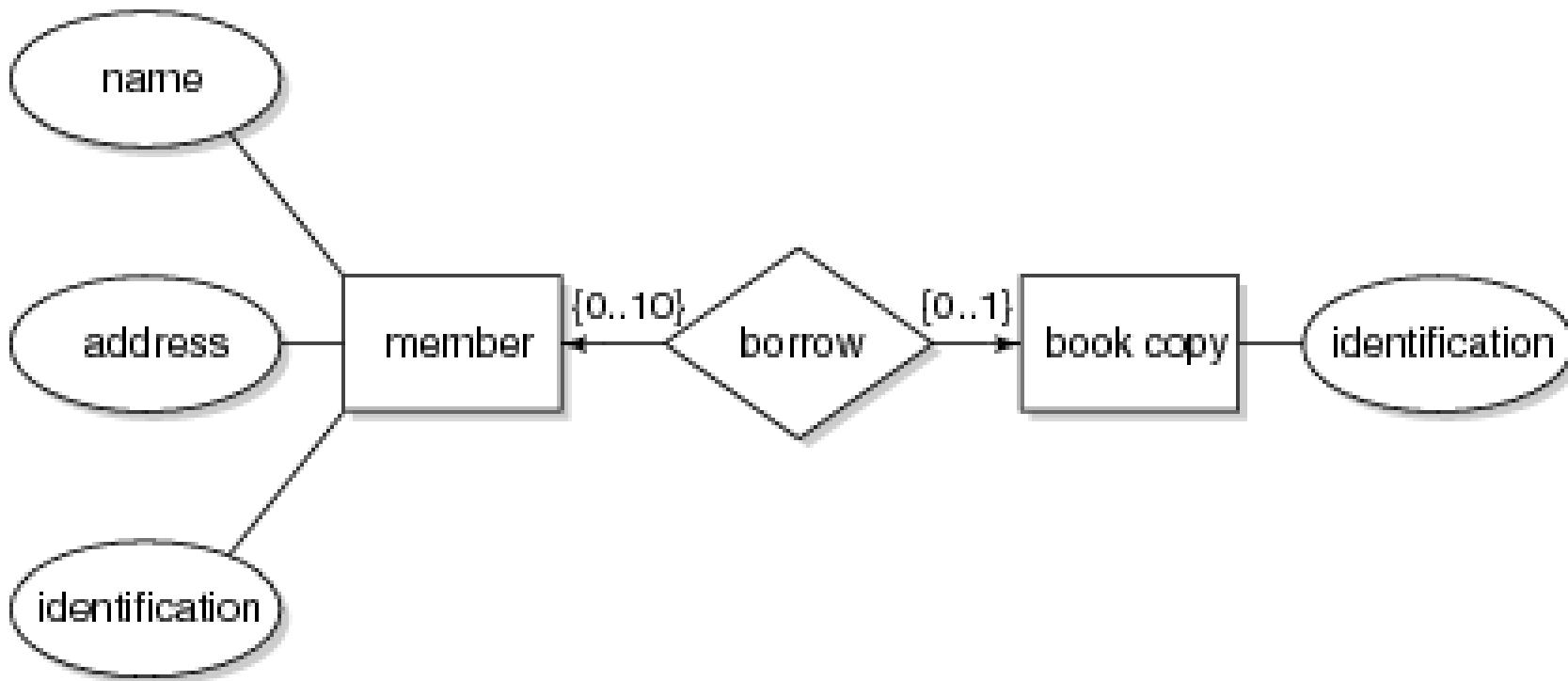
- Classic modeling techniques:
  - Entity-relationship modeling
  - Finite state machines
  - Data flow diagrams
  - CRC cards
- Object-oriented modeling: variety of UML diagrams



# Entity-Relationship Modeling

- entity: distinguishable object of some type
- entity type: type of a set of entities
- attribute value: piece of information (partially) describing an entity
- attribute: type of a set of attribute values
- relationship: association between two or more entities

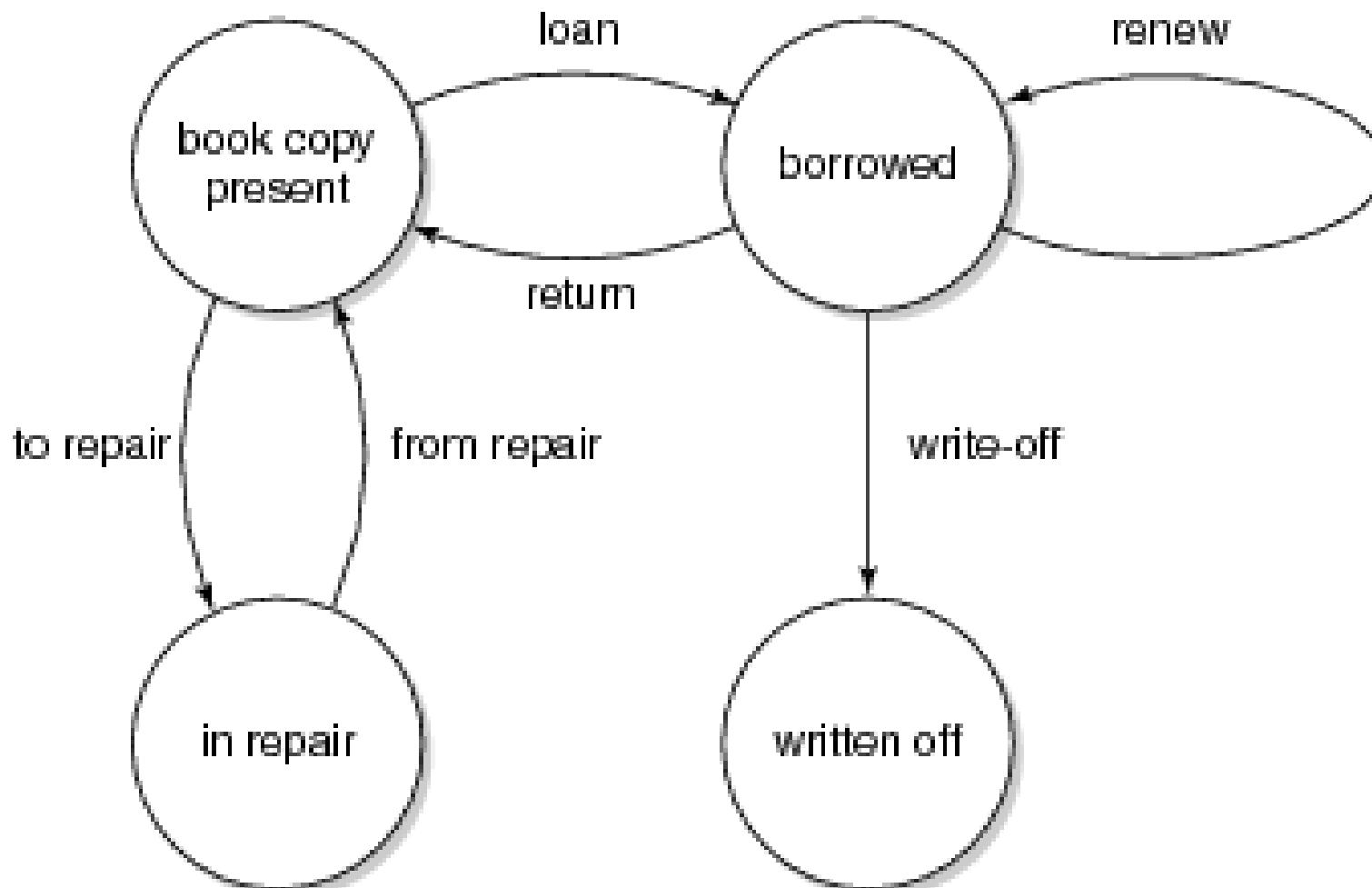
# Example ER-diagram



# Finite state machines

- Models a system in terms of (a finite number of) states, and transitions between those states
- Often depicted as state transition diagrams:
  - Each state is a bubble
  - Each transition is a labeled arc from one state to another
- Large system  $\Rightarrow$  large diagram hierarchical diagrams:  
statecharts $\Rightarrow$

# Example state transition diagram



# Data flow diagrams

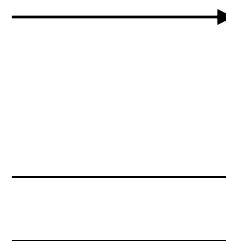
- external entities



- processes

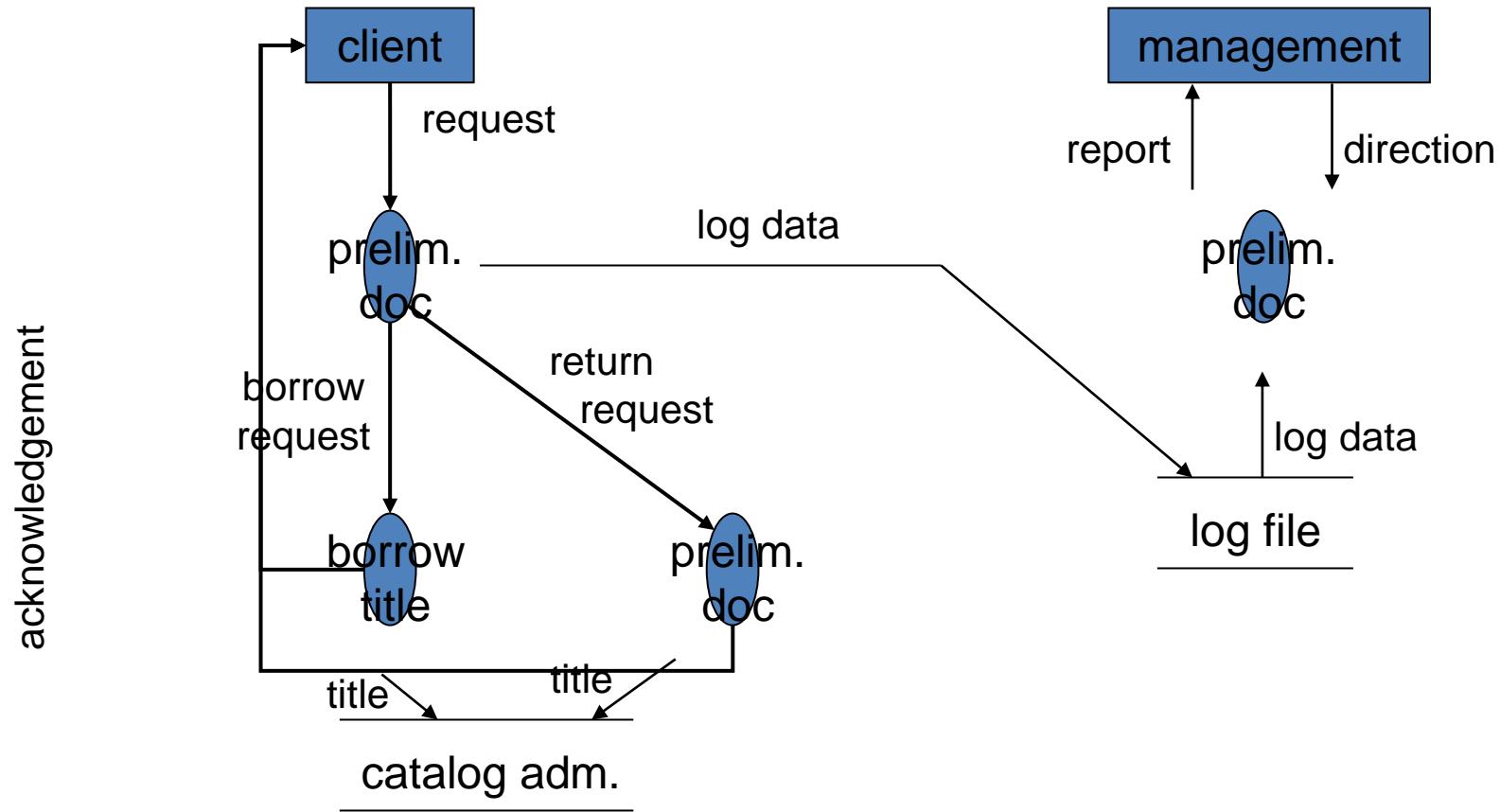


- data flows



- data stores

# Example data flow diagram



# CRC: Class, Responsibility, Collaborators

<b>Class</b> Reservations	<b>Collaborators</b> <ul style="list-style-type: none"><li>• Catalog</li><li>• User session</li></ul>
<b>Responsibility</b> <ul style="list-style-type: none"><li>• Keep list of reserved titles</li><li>• Handle reservations</li></ul>	

# Intermezzo: what is an object?

- Modeling viewpoint: model of part of the world
  - Identity+state+behavior
- Philosophical viewpoint: existential abstractions
  - Everything is an object
- Software engineering viewpoint: data abstraction
- Implementation viewpoint: structure in memory
- Formal viewpoint: state machine

# Objects and attributes

- Object is characterized by a set of attributes
  - A table *has* a top, legs, ...
- In ERM, attributes denote *intrinsic* properties; they do not depend on each other, they are descriptive
- In ERM, relationships denote *mutual* properties, such as the membership of a person of some organization
- In UML, these relationships are called *associations*
- Formally, UML does not distinguish attributes and relationships; both are properties of a class

# Objects, state, and behavior

- *State* = set of attributes of an object
- *Class* = set of objects with the same attributes
- Individual object: *instance*
- Behavior is described by *services*, a set of *responsibilities*
- Service is invoked by *sending a message*

# Relations between objects

- Specialization-generalization, is-a
  - A dog *is an* animal
  - Expressed in hierarchy
- Whole-part, has
  - A dog *has* legs
  - Aggregation of parts into a whole
  - Distinction between ‘real-world’ part-of and ‘representational’ part of (e.g. ‘Publisher’ as part of ‘Publication’)
- Member-of, has
  - A soccer team *has* players
  - Relation between a set and its members (usually not transitive)

# Specialization-generalization relations

- Usually expressed in hierarchical structure
  - If a tree: single inheritance
  - If a DAG: multiple inheritance
- Common attributes are defined at a higher level in the object hierarchy, and *inherited* by child nodes
- Alternative view: object hierarchy is a *type hierarchy*, with *types* and *subtypes*

# Unified Modeling Language (UML)

- Controlled by OMG consortium: Object Management Group
- Latest version: UML 2
- UML 2 has 13 diagram types
  - Static diagrams depict static structure
  - Dynamic diagrams show what happens during execution
- Most often used diagrams:
  - class diagram: 75%
  - Use case diagram and communication diagram: 50%
  - Often loose semantics

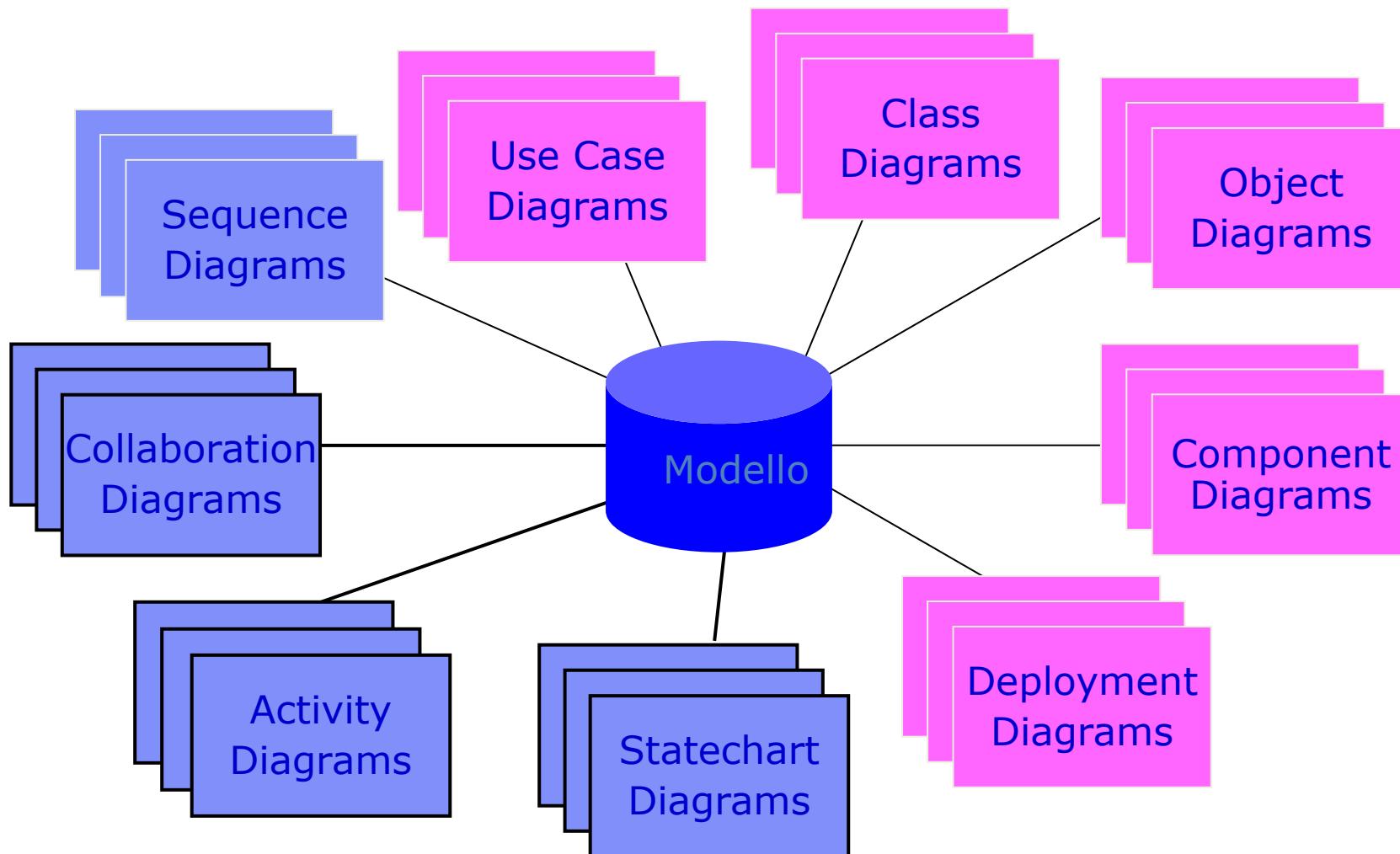
# UML

- **UML**: Linguaggio di Modellazione Unificato

- Definisce una:

- notazione grafica
- complesso di viste organizzate in diagrammi
  - strutturali e comportamentali
- sintassi mediante meta-modello
- semantica descritta in prosa
  - descrizione ridondante, frammentaria, ambigua

# Modelli e Diagrammi UML



# Viste in UML

## – Use-Case View

- Mostra le funzionalità che il sistema dovrebbe fornire così come sono percepite da attori esterni (black box view)

## – Logical View

- Analizza l'interno del sistema (struttura statica e dinamica) e descrive come le funzionalità sono fornite (white box view)

## – Component View

- Mostra l'organizzazione e le dipendenze delle componenti computazionali del sistema

## – Deployment View

- Descrive l'allocazione delle parti del sistema software in una architettura fisica

# Viste e diagrammi UML

## – Use-Case View

- Diagramma dei casi d'uso

## – Logical View

- Diagramma delle classi e degli oggetti
- Diagrammi di interazione
  - Diagrammi di sequenza e diagrammi di collaborazione
- Macchine di stato
- Diagramma di attività

## – Component View

- Diagramma delle componenti

## – Deployment View

- Diagramma di dislocamento

# UML diagram types

## *Static diagrams:*

- Class
- Component
- Deployment
- Interaction overview
- Object
- Package

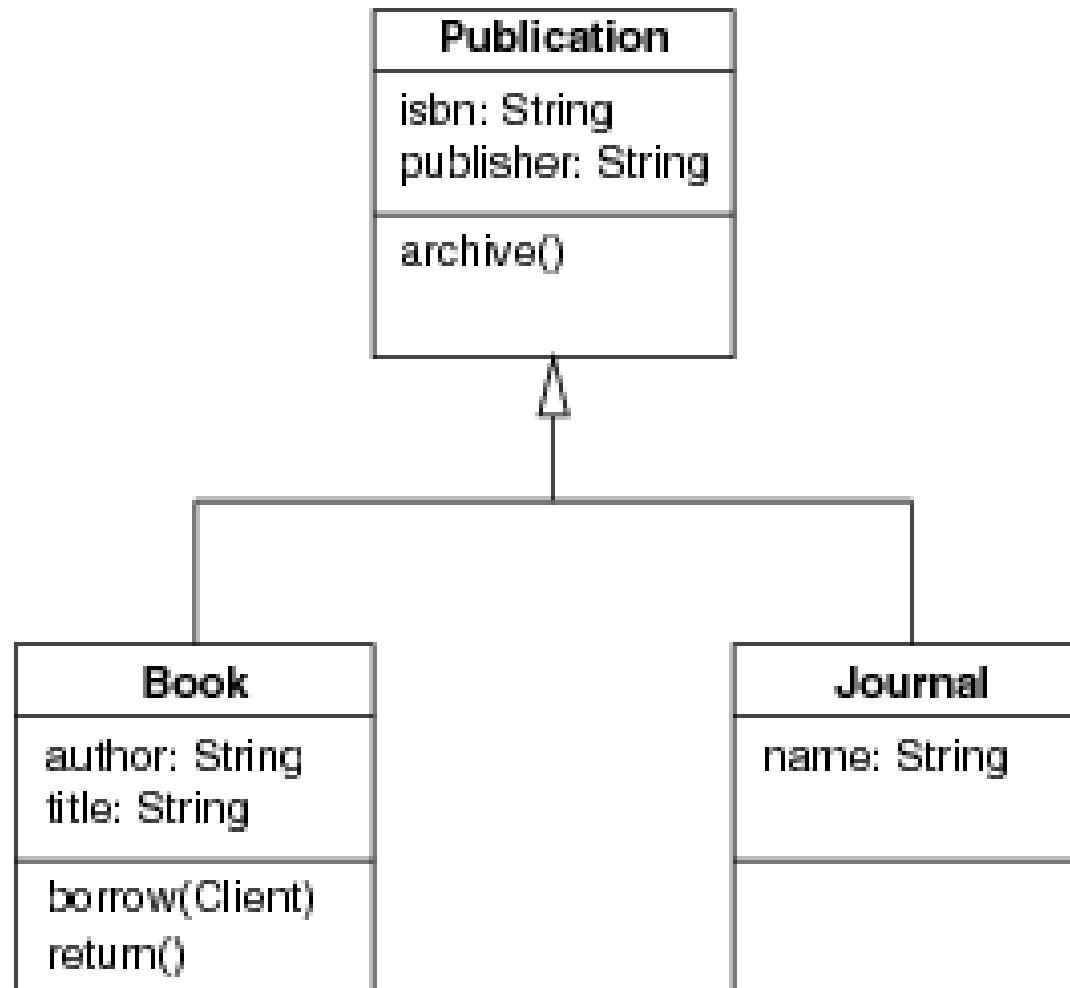
## *Dynamic diagrams:*

- Activity
- Communication
- Composite structure
- Sequence
- State machine
- Timing
- Use case

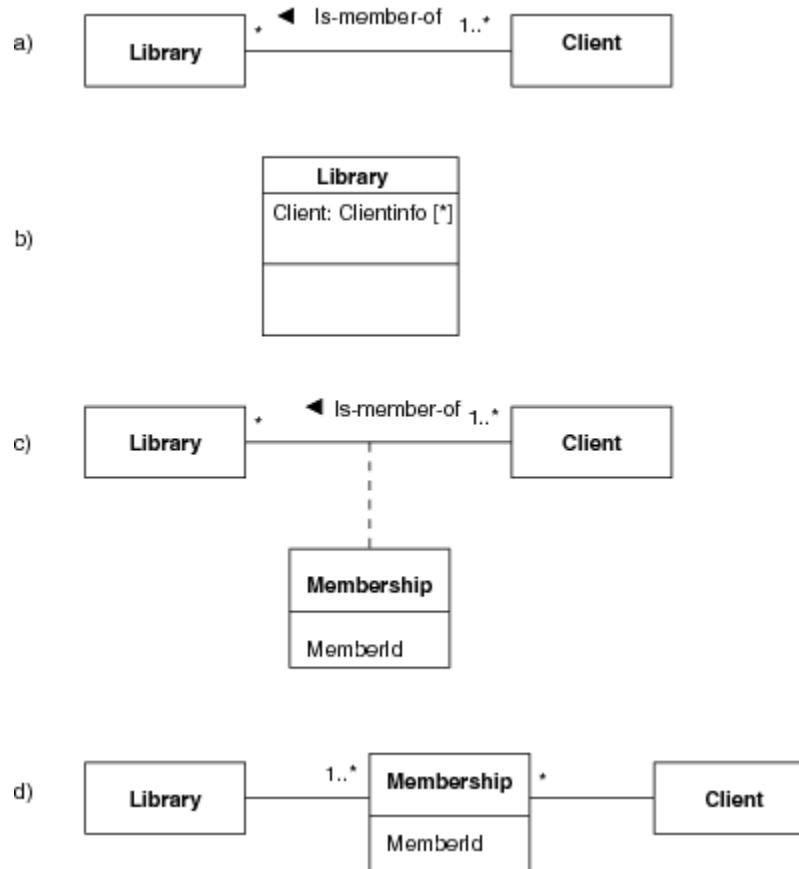
# UML class diagram

- depicts the static structure of a system
- most common example: subclass/superclass hierarchy
- also mutual properties between two or more entities (ER relationships, often called associations in OO)

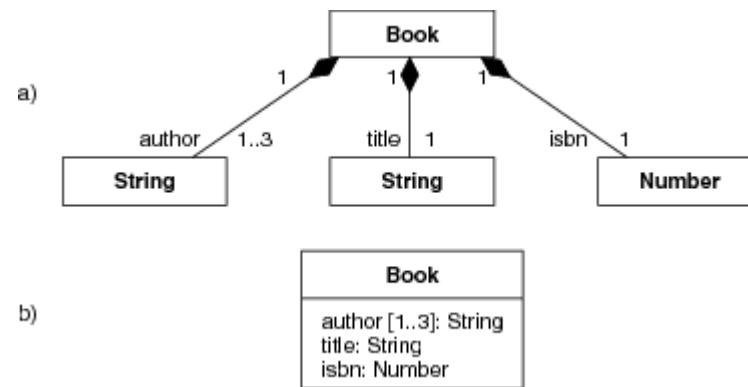
# Example class diagram (1): generalization



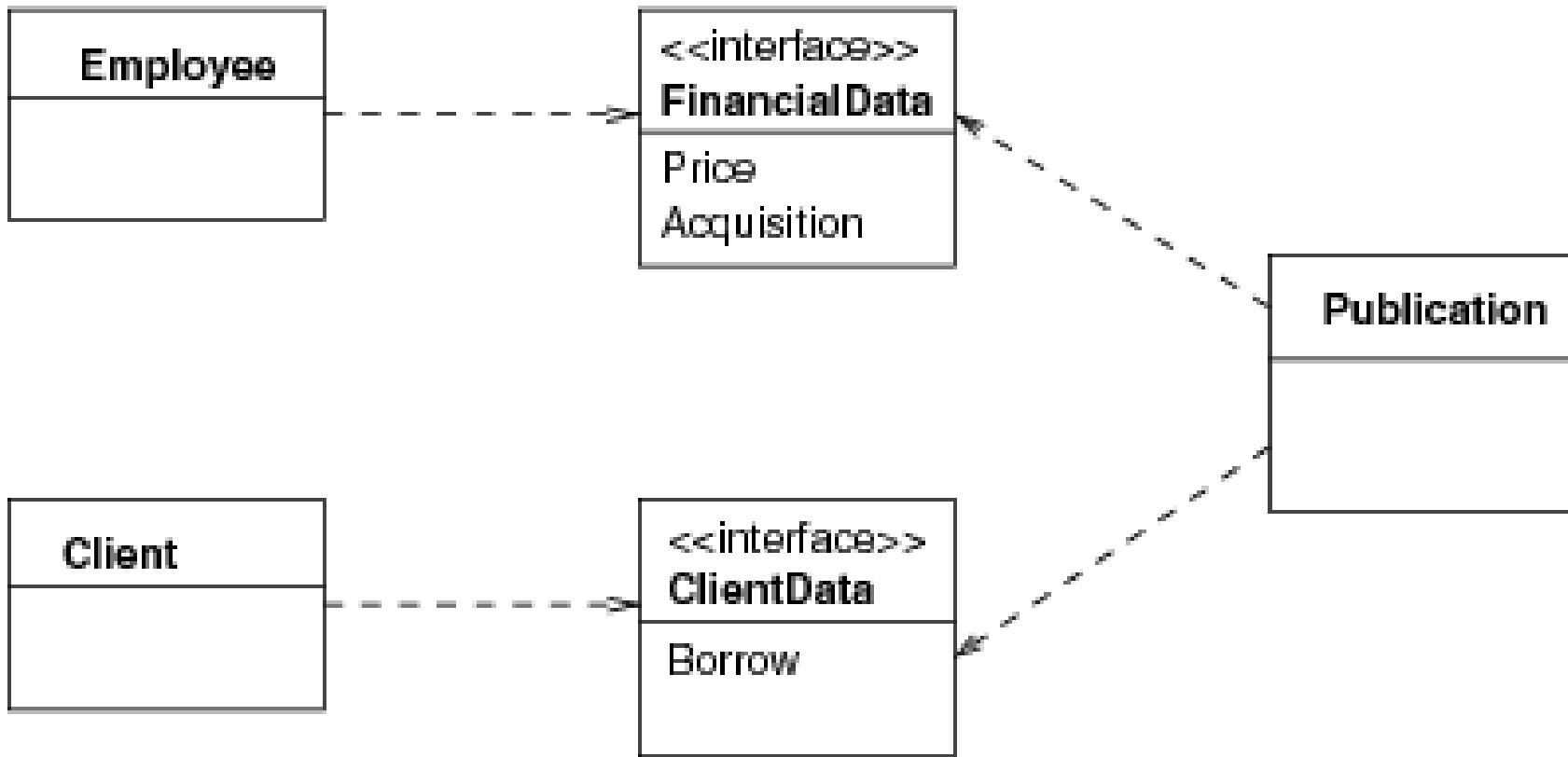
# Example class diagram (2) association



# Example class diagram (3): composition



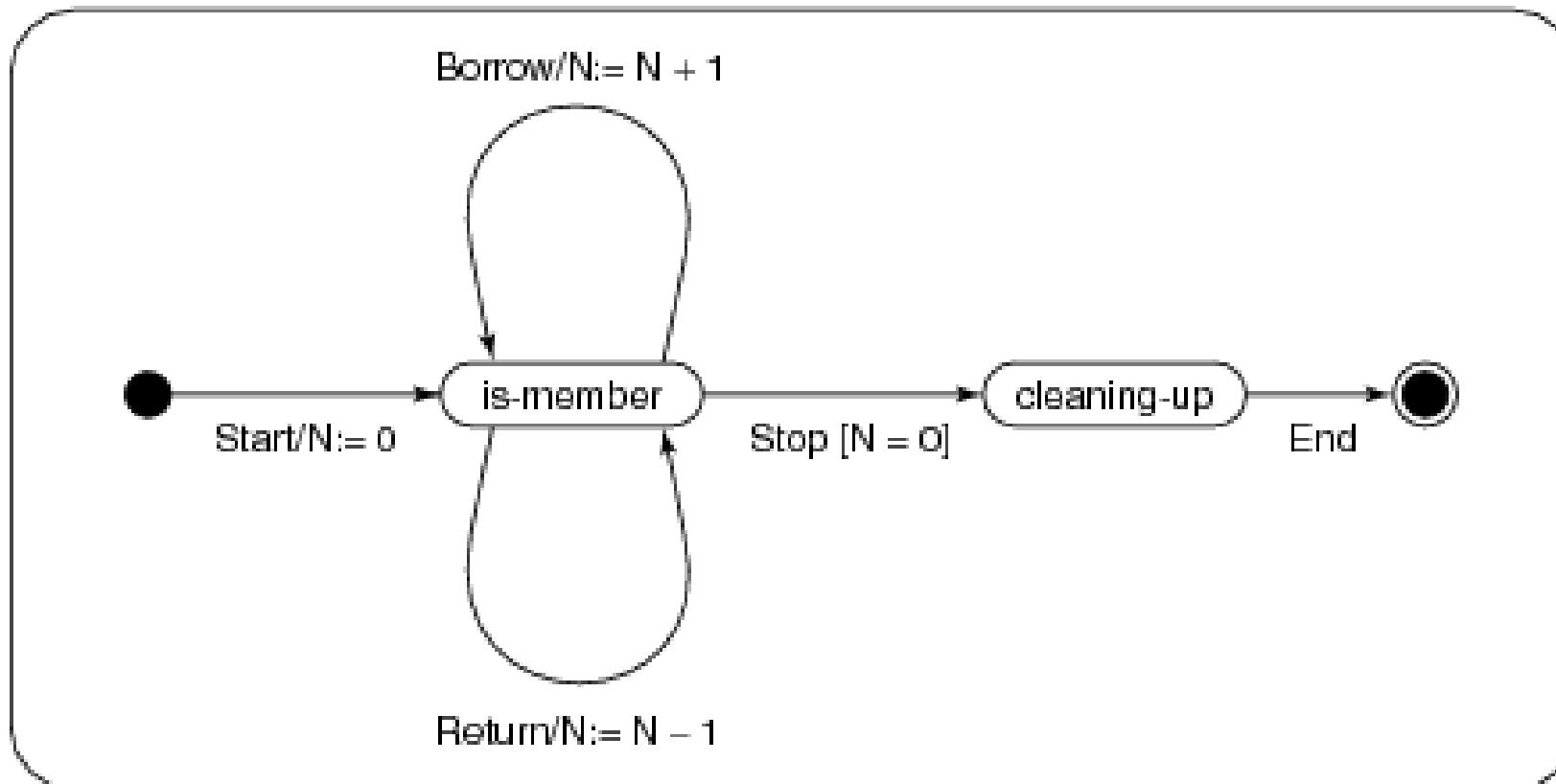
# Interface: class with abstract features



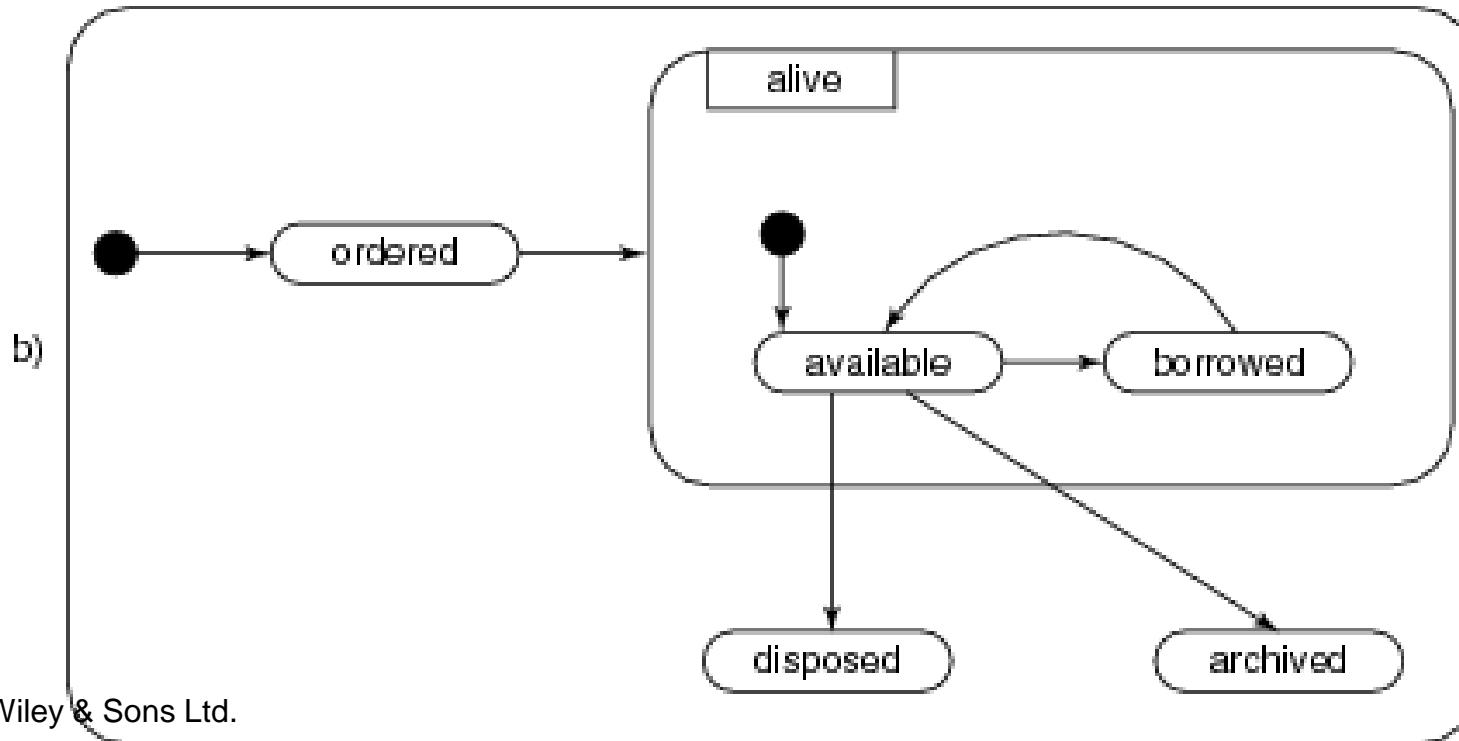
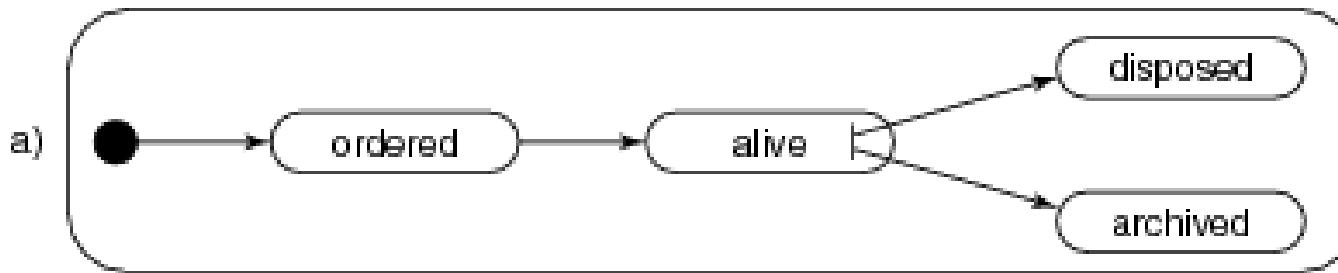
# State machine diagram

- Resembles finite state machines, but:
- Usually allows for local variables of states
- Has external inputs and outputs
- Allows for hierarchical states

# Example state machine diagram



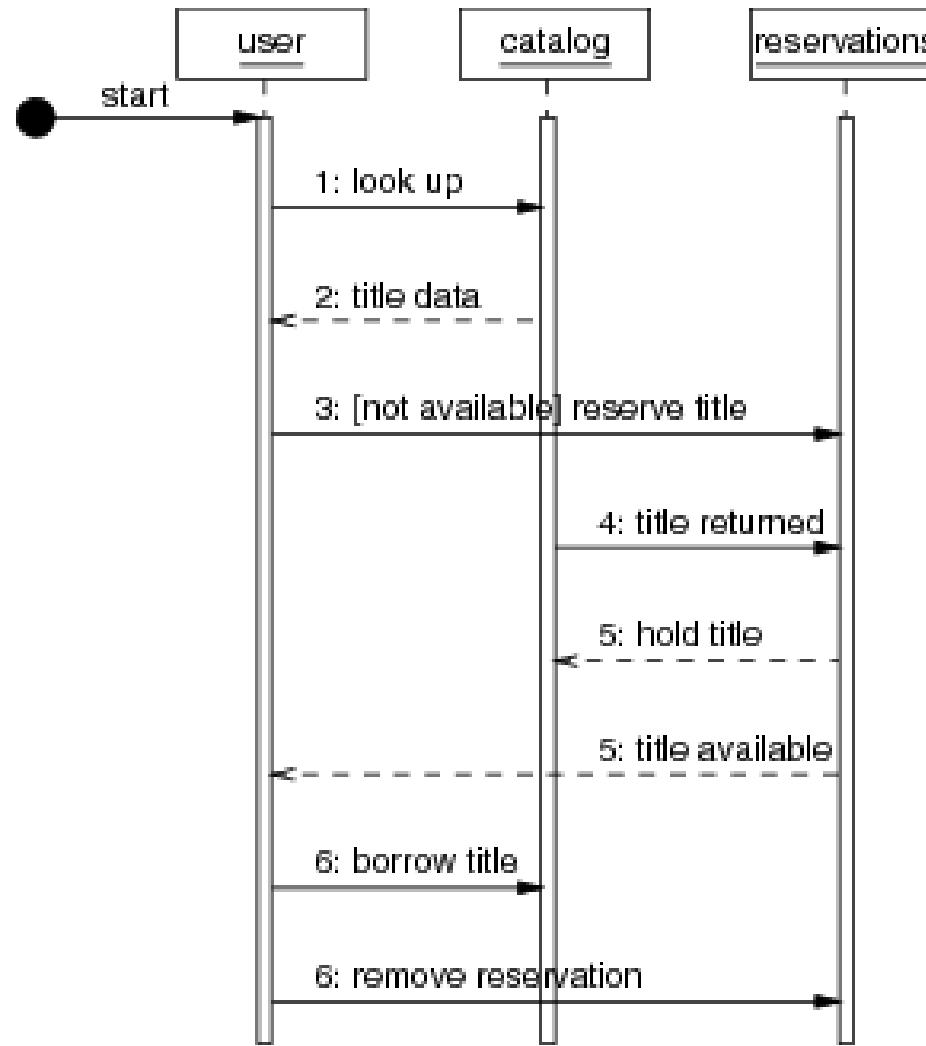
# Example state machine diagram: global and expanded view



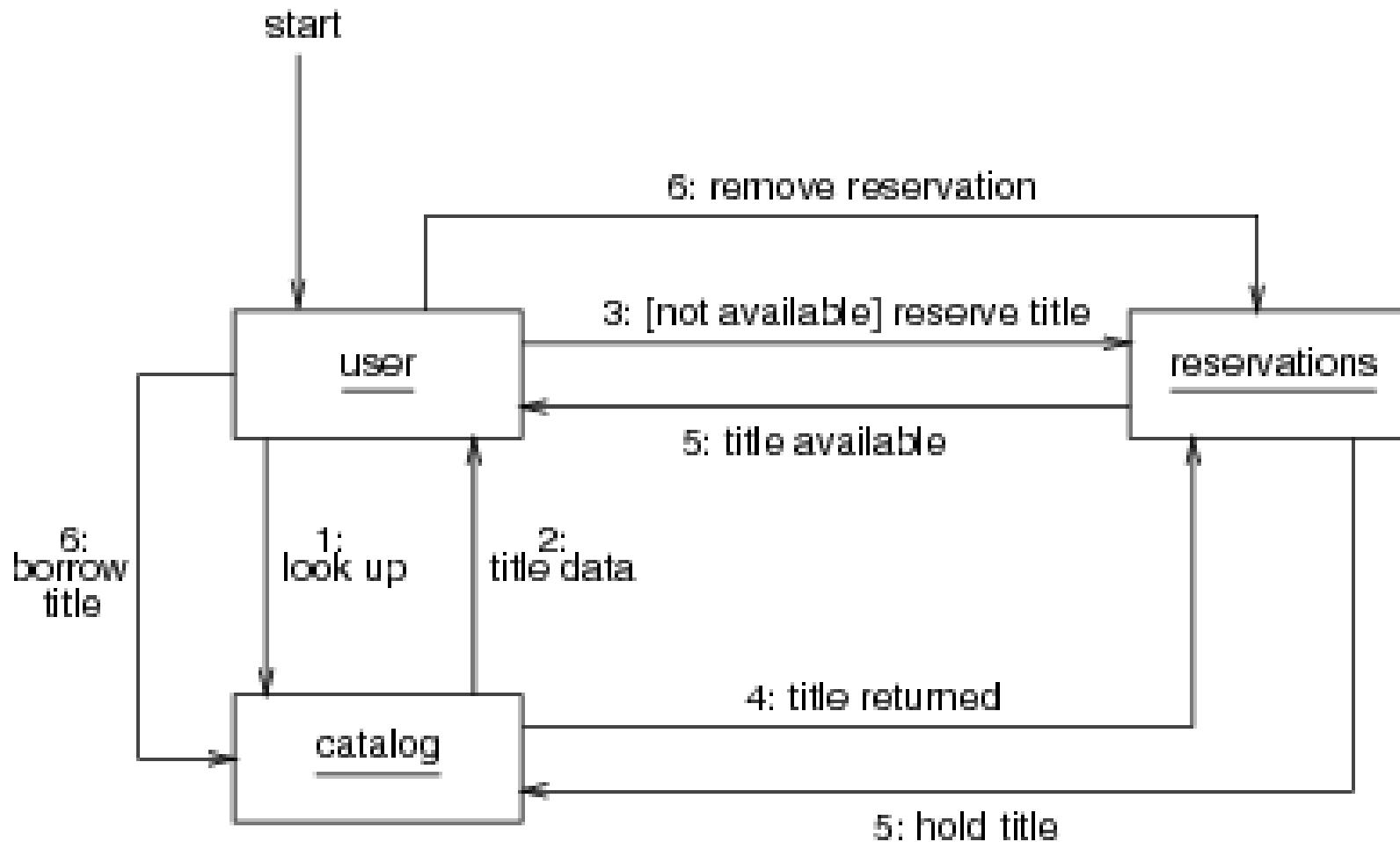
# Interaction diagram

- Two types: sequence diagram and communication diagram
- Sequence diagram: emphasizes the ordering of events, using a *lifeline*
- Communication diagram emphasizes objects and their relationships

# Example sequence diagram



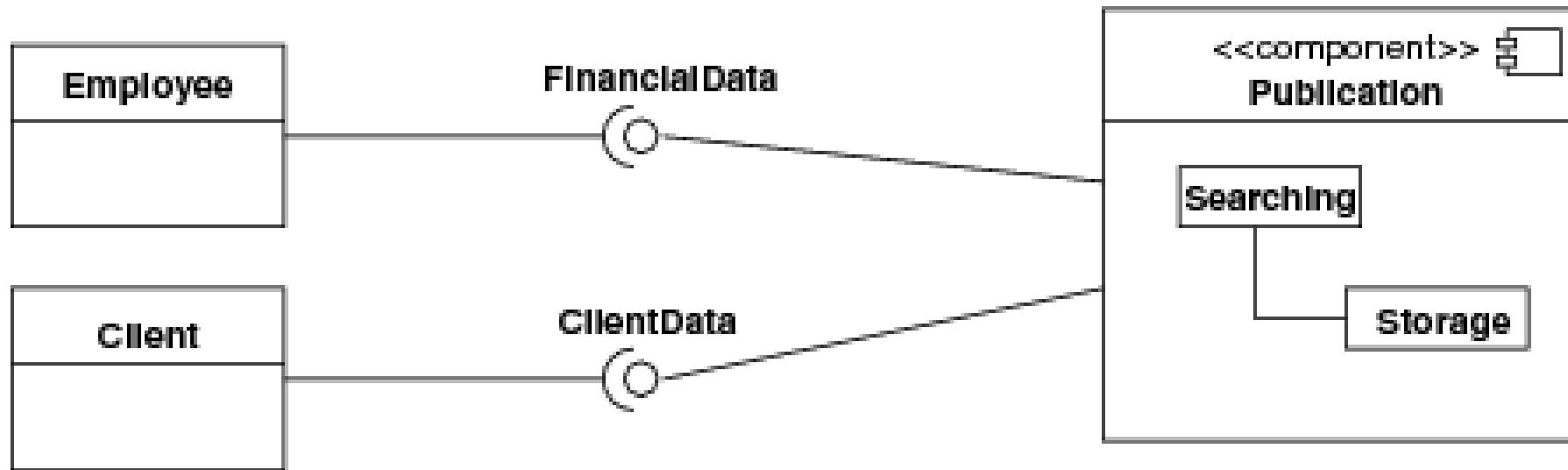
# Example communication diagram



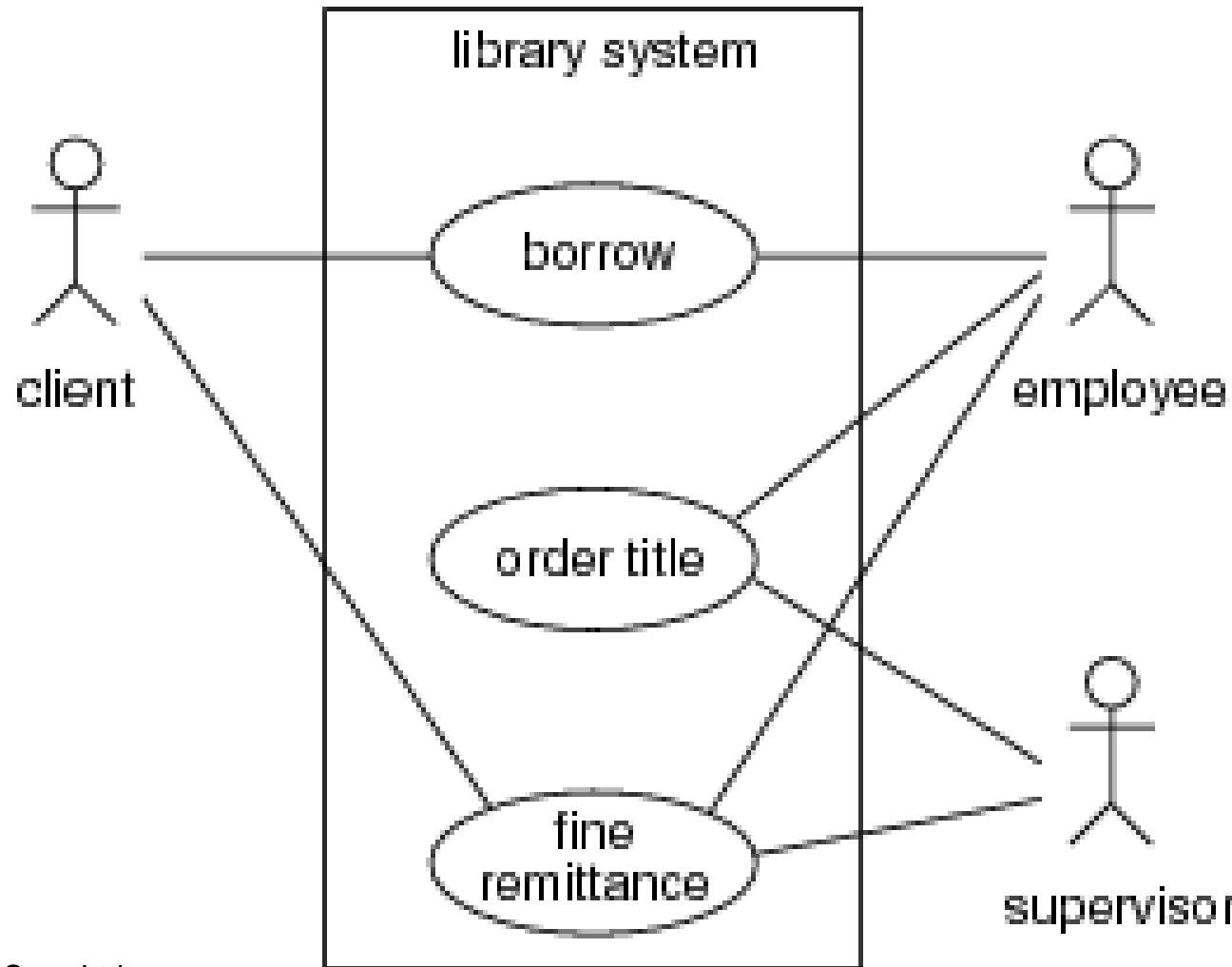
# Component diagram

- Class diagram with stereotype <<component>>
- Way to identify larger entities
- One way to depict a module view (see Software Architecture chapter)
- Components are connected by interfaces

# Example component diagram



# Use case diagram



# Summary



- Classic notations:
  - Entity-relationship diagrams
  - Finite state machines
  - Data flow diagrams
  - CRC cards
- Unified Modeling Language (UML)
  - evolved from earlier OO notations
  - 13 diagram types
  - widely used

# **Software Architecture**

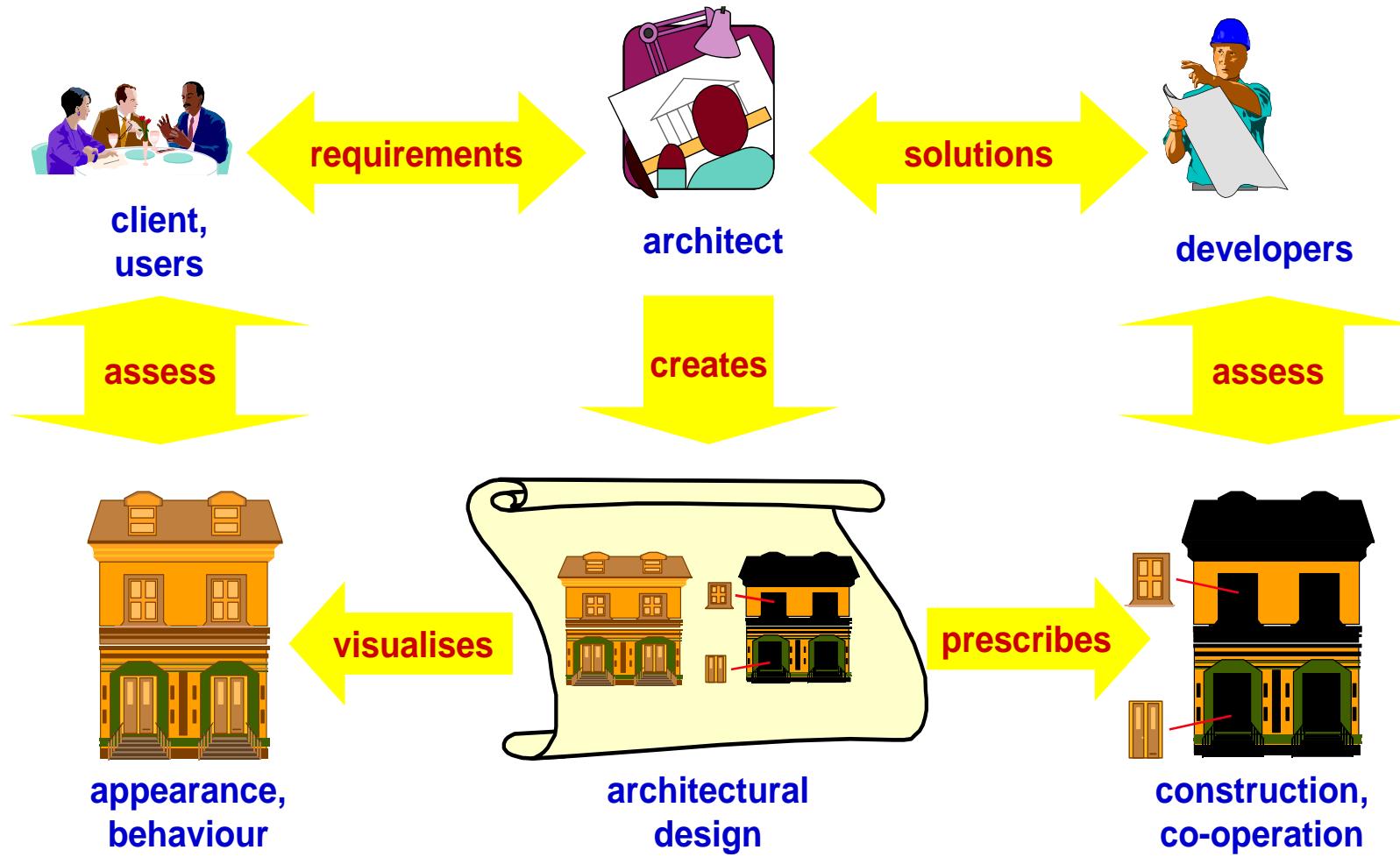
**ING. Del sw AA 2122**  
**Angelo Gargantini**

# Overview

- **What is it, why bother?**
- Architecture Design
- Viewpoints and view models
- Architectural styles
- Architecture assessment
- Role of the software architect



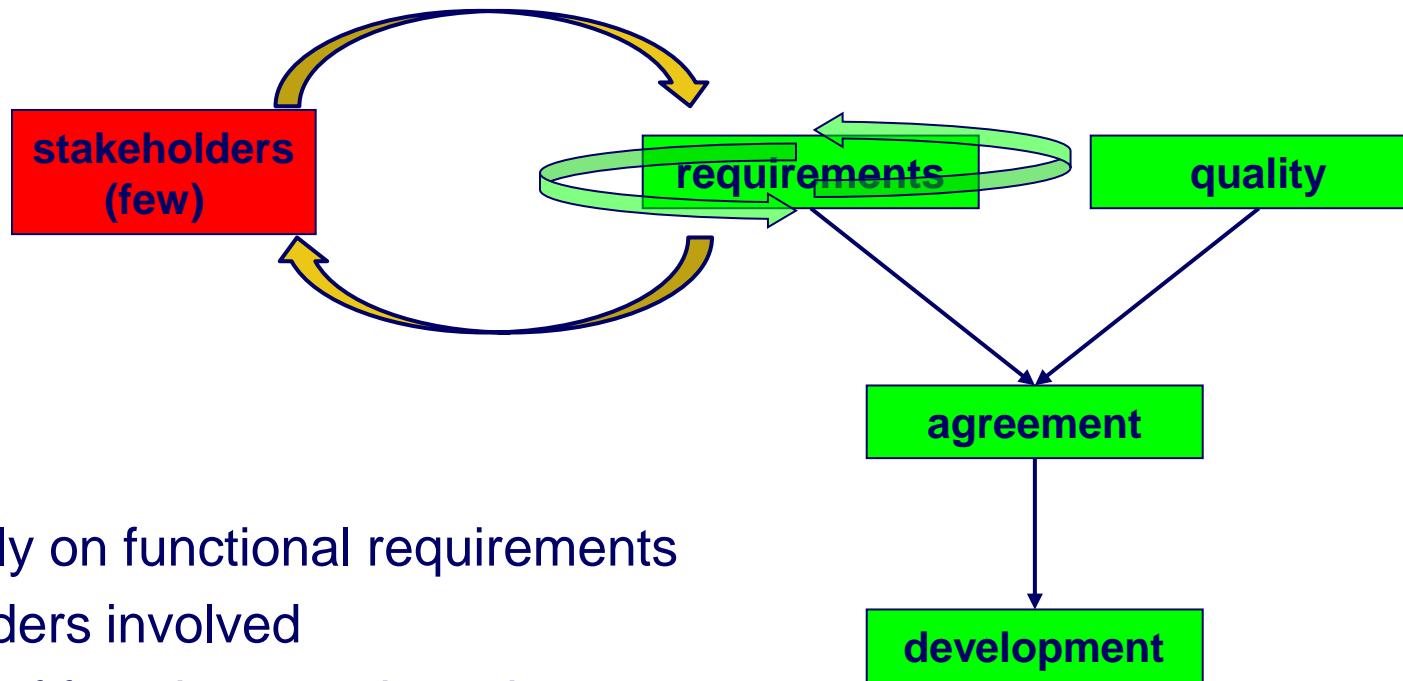
# The Role of the Architect



# What is sw architecture

- During the design phase, the system is decomposed into a number of interacting components.
- The top-level decomposition of a system into major components together with a characterization of how these components interact, is called its **software architecture**.
- Viewed this way, software architecture is synonymous with global design.

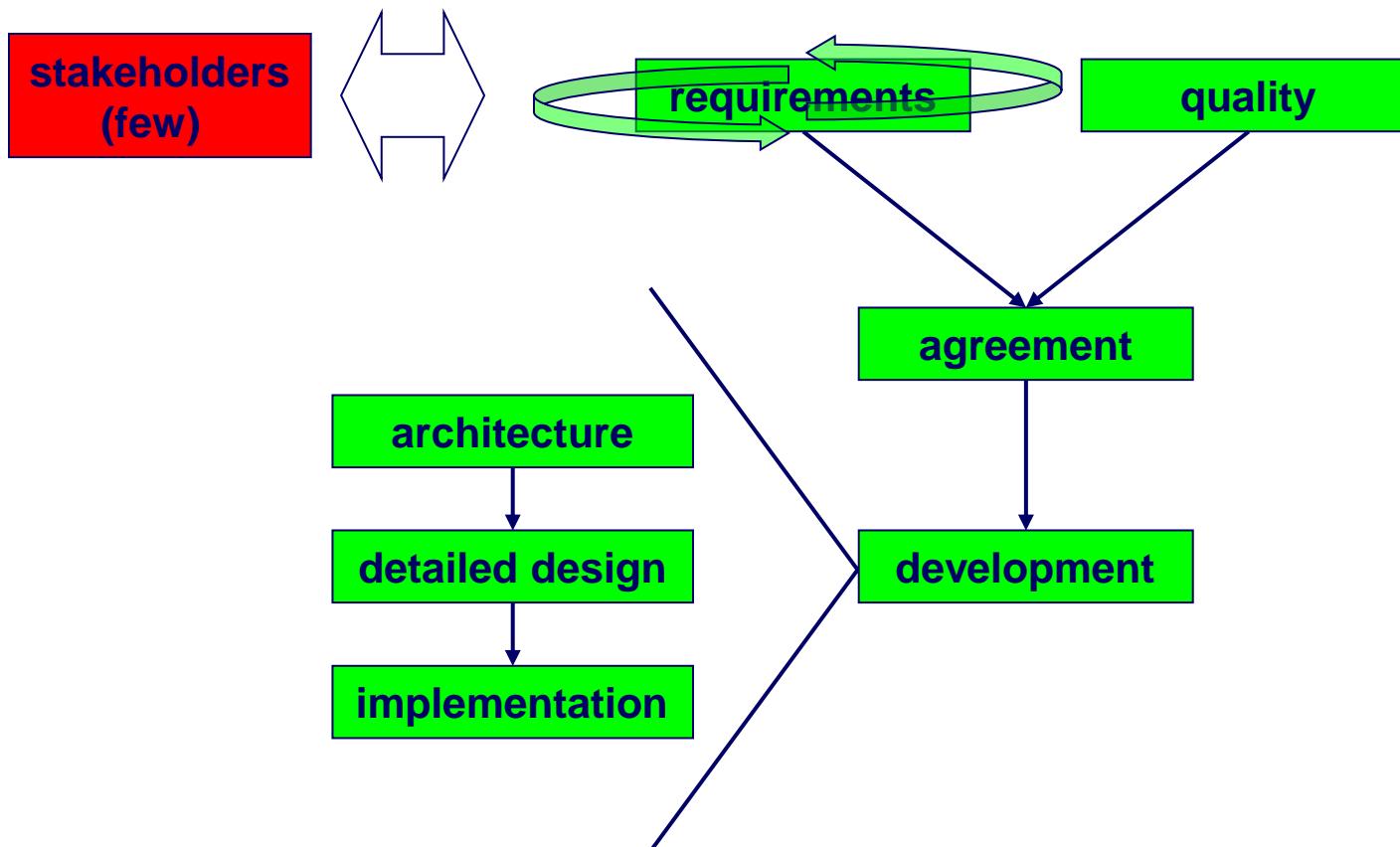
# Pre-architecture life cycle



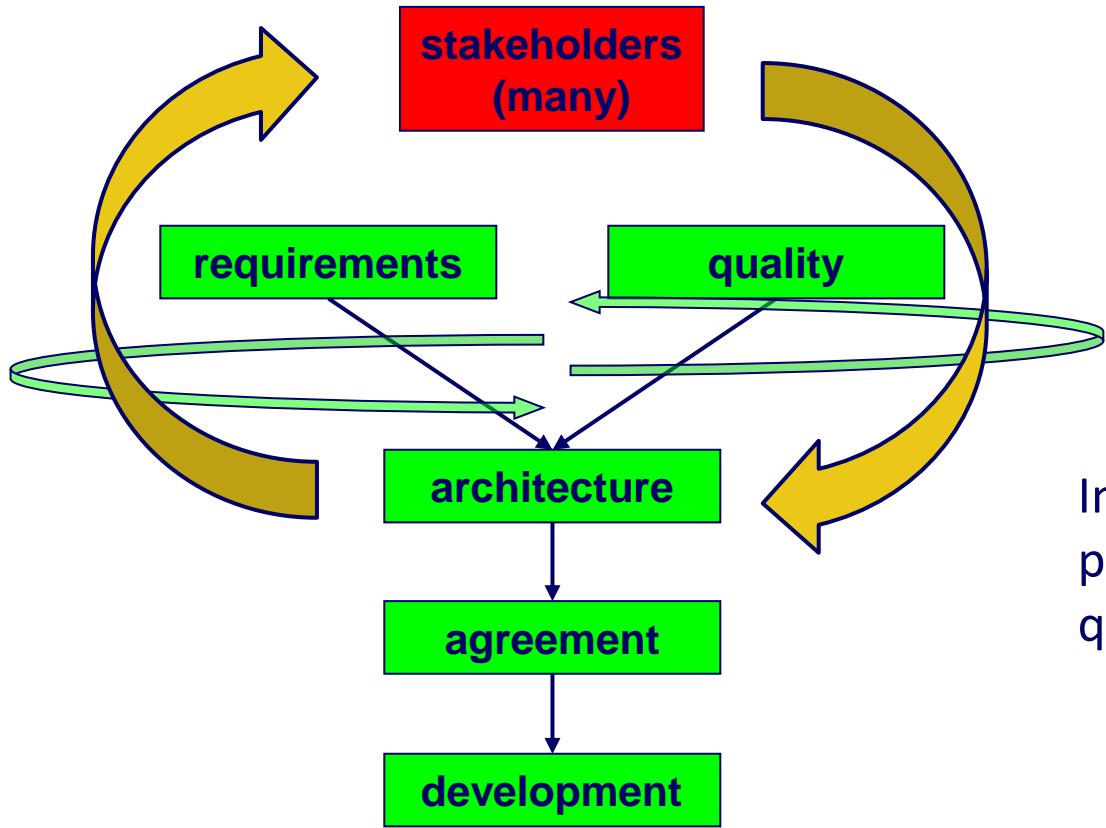
- Iteration mainly on functional requirements
- Few stakeholders involved
- No balancing of functional and quality requirements

# Adding architecture, the easy way

In traditional models, iteration only concerns functional requirements. Once the functional requirements are agreed upon, design starts.



# Architecture in the life cycle



In process models that include a software architecture phase, iteration involves both functional and quality requirements.

# **Characteristics – with sw architecture phase**

- **Iteration on both functional and quality requirements**
- **Many stakeholders involved**
- **Balancing of functional and quality requirements**

# Why Is Architecture Important?



- **Architecture is the vehicle for stakeholder communication**
- **Architecture manifests the earliest set of design decisions**
  - Constraints on implementation
  - Dictates organizational structure
  - Inhibits or enable quality attributes
- **Architecture is a transferable abstraction of a system**
  - Product lines share a common architecture
  - Allows for template-based development
  - Basis for training

# Where did it start?



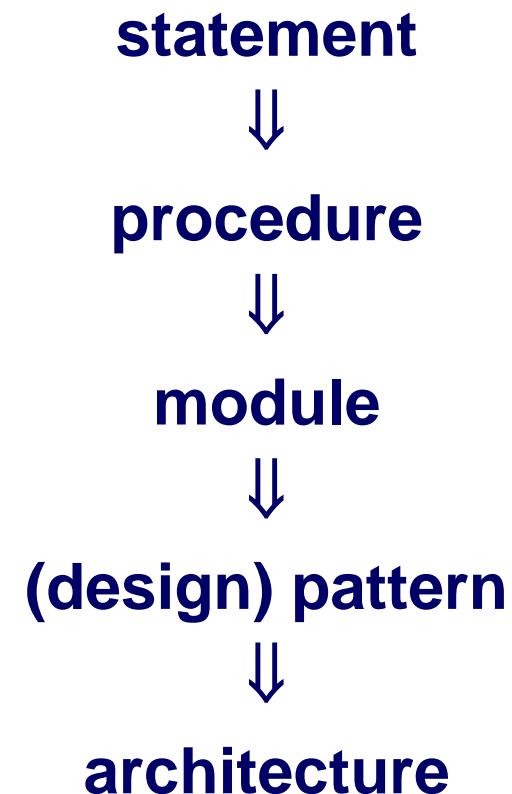
- 1992: Perry & Wolf
- 1987: J.A. Zachman; 1989: M. Shaw
- 1978/79: David Parnas, program families
- 1972 (1969): Edsger Dijkstra, program families
- 1969: I.P. Sharp @ NATO Software Engineering conference:  
**“I think we have something in addition to software engineering [...] This is the subject of software architecture. Architecture is different from engineering.”**

# Software architecture, definition (1)

**The architecture of a software system defines that system in terms of computational components and interactions among those components.**

**(from Shaw and Garlan, *Software Architecture, Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.)**

# Software Architecture



## Software Architecture, definition (2)

**The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.**

**(from Bass, Clements, and Kazman, *Software Architecture in Practice*, SEI Series in Software Engineering. Addison-Wesley, 2003.)**

# Software Architecture

- **Important issues raised in this definition:**
  - multiple system structures;
  - externally visible (observable) properties of components.
- **The definition does not include:**
  - the process;
  - rules and guidelines;
  - architectural styles.

# Architectural Structures

- **module structure**
- **conceptual, or logical structure**
- **process, or coordination structure**
- **physical structure**
- **uses structure**
- **calls structure**
- **data flow**
- **control flow**
- **class structure**



## **Software Architecture, definition (3)**

**Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution**

**(from IEEE Standard on the Recommended Practice for Architectural Descriptions, 2000.)**

# Software Architecture

- **Architecture is *conceptual*.**
- **Architecture is about *fundamental* things.**
- **Architecture exists in some *context*.**

## **Other points of view**

- **Architecture is high-level design**
- **Architecture is overall structure of the system**
- **Architecture is the structure, including the principles and guidelines governing their design and evolution over time**
- **Architecture is components and connectors**

# Software Architecture & Quality

- **The notion of quality is central in software architecting: a software architecture is devised to gain insight in the qualities of a system at the earliest possible stage.**
- **Some qualities are observable via execution: performance, security, availability, functionality, usability**
- **And some are not observable via execution: modifiability, portability, reusability, integrability, testability**

# Overview

- What is it, why bother?
- **Architecture Design**
  - how to design a good sw architecture?
- Viewpoints and view models
- Architectural styles
- Architecture assessment
- Role of the software architect



# Attribute-Driven Design (Bass et al, Ch 7)

- a top-down decomposition process
- **Choose module to decompose**
- **Refine this module:**
  - choose architectural drivers (**quality** is driving force)
  - choose pattern that satisfies drivers
  - apply pattern
- **Repeat steps**

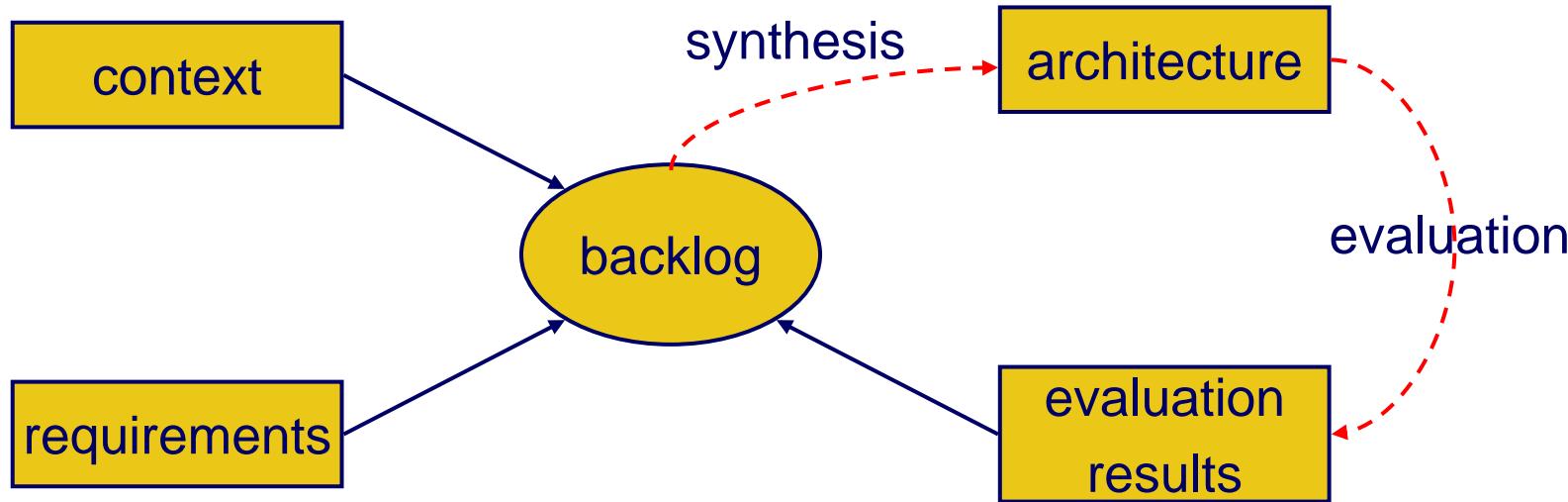
## Example ADD iterations

- **Top-level: usability ⇒ separate user interface ⇒ See three tier architecture**
- **Lower-level, within user interface: security ⇒ authenticate users**
- **Lower-level, within data layer: availability ⇒ active redundancy**

# **Generalized model**

- **Understand problem**
- **Solve it**
- **Evaluate solution**

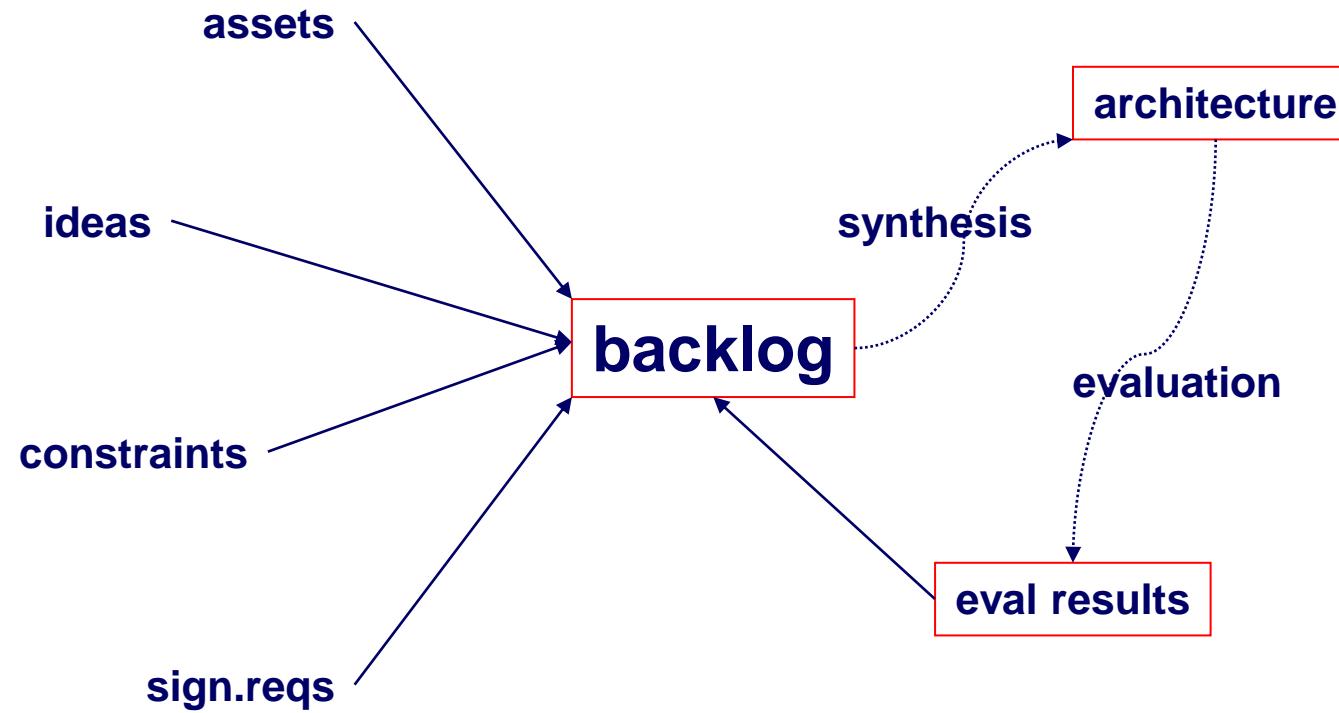
# Global workflow in architecture design



Three inputs to the backlog:

- **Context:** ideas the architect may have, available assets that can be used, constraints set, and so on
- **requirements** constitute another important input.
- The result of this transformation is evaluated (usually rather informally) and this **evaluation** may in turn change the contents of the backlog.

## Generalized model (cont'd)

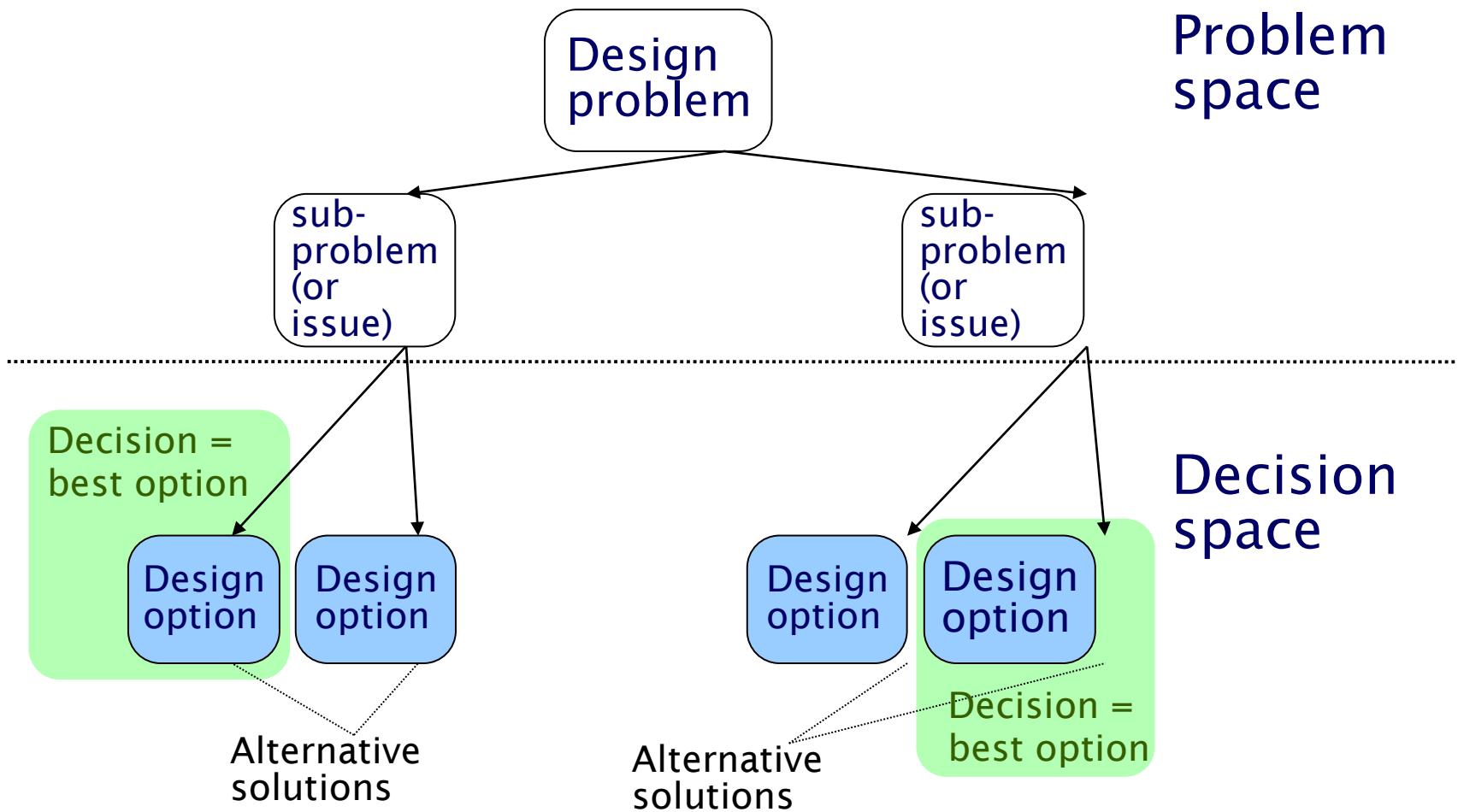


# Design issues, options and decisions

## A designer is faced with a series of design issues

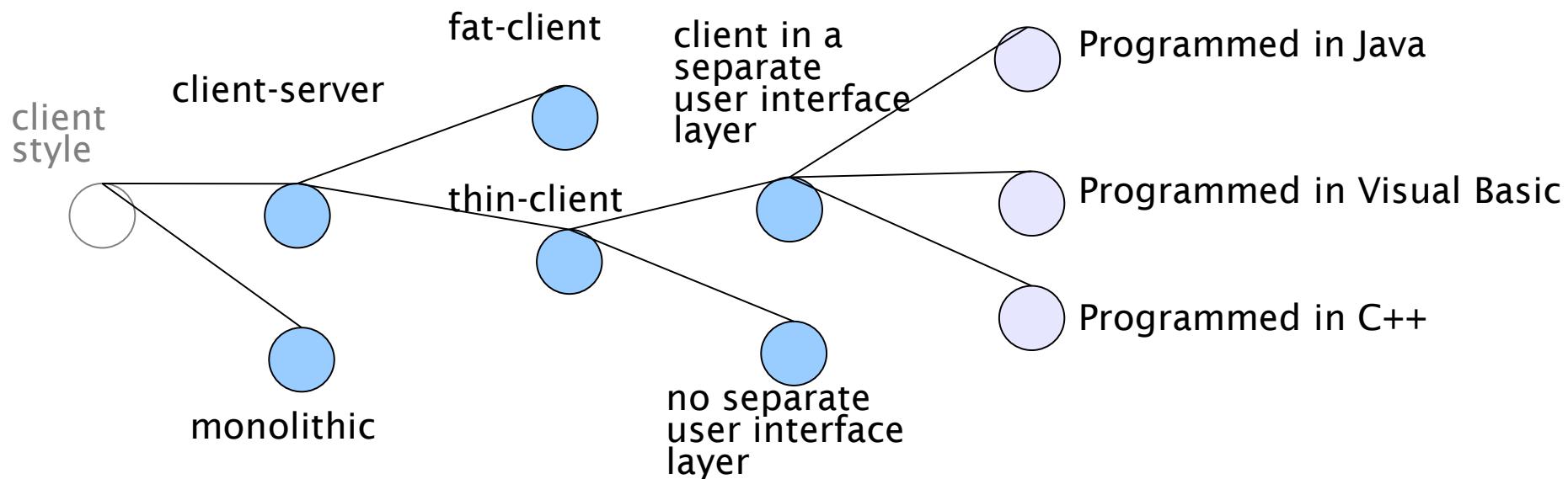
- These are sub-problems of the overall design problem.
- Each issue normally has several alternative solutions (or design options)
- The designer makes a design decision to resolve each issue.
  - This process involves choosing the best option from among the alternatives.

# Taking decisions



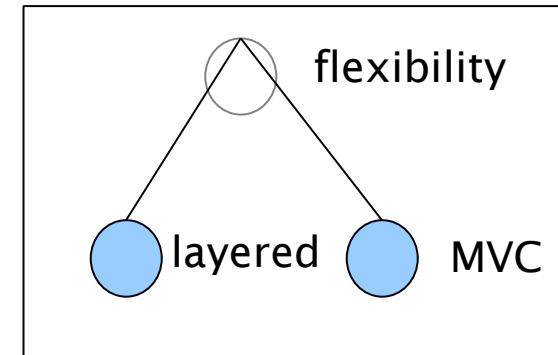
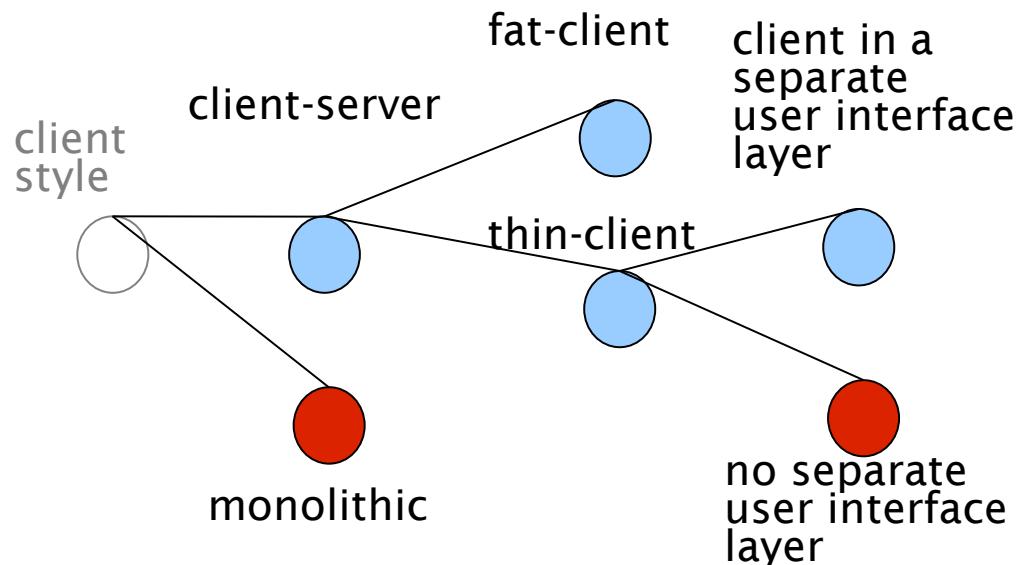
# Decision space

The space of possible designs that can be achieved by choosing different sets of alternatives.



# Tree or graph?

- Issues and options are not independent ...



# More than just IT

- **Technical and non-technical issues and options are intertwined**
  - Architects deciding on the type of database versus
  - Management deciding on new strategic partnership or
  - Management deciding on budget

# Types of decisions

- **Implicit, undocumented**
  - Unaware, tacit, of course knowledge
- **Explicit, undocumented**
  - Vaporizes over time
- **Explicit, explicitly undocumented**
  - Tactical, personal reasons
- **Explicit, documented**
  - Preferred, exceptional situation

# Why is documenting design decisions important?

- Prevents repeating (expensive) past steps
- Explains why this is a good architecture
- Emphasizes qualities and criticality for requirements/goals
- Provides context and background

# Uses of design decisions

- **Identify key decisions for a stakeholder**
  - Make the key decisions quickly available. E.g., introducing new people and make them up2date.
  - ..., Get a rationale, Validate decisions against reqs
- **Evaluate impact**
  - If we want to change an element, what are the elements impacted (decisions, design, issues)?
  - ..., Cleanup the architecture, identify important architectural drivers

# Elements of a design decision

- **Issues:** design issues being addressed
- **Decision**
- **Status:** e.g., pending, approved
- **Assumptions:** underlying assumptions
- **Alternatives**
- **Rationale;** the *why* of the decision taken
- **Implications:** e.g. need for further decisions

# Example of design decisions

Element	Description
Issues	The system has to be structured so that it is maintainable, reusable, and robust.
Decision	A three-tier architecture, consisting of a presentation layer, a business logic layer, and a data management layer.
Status	Approved.
Assumptions	The system has no hard real-time requirements.
Alternatives	A service-oriented architecture (SOA) or a different type of X-tier architecture (e.g. one with a fat client including both presentation and business logic, and a data management tier).
Rationale	Maintenance is supported and extensions are easy to realize because of the loose coupling between layers. Both the presentation layer and the data management layer can be reused in other applications. Robustness is supported because the different layers can easily be split over different media and well-defined layer interfaces allow for smoother testing.
Implications	Performance is hampered since all layers have to be gone through for most user actions.
Notes	None.

# Pointers on design decisions

- Hofmeister et al, Generalizing a Model of Software Architecture Design from Five Industrial Approaches, Journal of Systems and Software, 2007
- Tyree and Ackerman, Architecture decisions: demystifying architecture, IEEE Software, vol. 22(2), 2005.
- Kruchten, Lago and van Vliet, Building up and exploiting architectural knowledge, WICSA, 2005.
- Lago and van Vliet, Explicit assumptions enrich architectural models, ICSE, 2005.

# Overview



- What is it, why bother?
- Architecture Design
- **Viewpoints and view models**
- Architectural styles
- Architecture assessment
- Role of the software architect

# **Software design in UML**

- **Class diagrams, state diagrams, sequence diagram, etc**
- **Who can read those diagrams?**
- **Which type of questions do they answer?**
- **Do they provide enough information?**

# Who can read those diagrams?

- Designer, programmer, tester, maintainer, etc.
- Client?
- User?

# **Which type of questions do they answer?**

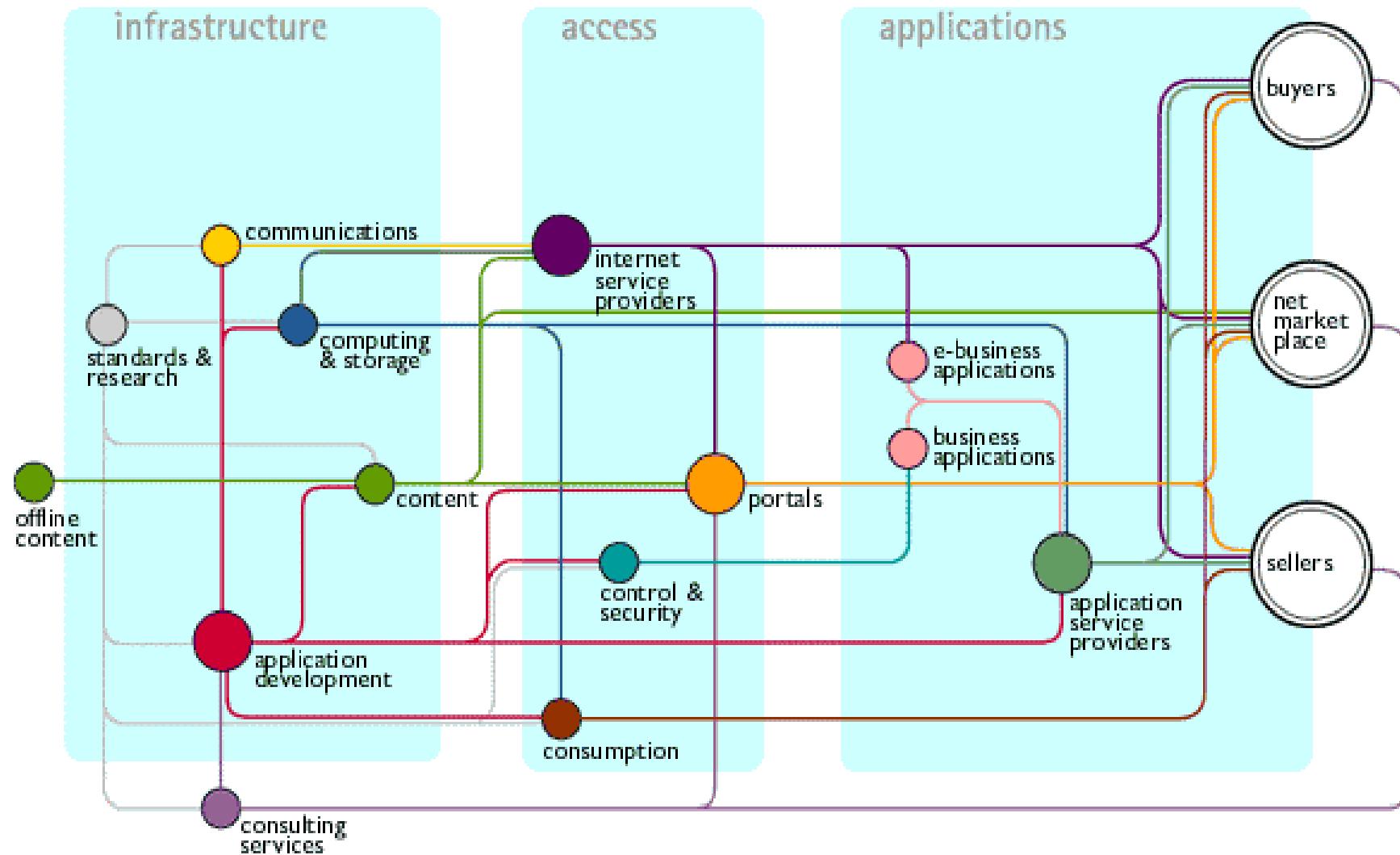
- **How much will it cost?**
- **How secure will the system be?**
- **Will it perform?**
- **How about maintenance cost?**
- **What if requirement A is replaced by requirement B?**

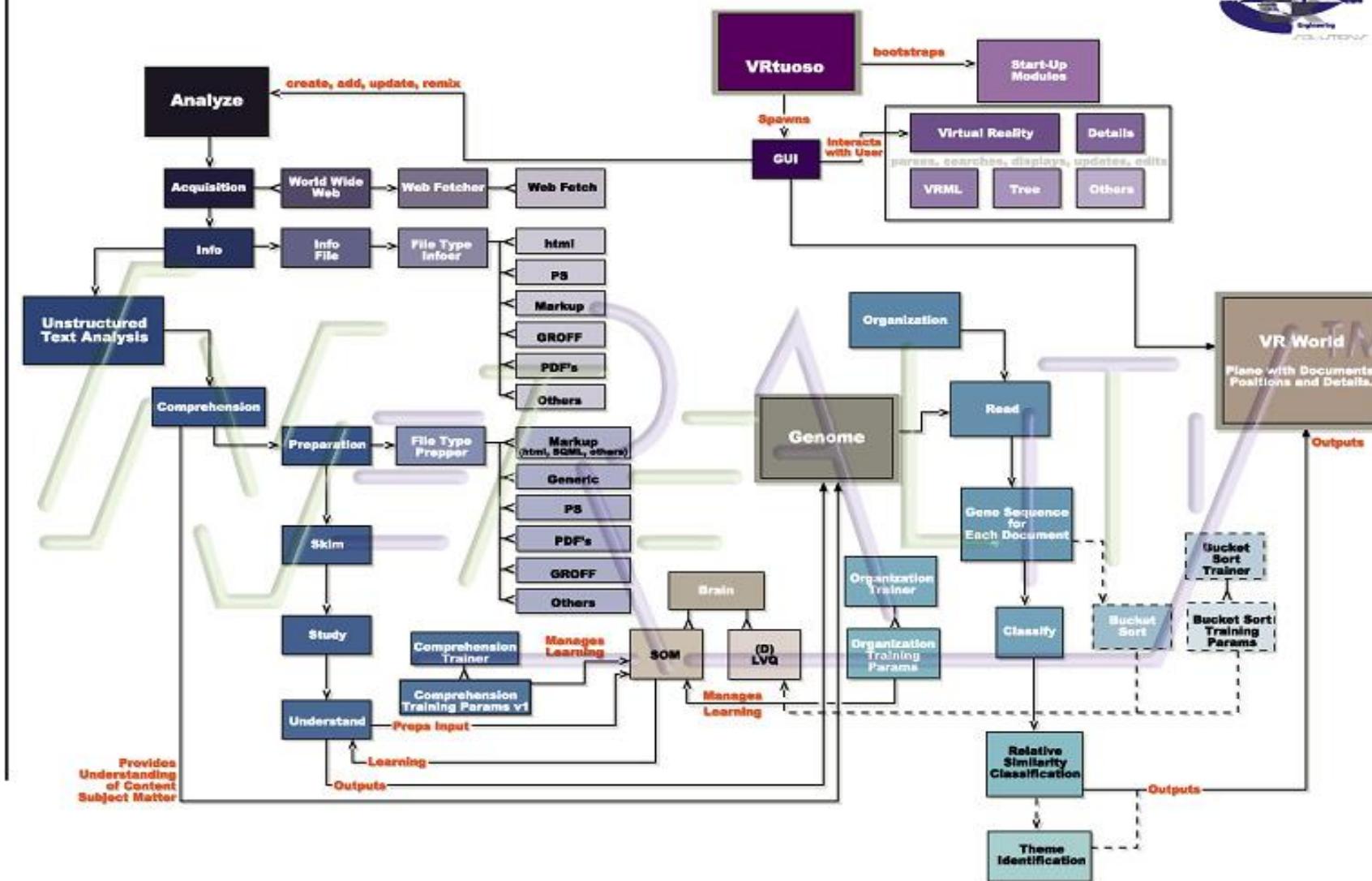
# Analogy with building architecture

- Overall picture of building (client)
- Front view (client, “beauty” committee)
- Separate picture for water supply (plumber)
- Separate picture for electrical wiring (electrician)
- etc

# Architecture presentations in practice

- **By and large two flavors:**
  - Powerpoint slides – for managers, users, consultants, etc
  - UML diagrams, for technicians
- **A small sample ...**





## ARCHITECTURAL OVERVIEW

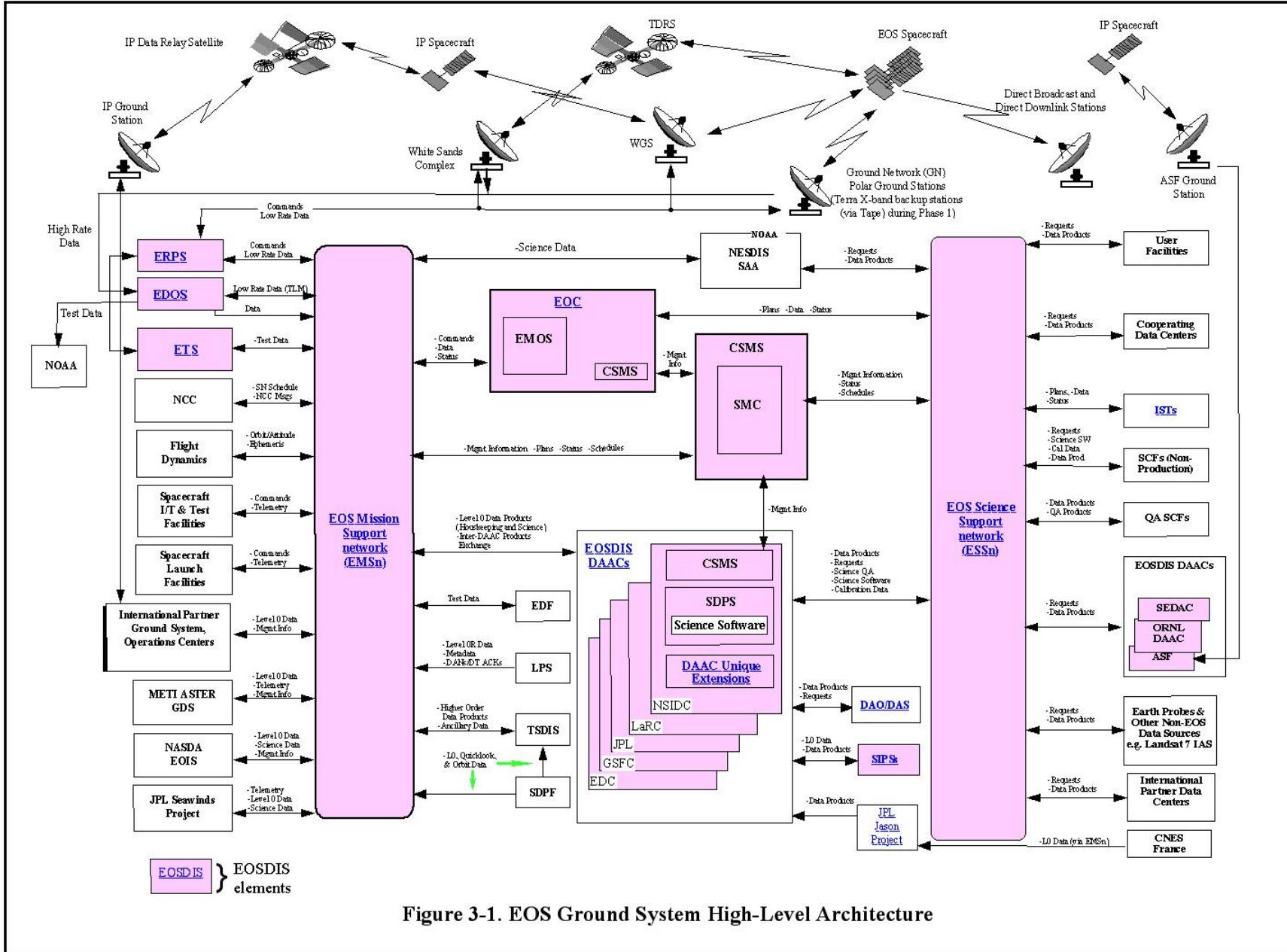
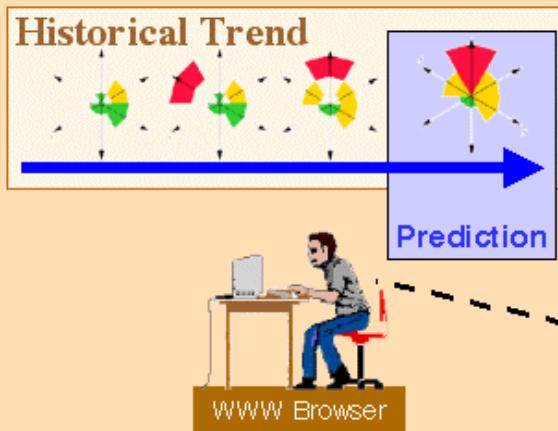


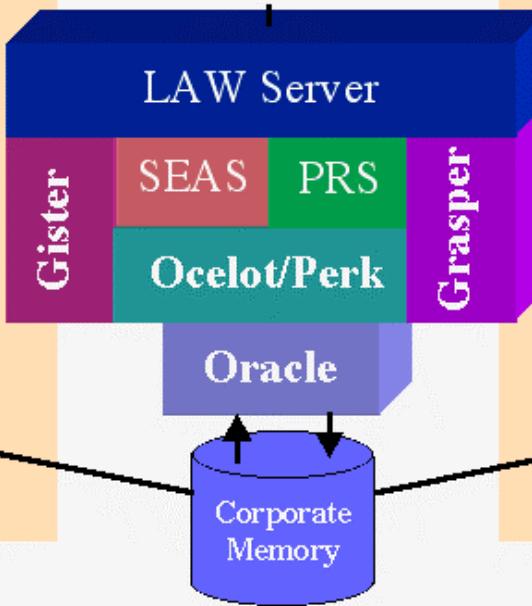
Figure 3-1. EOS Ground System High-Level Architecture

## Historical Perspective



**Collaborating with Historical Analysts**

**Historical Matches/Patterns/Strategies**

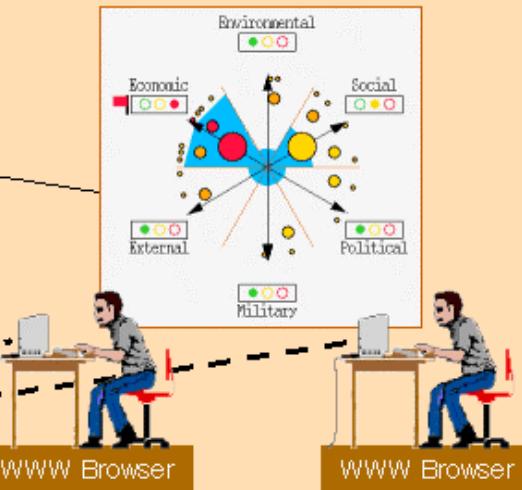


## Current Perspective



**Decision Making**

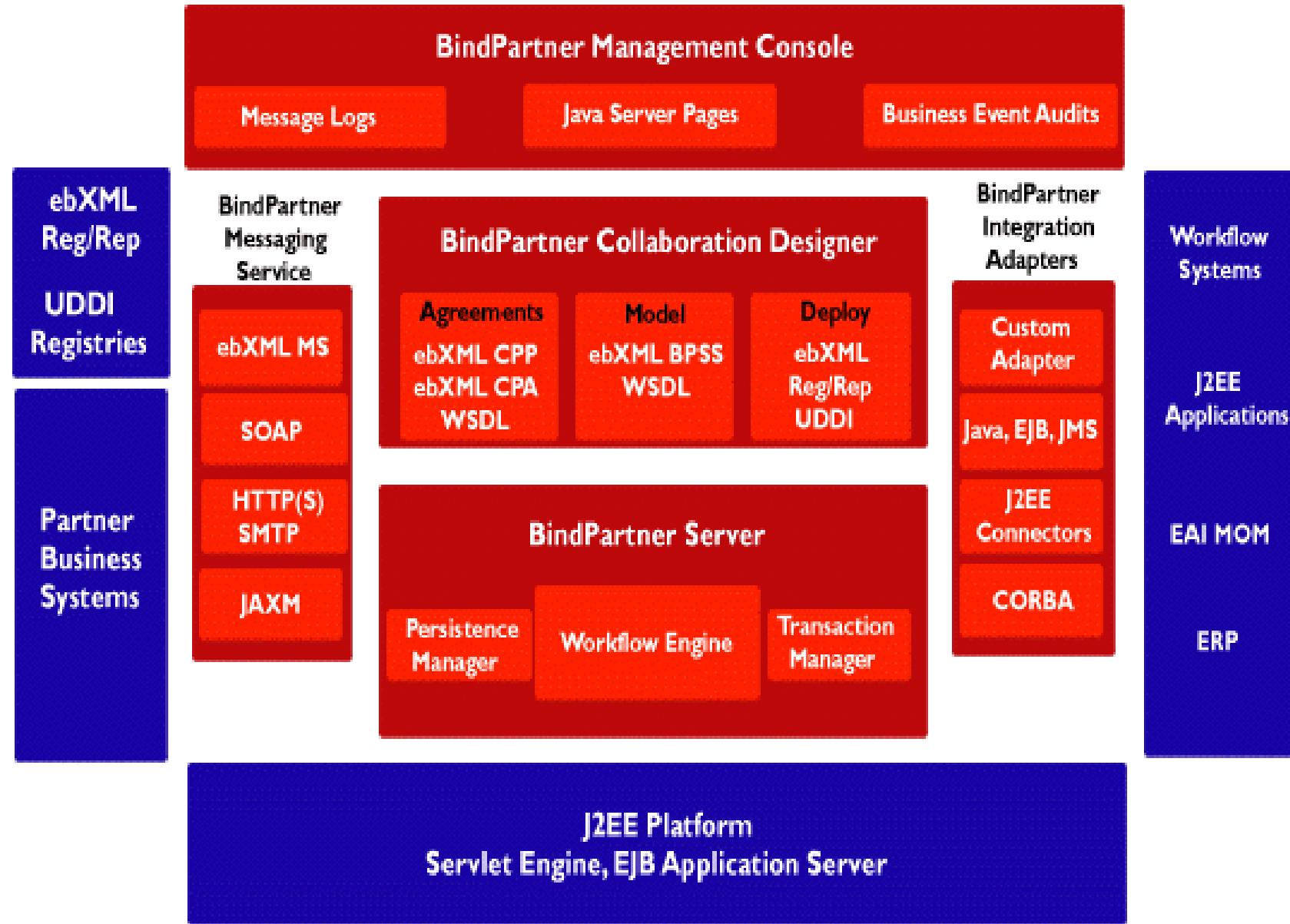
**WWW**



**Collaborating with Contemporary Analysts**

**Current Matches/Patterns/Strategies**

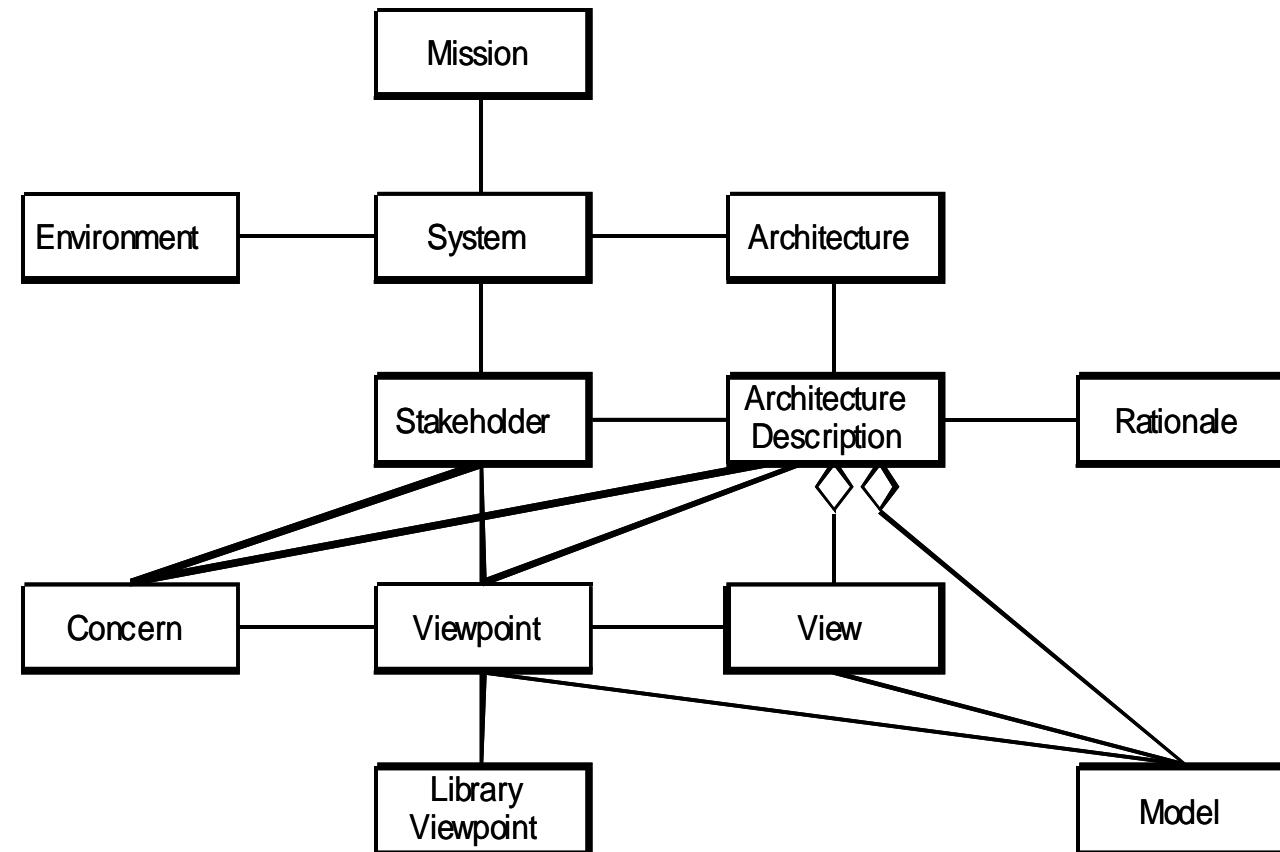




**So, ...**

- **Different representations**
- **For different people**
- **For different purposes**
- **These representations are both descriptive and prescriptive**

# IEEE model for architectural descriptions



## Some terms (from IEEE standard)

- **System stakeholder (parti interessate):** an individual, team, or organization (or classes hereof) with interests in, or concerns relative to, a system.
- **View:** a representation of a whole system from the perspective of a related set of concerns.
- **Viewpoint:** A viewpoint establishes the purposes and audience for a view and the techniques or methods employed in constructing a view.

# **Stakeholders**

- **Architect**
- **Requirements engineer**
- **Designer (also of other systems)**
- **Implementor**
- **Tester, integrator**
- **Maintainer**
- **Manager**
- **Quality assurance people**

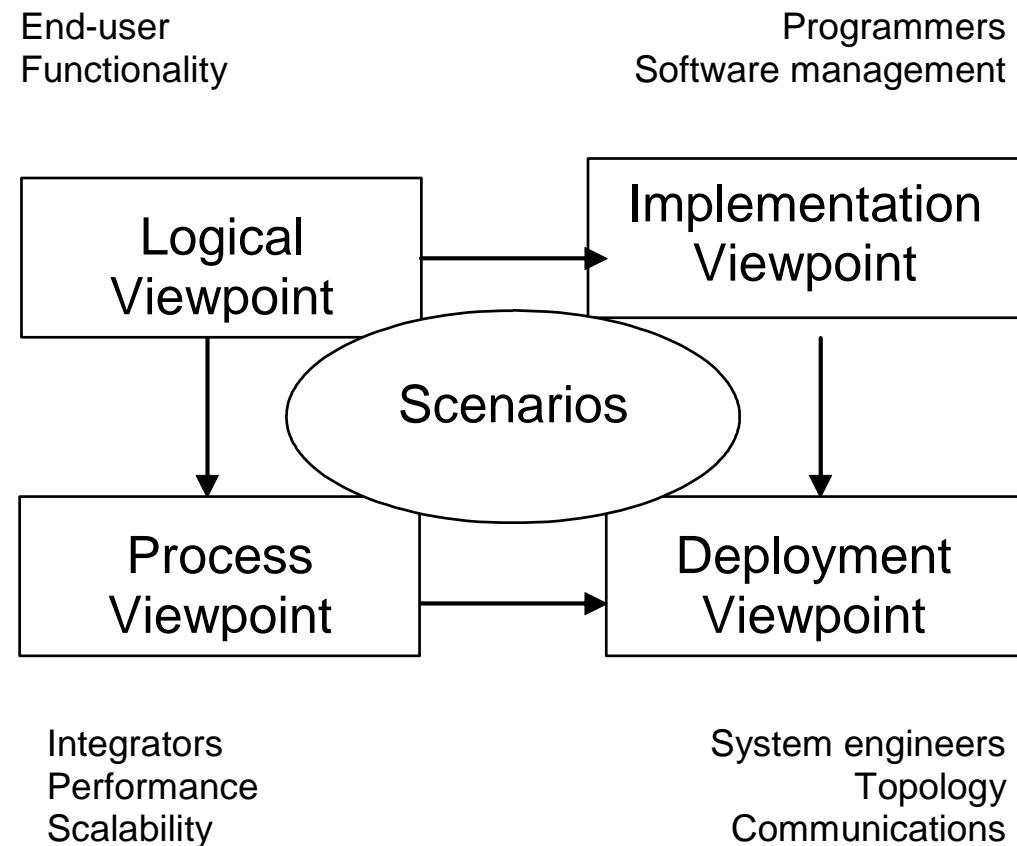
# **Viewpoint specification**

- **Viewpoint name**
- **Stakeholders addressed**
- **Concerns addressed**
- **Language, modeling techniques**

# Kruchten's 4+1 view model

Many organizations have developed their own set of library viewpoints

A well-known set of library viewpoints is known as the '4 + 1 model' (Kruchten, [1995](#))



## 4 + 1: Logical Viewpoint

- **The logical viewpoint supports the functional requirements, i.e., the services the system should provide to its end users.**
- **Typically, it shows the key abstractions (e.g., classes and interactions amongst them).**

## 4 + 1: Process Viewpoint

- **Addresses concurrent aspects at runtime (tasks, threads, processes and their interactions)**
- **It takes into account some nonfunctional requirements, such as performance, system availability, concurrency and distribution, system integrity, and fault-tolerance.**

## 4 + 1: Deployment Viewpoint

- The deployment viewpoint defines how the various elements identified in the logical, process, and implementation viewpoints—networks, processes, tasks, and objects—must be mapped onto the various nodes.
- It takes into account the system's nonfunctional requirements such as system availability, reliability (fault-tolerance), performance (throughput), and scalability.

## **4 + 1: Implementation Viewpoint**

- **The implementation viewpoint focuses on the organization of the actual software modules in the software-development environment.**
- **The software is packaged in small chunks-program libraries or subsystems-that can be developed by one or more developers.**

## 4 + 1: Scenario Viewpoint

- **The scenario viewpoint consists of a small subset of important scenarios (e.g., use cases) to show that the elements of the four viewpoints work together seamlessly.**
- **This viewpoint is redundant with the other ones (hence the "+1"), but it plays two critical roles:**
  - it acts as a driver to help designers discover architectural elements during the architecture design;
  - it validates and illustrates the architecture design, both on paper and as the starting point for the tests of an architectural prototype.

# Architectural views from Bass et al (view = representation of a structure)

- **Module views**
  - Module is unit of implementation
  - Decomposition, uses, layered, class
- **Component and connector (C & C) views**
  - These are runtime elements
  - Process (communication), concurrency, shared data (repository), client-server
- **Allocation views**
  - Relationship between software elements and environment
  - Work assignment, deployment, implementation

# Examples of viewpoint

- . **Module viewpoints** give a static view of the system. They are usually depicted in the form of box-and-line diagrams where the boxes denote system components and the lines denote some relation between those components.
- . **Component-and-connector** viewpoints give a dynamic view of the system, i.e. they describe the system in execution. Again, they are usually depicted as box-and-line diagrams.
- . **Allocation** viewpoints give a relation between the system and its environment, such as who is responsible for which part of the system.

# Module views

- **Decomposition:** units are related by “is a submodule of”, larger modules are composed of smaller ones
- **Uses:** relation is “uses” (calls, passes information to, etc). Important for modifiability
- **Layered** is special case of uses, layer  $n$  can only use modules from layers  $< n$
- **Class:** generalization, relation “inherits from”

## Table 11.3 module viewpoints example

Viewpoint	Description
Decomposition	Elements are related by the ‘is a submodule of’ relation. Larger elements are composed of smaller ones. It is the result of a top-down refinement process. The decomposition viewpoint often forms the basis for the project organization and the system’s documentation. It is the viewpoint we are most used to in software design.
Uses	The relation between elements is ‘uses’ (A calls B, A passes information to B, etc.). The uses relation goes back to Parnas ( <a href="#">1972</a> ); see also <a href="#">Chapter 12</a> . The uses relation is important when we want to assess modifiability: if an element is changed, all elements it is used by potentially have to be changed as well. It is also useful to determine incremental subsets of a system: if an element is in a given subset, all elements it uses must also be in that subset.
Layered	This is a special case of the uses viewpoint. It is useful if we want to view the system as a series of layers, where elements from layer n can only use elements from layers < n. Layers can often be interpreted as virtual machines.
Class	Elements are described as a generalization of other elements. The relation between elements is ‘inherits from’. It is obviously most applicable for object-oriented systems.

# Component and connector views

- **Process:** units are processes, connected by communication or synchronization
- **Concurrency:** to determine opportunities for parallelism (connector = logical thread)
- **Shared data:** shows how data is produced and consumed
- **Client-server:** cooperating clients and servers

## Table 11.4 Component-and-connector viewpoints

Viewpoint	Description
Process	The system is described as a series of processes, connected by communication or synchronization links. It is useful if we want to reason about the performance or the availability of the system.
Concurrency	To determine opportunities for parallelism, a sequence of computations that can be allocated to a separate physical thread later in the design process is collected in a ‘logical thread’. It is used to manage issues related to concurrent execution.
Shared data	Describes how persistent data is produced, stored, and consumed. It is particularly useful if the system centers around the manipulation of large amounts of data. It can be used to assess qualities such as performance and data integrity.
Client–server	Describes a system that consists of cooperating clients and servers. The connectors are the protocols and messages that clients and servers exchange. This viewpoint expresses separation of concerns and physical distribution of processing elements.

# Allocation views

- **Deployment:** how software is assigned to hardware elements
- **Implementation:** how software is mapped onto file structures
- **Work assignment:** who is doing what

## Table 11.5 Allocation viewpoints

Viewpoint	Description
<b>Deployment</b>	Shows how software is assigned to hardware elements and which communication paths are used. It allows one to reason about, e.g., performance, security, and availability.
<b>Implementation</b>	Indicates how software is mapped onto file structures. It is used in the management of development activities and for build processes.
<b>Work assignment</b>	Shows who is doing what and helps to determine which knowledge is needed where. For instance, one may decide to assign functional commonality to a single team.

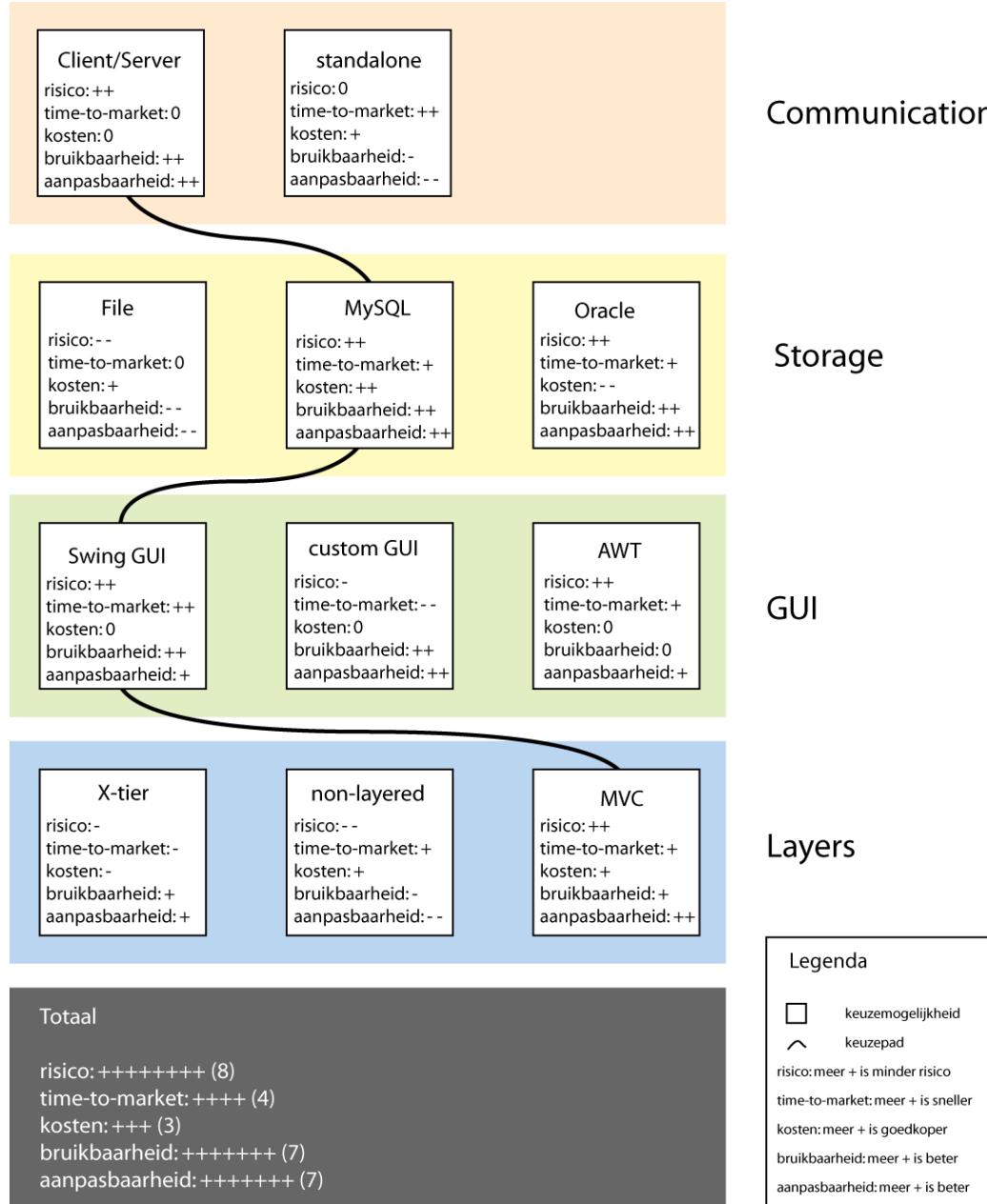
# **How to decide on which viewpoints**

- What are the stakeholders and their concerns?**
- Which viewpoints address these concerns?**
- Prioritize and possibly combine viewpoints**

# Decision visualization



# Business viewpoint

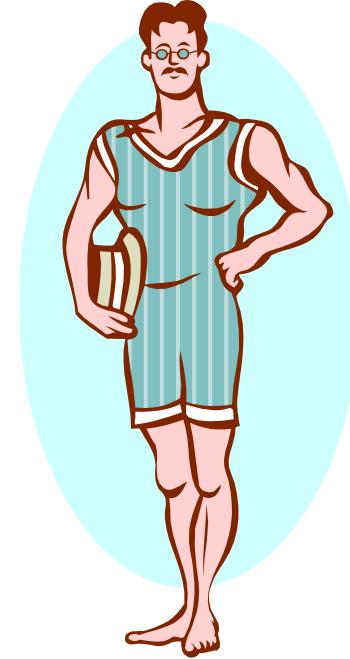


## A caveat on quality

- A view can be used to assess one or more quality attributes
- E.g., some type of module view can be used to assess modifiability
- It should then expose the design decisions that affect this quality attribute

# Overview

- What is it, why bother?
- Architecture Design
- Viewpoints and view models
  
- **Architectural styles**
  
- Architecture assessment
- Role of the software architect



# Architectural styles

- An architectural style is a description of component and connector types and a pattern of their runtime control and/or data transfer.
- Examples:
  - main program with subroutines
  - data abstraction
  - implicit invocation
  - pipes and filters
  - repository (blackboard)
  - layers of abstraction

# Using schemas: pattern, framework, idioms...

- Programming plans, program fragments that correspond to stereotypical actions, and rules that describe programming conventions
  - For example, to compute the sum of a series of numbers, ....
- **Design patterns** are collections of a few modules (or, in object-oriented circles, classes) that are often used in combination, and which together provide a useful abstraction. A design pattern is a recurring solution to a standard problem.
  - Vedremo nel 12.5 con prof Scandurra
- An **application framework** is a partially complete system which needs to be instantiated to obtain a complete system. It describes the architecture of a family of similar systems. It is thus tied to a particular application domain. The best-known examples are frameworks for building user interfaces.
- An **idiom** is a low-level pattern, specific to some programming language
  - Example static counter

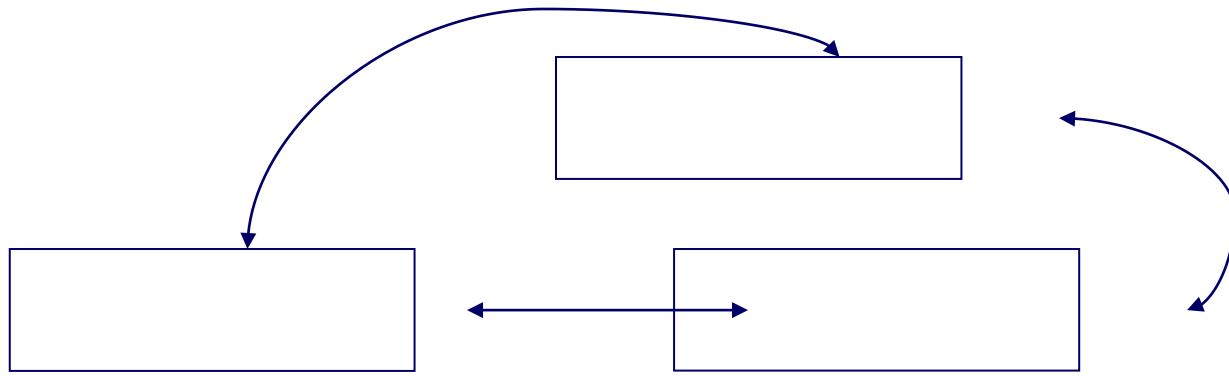
## Alexander's patterns

- There is abundance evidence to show that high buildings make people crazy.
- High buildings have no genuine advantage, except in speculative gains. They are not cheaper, they do not help to create open space, they make life difficult for children, they wreck the open spaces near them. But quite apart from this, empirical evidence shows that they can actually damage people's minds and feelings.
- In any urban area, keep the majority of buildings four stories high or less. It is possible that certain buildings should exceed this limit, but they should never be buildings for human habitation.

# General flavor of a pattern

- **IF you find yourself in <context>, for example <examples>, with <problem>**
- **THEN for some <reasons>, apply <pattern> to construct a solution leading to a <new context> and <other patterns>**

# Components and Connectors



- Components are connected by connectors.
- They are the building blocks with which an architecture can be described.
- No standard notation has emerged yet.

# Types of components

- ***computational***: does a computation of some sort.
  - E.g. function, filter.
- ***memory***: maintains a collection of persistent data.
  - E.g. data base, file system, symbol table.
- ***manager***: contains state + operations. State is retained between invocations of operations.
  - E.g. adt, server.
- ***controller***: governs time sequence of events.
  - E.g. control module, scheduler.

# Types of connectors

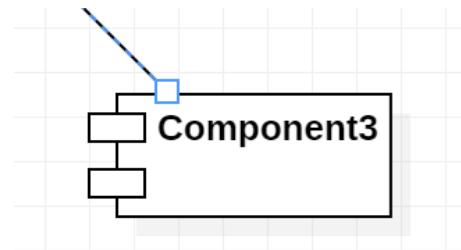
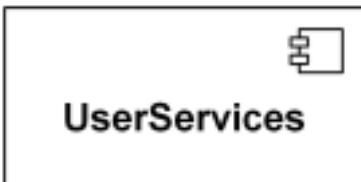
- **procedure call (including Remote Procedure Call)**
  - Control is transferred to another component
- **data flow (e.g. pipes)**
  - Data are transferred from one component to another
- **implicit invocation**
  - A component require a service when an event occur
- **message passing**
  - Both async and sync
- **shared data (e.g. blackboard or shared data base)**
- **Instantiation**

# Component diagram in UML

- A **component** is a class representing a modular part of a system with encapsulated content.
  - A component is shown as a classifier rectangle with the keyword «component».

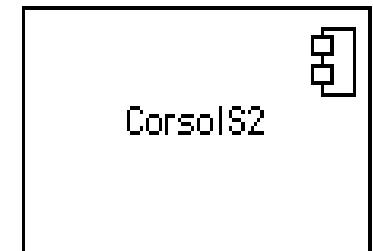


- Optionally, a component icon similar to the UML 1.4 icon can be displayed in the top right corner of the component rectangle. If the icon symbol is shown, the keyword «component» may be omitted.



# COMPONENT NOTATION

- A component is shown as a rectangle with
  - A keyword <<component>>
- Optionally, in the right hand corner a component icon can be displayed
  - A component icon is a rectangle with two smaller rectangles jutting out from the left-hand side
  - This symbol is a visual stereotype
  - The component name
- Components can be labelled with a stereotype there are a number of standard stereotypes ex: <<entity>>, <<subsystem>>



# Component ELEMENTS

- **A component can have**

- Interfaces
  - An interface represents a declaration of a set of operations and obligations
- Usage dependencies
  - A usage dependency is relationship which one element requires another element for its full implementation
- Ports
  - Port represents an interaction point between a component and its environment
- Connectors
  - Connect two components
  - Connect the external contract of a component to the internal structure

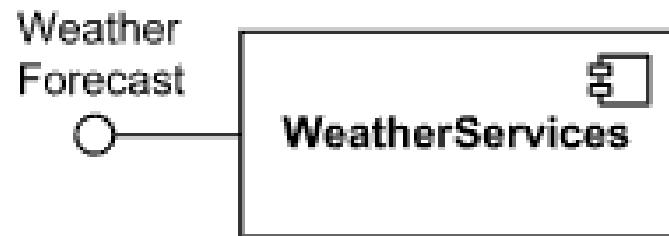
# interfaces

- **A component has its behavior defined in terms of provided interfaces and required interfaces (potentially exposed via ports).**
  - Component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant.

# INTERFACE

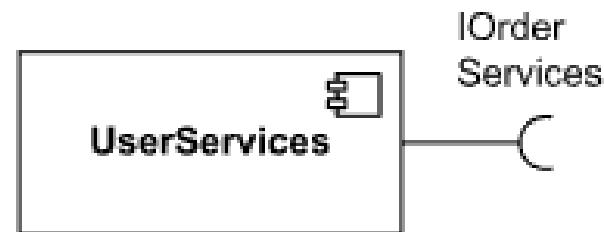
## ▪ A provided interface

- Characterize services that the component offers to its environment
- Is modeled using a ball, labelled with the name, attached by a solid line to the component



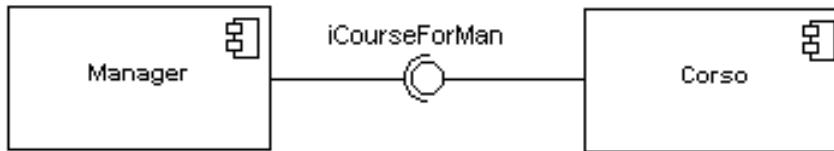
## ▪ A required interface

- Characterize services that the component expects from its environment
- Is modeled using a socket, labelled with the name, attached by a solid line to the component
- In UML 1.x were modeled using a dashed arrow

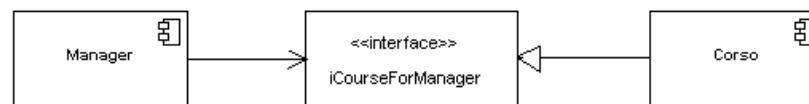


# INTERFACE

- Where two components/classes provide and require the same interface, these two notations may be combined



- The ball-and-socket notation hint at that interface in question serves to mediate interactions between the two components
  - If an interface is shown using the rectangle symbol, we can use an alternative notation, using dependency arrows

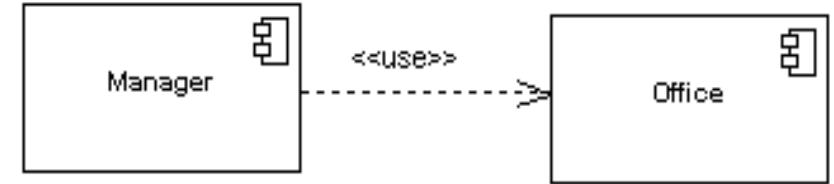


# DEPENDENCIES

- Components can be connected by usage dependencies

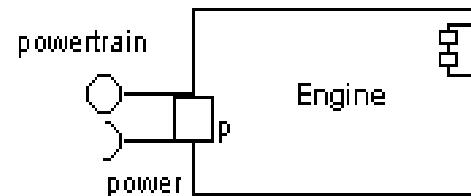
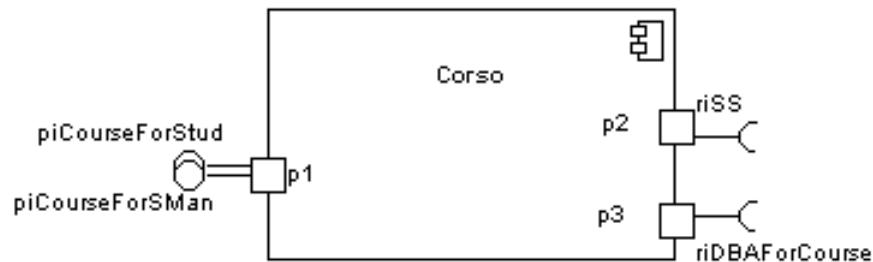
## ▪ Usage Dependency

- A usage dependency is relationship which one element requires another element for its full implementation
- Is a dependency in which the client requires the presence of the supplier
- Is shown as dashed arrow with a <<use>> keyword
- The arrowhead point from the dependent component to the one of which it is dependent

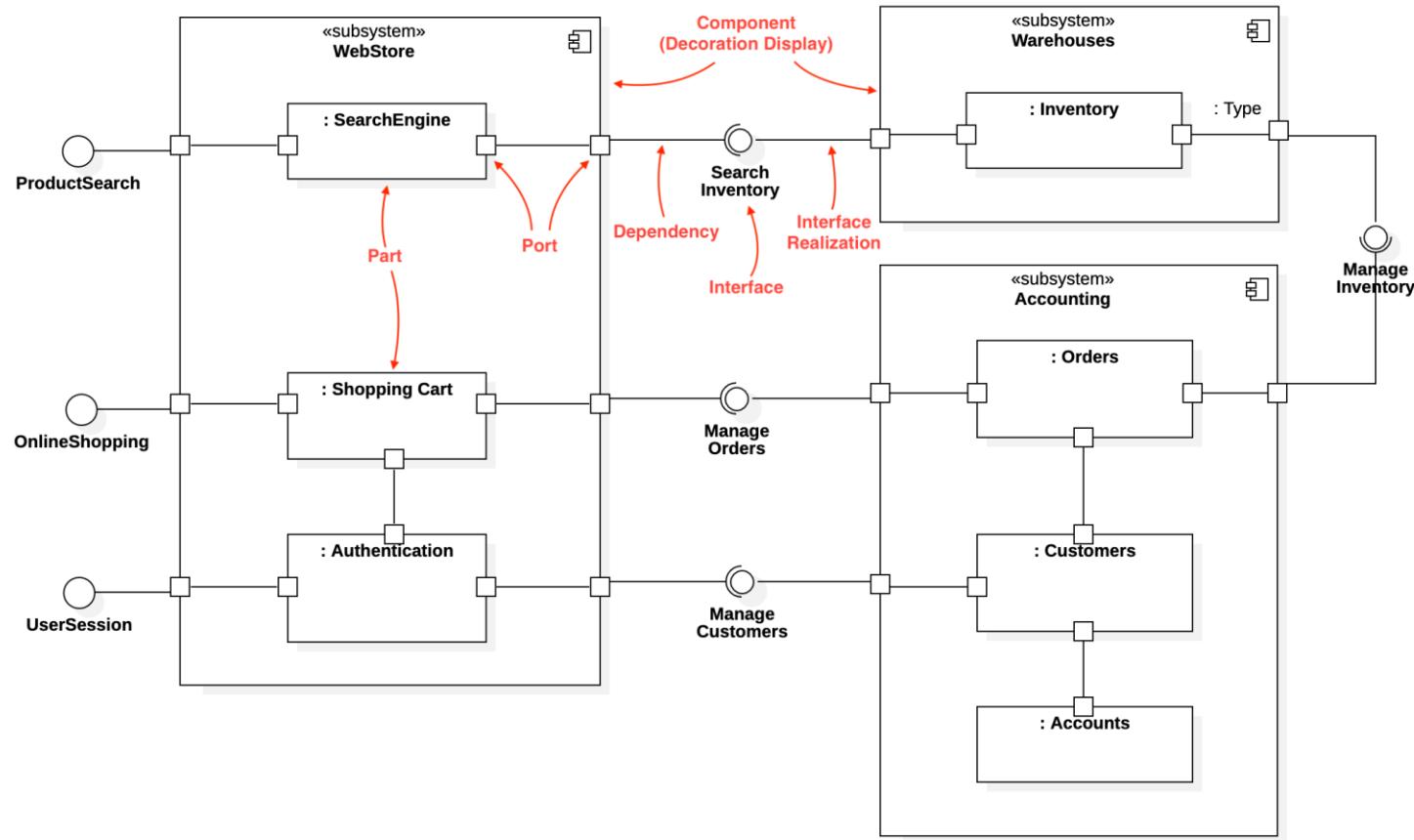


# PORT

- Ports can support unidirectional communication or bi-directional communication



# Complete example



# Framework for describing architectural styles

- ***problem:*** type of problem that the style addresses. Characteristics of the req's's guide the designer in his choice for a particular style.
- ***context:*** characteristics of the environment that constrain the designer, req's imposed by the style.
- ***solution:*** in terms of components and connectors (choice not independent), and control structure (order of execution of components)
- ***variants***
- ***examples***

# Main-program-with-subroutines style

**problem: hierarchy of functions; result of functional decomposition, single thread of control**

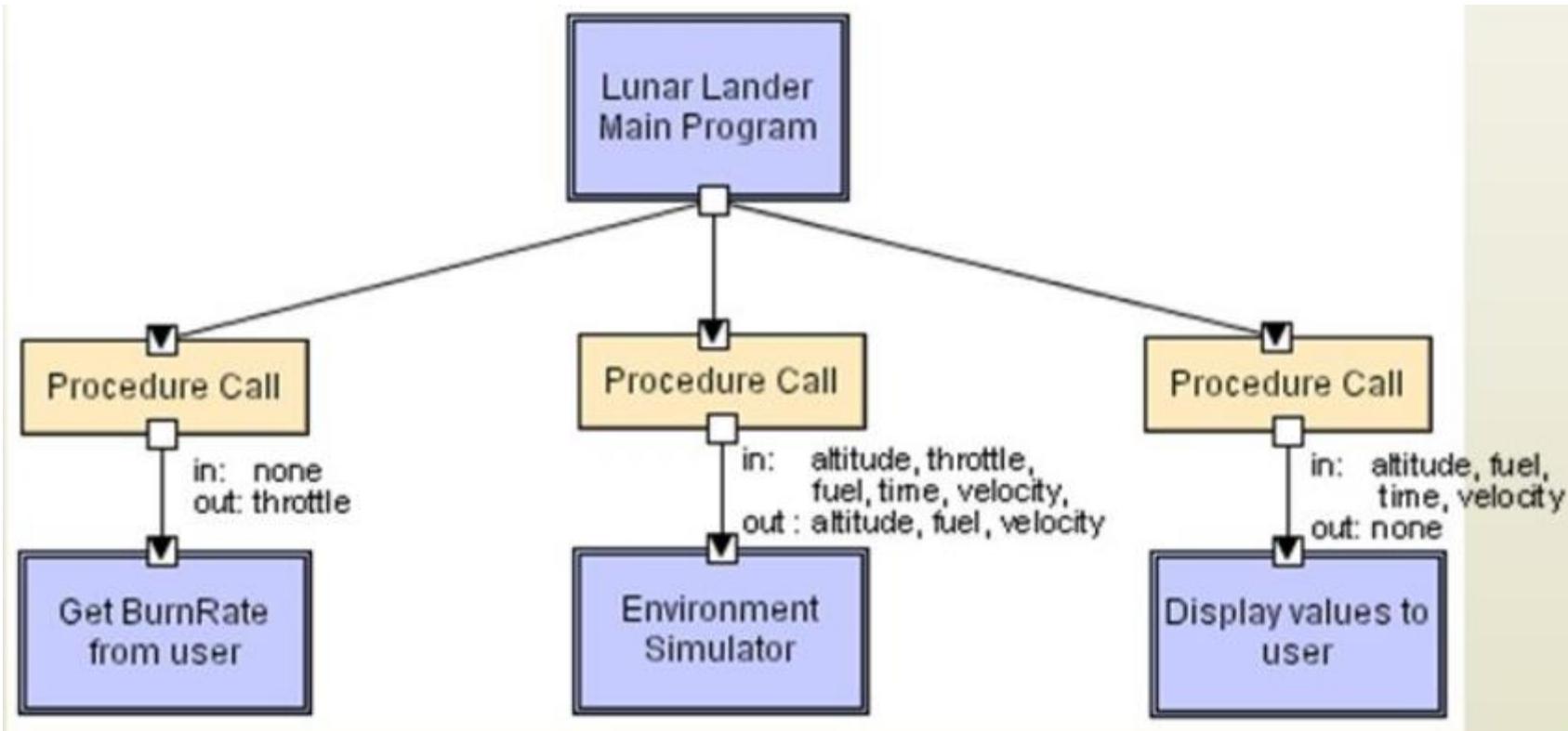
**context: language with nested procedures**

**solution:**

- system model: modules in a hierarchy, may be weak or strong, coupling/cohesion arguments
- components: modules with local data, as well as global data
- connectors: procedure call
- control structure: single thread, centralized control: main program pulls the strings

**variants: OO versus non-OO**

# Example



# **Abstract-data-type style**

**problem: identify and protect related bodies of information. Data representations likely to change.**

**context: OO-methods which guide the design, OO-languages which provide the class-concept**

**solution:**

- system model: component has its own local data (= secret it hides)
- components: managers (servers, objects, adt's)
- connectors: procedure call (message)
- control structure: single thread, usually; control is decentralized

**variants: caused by language facilities**

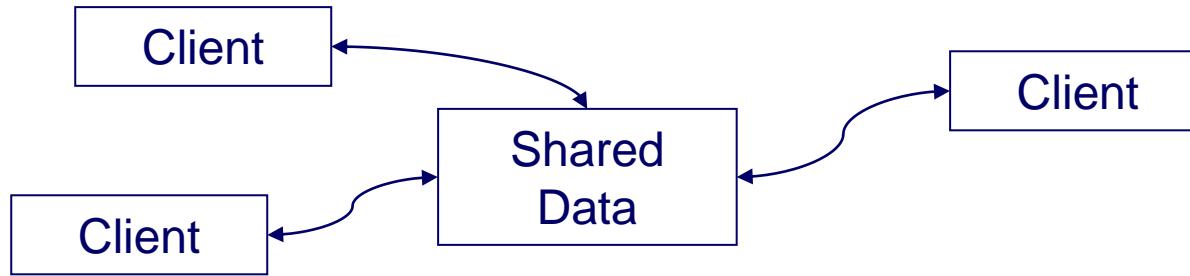
# Implicit-invocation style

- **problem: loosely coupled collection of components. Useful for applications which must be reconfigurable.**
- **context: requires event handler, through OS or language.**
- **solution:**
  - system model: independent, reactive processes, invoked when an event is raised
  - components: processes that signal events and react to events
  - connectors: automatic invocation
  - control structure: decentralized control. Components do not know who is going to react.
- **variants: Tool-integration frameworks, and languages with special features.**

# Pipes-and-filters style

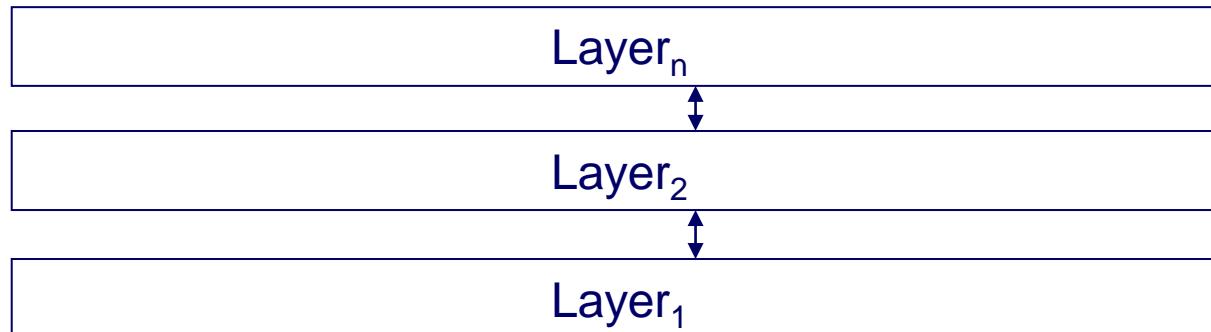
- **problem:** independent, sequential transformations on ordered data. Usually incremental, Ascii pipes.
- **context:** series of incremental transformations. OS-functions transfer data between processes. Error-handling difficult.
- **solution:**
  - system model: continuous data flow; components incrementally transform data
  - components: filters for local processing
  - connectors: data streams (usually plain ASCII)
  - control structure: data flow between components; component has own flow
- **variants:** From pure filters with little internal state to batch processes

# Repository style



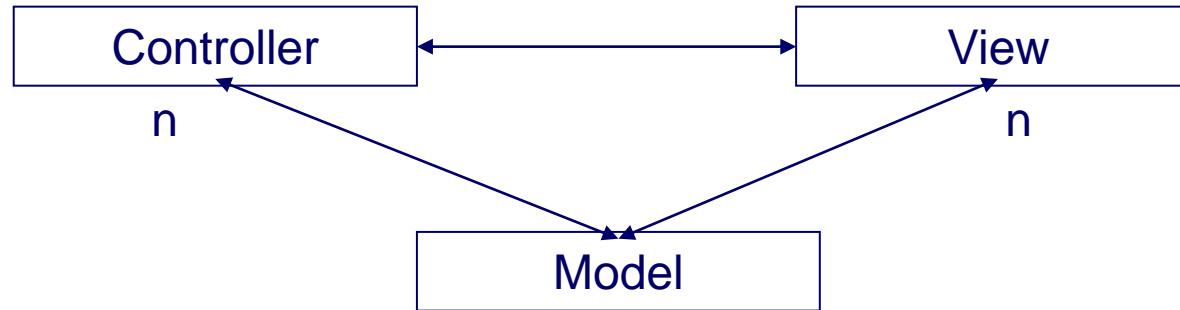
- **problem:** manage richly structured information, to be manipulated in many different ways. Data is long-lived.
- **context:** shared data to be acted upon by multiple clients
- **solution:**
  - system model: centralized body of information. Independent computational elements.
  - components: one memory, many computational
  - connectors: direct access or procedure call
  - control structure: varies, may depend on input or state of computation
- **variants:** traditional data base systems, compilers, blackboard systems

# Layered style



- **problem: distinct, hierarchical classes of services. “Concentric circles” of functionality**
- **context: a large system that requires decomposition (e.g., virtual machines, OSI model)**
- **solution:**
  - system model: hierarchy of layers, often limited visibility
  - components: collections of procedures (module)
  - connectors: (limited) procedure calls
  - control structure: single or multiple threads
- **variants: relaxed layering**

# Model-View-Controller (MVC) style



- **problem:** separation of UI from application is desirable due to expected UI adaptations
- **context:** interactive applications with a flexible UI
- **solution:**
  - system model: UI (View and Controller Component(s)) is decoupled from the application (Model component)
  - components: collections of procedures (module)
  - connectors: procedure calls
  - control structure: single thread
- **variants:** Document-View

# Overview

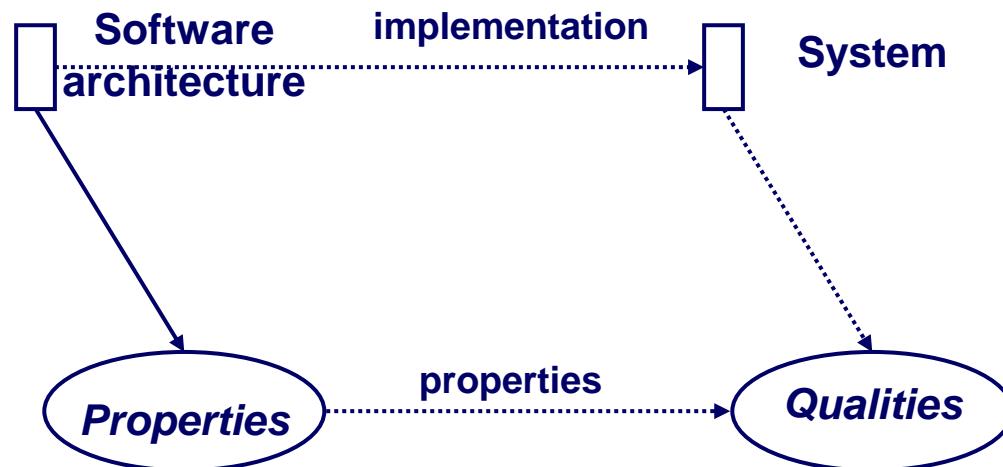
- What is it, why bother?
- Architecture Design
- Viewpoints and view models
- Architectural styles
- **Architecture assessment SKIP**
- Role of the software architect



# Architecture evaluation/analysis

- **Assess whether architecture meets certain quality goals, such as those w.r.t. maintainability, modifiability, reliability, performance**
- **Mind: the architecture is assessed, while we hope the results will hold for a system yet to be built**

# Software Architecture Analysis



# Analysis techniques

- **Questioning techniques:** how does the system react to various situations; often make use of scenarios
- **Measuring techniques:** rely on quantitative measures; architecture metrics, simulation, etc

# Scenarios in Architecture Analysis

- **Different types of scenarios, e.g. use-cases, likely changes, stress situations, risks, far-into-the-future scenarios**
- **Which stakeholders to ask for scenarios?**
- **When do you have enough scenarios?**

# **Preconditions for successful assessment**

- **Clear goals and requirements for the architecture**
- **Controlled scope**
- **Cost-effectiveness**
- **Key personnel availability**
- **Competent evaluation team**
- **Managed expectations**

# Architecture Tradeoff Analysis Method (ATAM)

- **Reveals how well architecture satisfies quality goals, how well quality attributes interact, i.e. how they trade off**
- **Elicits business goals for system and its architecture**
- **Uses those goals and stakeholder participation to focus attention to key portions of the architecture**

# Benefits

- **Financial gains**
- **Forced preparation**
- **Captured rationale**
- **Early detection of problems**
- **Validation of requirements**
- **Improved architecture**

# **Participants in ATAM**

- **Evaluation team**
- **Decision makers**
- **Architecture stakeholders**

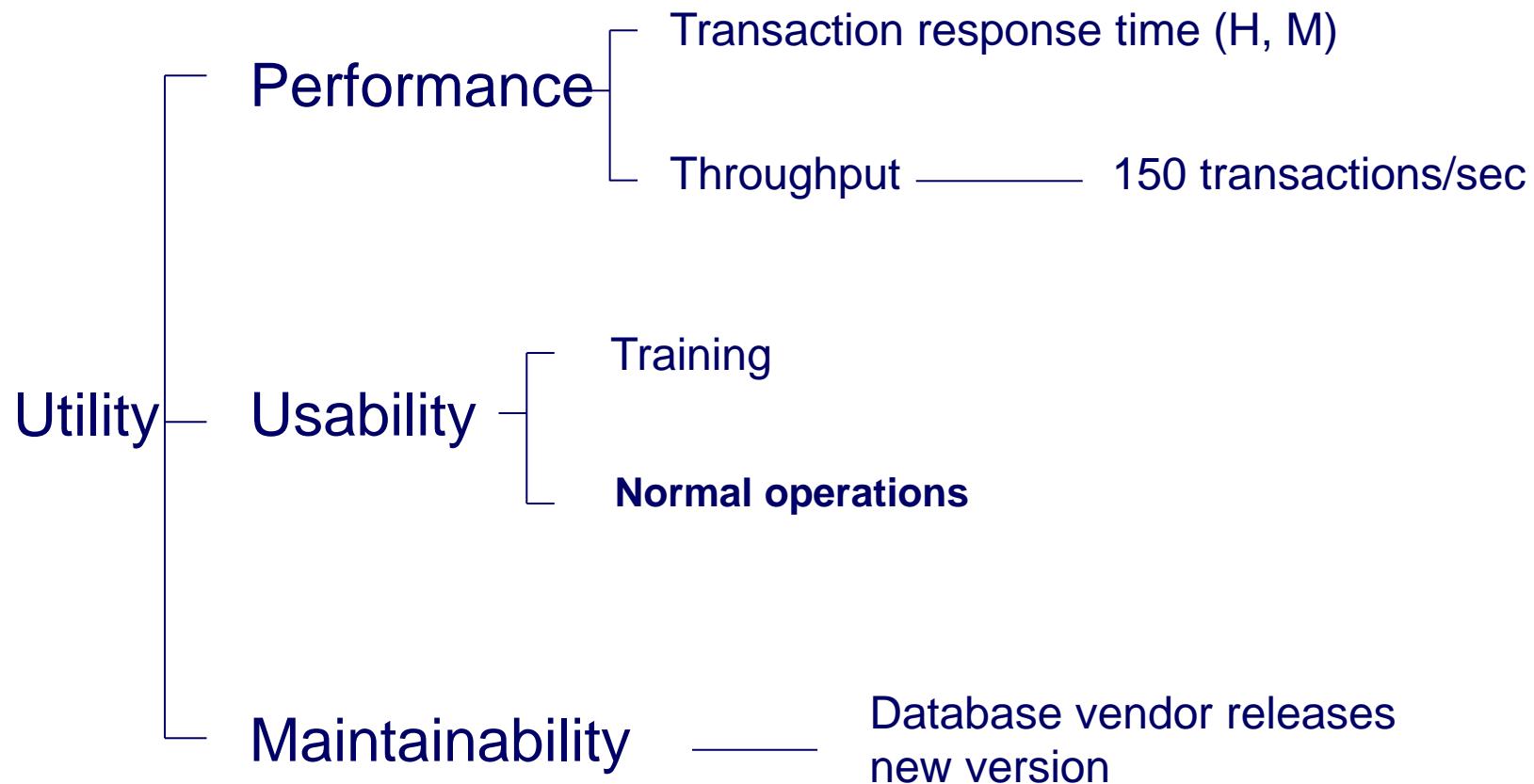
# Phases in ATAM

- 0: partnership, preparation (informally)
- 1: evaluation (evaluation team + decision makers, one day)
- 2: evaluation (evaluation team + decision makers + stakeholders, two days)
- 3: follow up (evaluation team + client)

# **Steps in ATAM (phase 1 and 2)**

- **Present method**
- **Present business drivers (by project manager of system)**
- **Present architecture (by lead architect)**
- **Identify architectural approaches/styles**
- **Generate quality attribute utility tree (+ priority, and how difficult)**
- **Analyze architectural approaches**
  
- **Brainstorm and prioritize scenarios**
- **Analyze architectural approaches**
- **Present results**

# Example Utility tree



# Outputs of ATAM

- **Concise presentation of the architecture**
- **Articulation of business goals**
- **Quality requirements expressed as set of scenarios**
- **Mapping of architectural decisions to quality requirements**
- **Set of sensitivity points and tradeoff points**
- **Set of risks, nonrisks, risk themes**

# Important concepts in ATAM

- **Sensitivity point: decision/property critical for certain quality attribute**
- **Tradeoff point: decision/property that affects more than one quality attribute**
- **Risk: decision/property that is a potential problem**
- **These concepts are overlapping**

# Software Architecture Analysis Method (SAAM)

- **Develop scenarios for**
  - kinds of activities the system must support
  - kinds of changes anticipated
- **Describe architecture(s)**
- **Classify scenarios**
  - direct -- use requires no change
  - indirect -- use requires change
- **Evaluate indirect scenarios: changes and cost**
- **Reveal scenario interaction**
- **Overall evaluation**

# Scenario interaction in SAAM

- **Two (indirect) scenarios interact if they require changes to the same component**
- **Scenario interaction is important for two reasons:**
  - Exposes allocation of functionality to the design
  - Architecture might not be at right level of detail

# Overview



- What is it, why bother?
- Architecture Design
- Viewpoints and view models
- Architectural styles
- Architecture assessment
  
- **Role of the software architect**

# Role of the software architect

- Key technical consultant
- *Make decisions*
- Coach of development team
- Coordinate design
- Implement key parts
- Advocate software architecture

# Summary

- new and immature field
- proliferation of terms: architecture - design pattern - framework - idiom
- architectural styles and design pattern *describe* (how are things done) as well as *prescribe* (how should things be done)
- stakeholder communication
- early evaluation of a design
- transferable abstraction

# Further reading

- **Mary Shaw and David Garlan, Software Architecture; Perspectives of an Emerging Discipline, 1995.**
- **Philippe B. Kruchten, The 4+1 view model of architecture, IEEE Software, 12(6):42-50, November 1995.**
- **Frank Buschmann et al., Pattern-Oriented Software Architecture: A System of Patterns, 1996. Part II: 2001.**
- **Erich Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, 1995.**
- **Len Bass et al, Sofware Architecture in Practice, 2003 (2<sup>nd</sup> edition).**
- **C. Hofmeister et al., Applied Software Architecture, 1999.**
- **Jan Bosch, Design & Use of Software Architectures, 2000.**

# Software Design

Main issues:

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

Ingegneria del Software

AA 21/22

Angelo Gargantini

# Overview

- Introduction
- Design principles
- Design methods
- Conclusion

# Programmer's Approach to Software Engineering



Skip requirements engineering and design phases;  
start writing code

# Point to ponder

Is this the same as eXtreme Programming?

Or is there something additional in XP?

# Why this programmer's 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

# 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 as 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

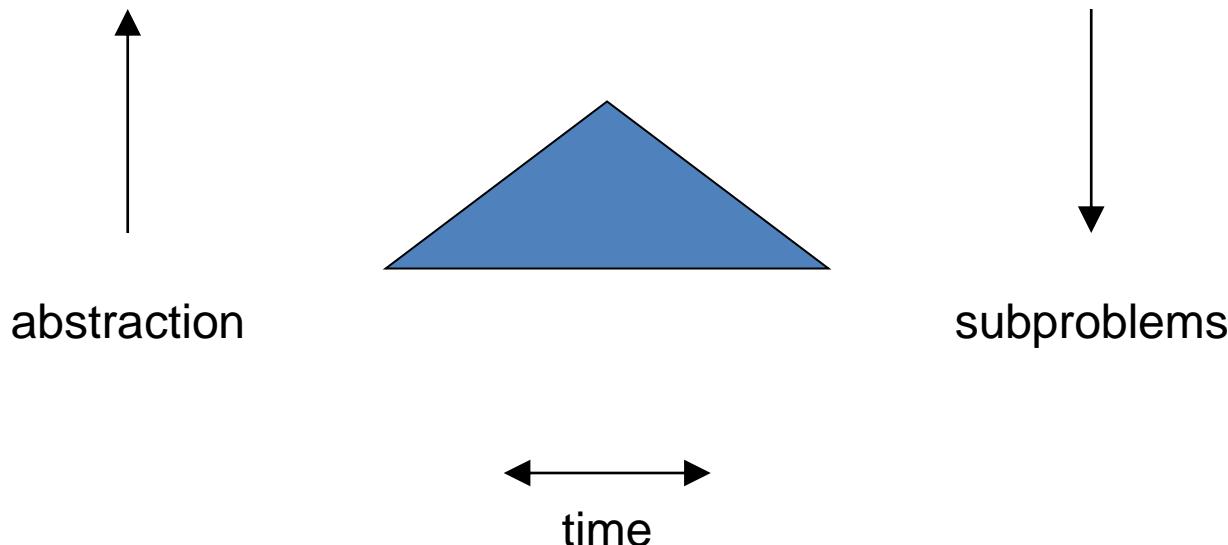
# Design principles



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

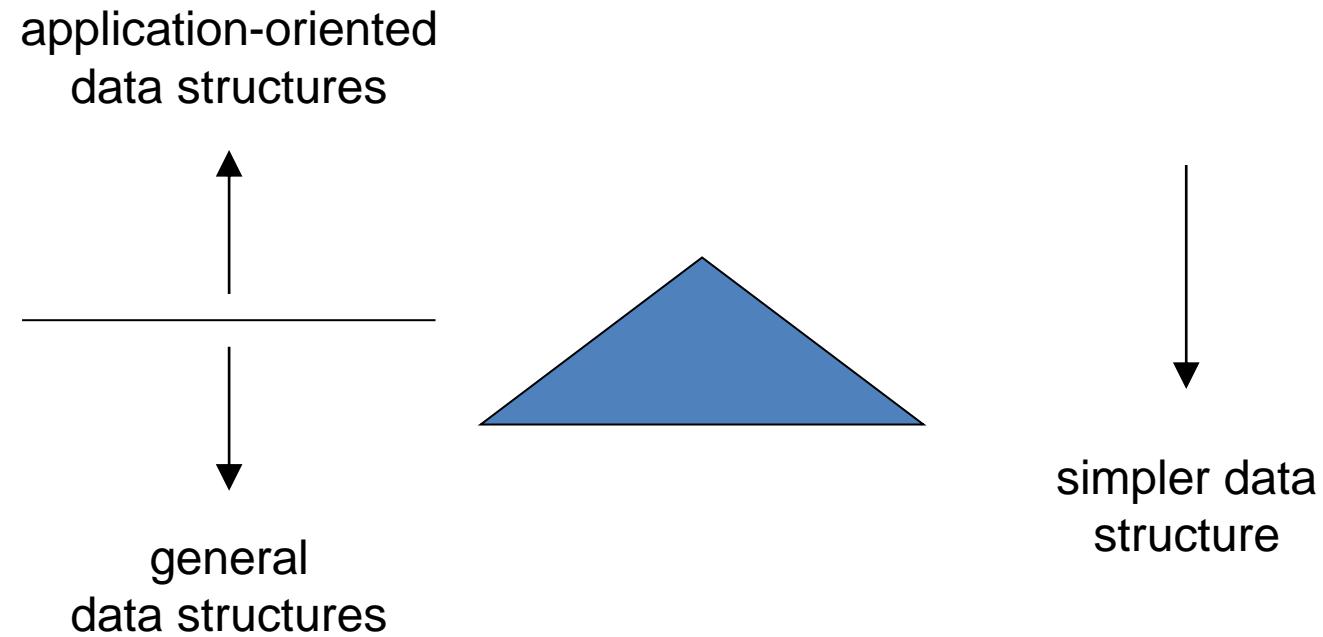
# Abstraction

- procedural abstraction: natural consequence of stepwise refinement: name of procedure denotes sequence of actions



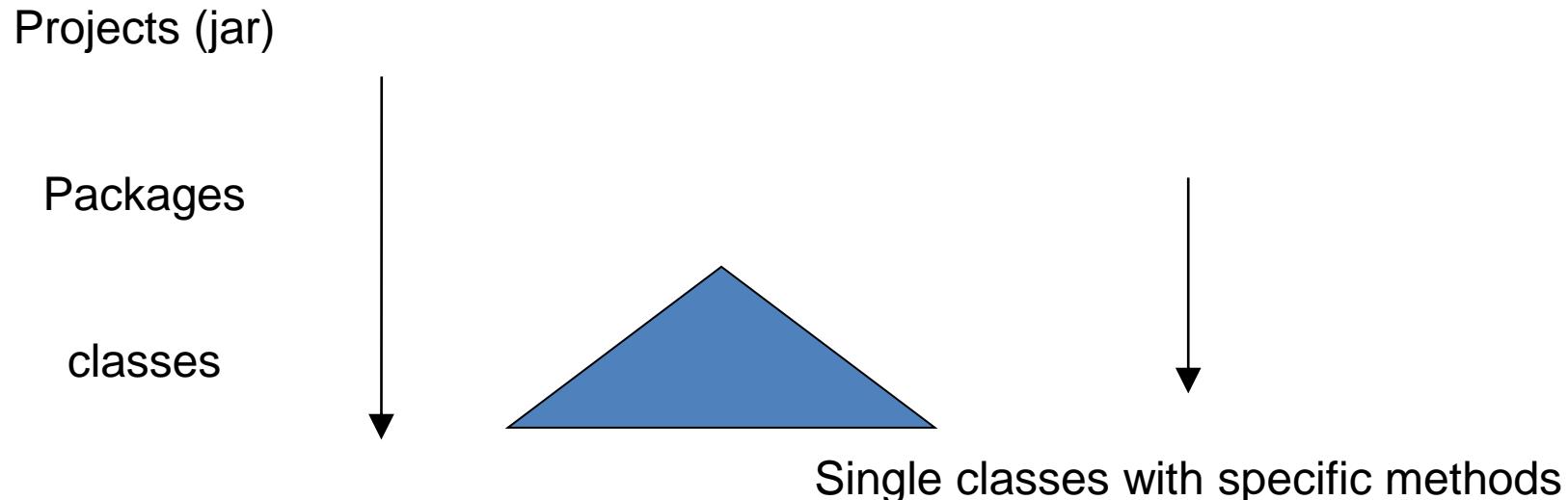
# Abstraction

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

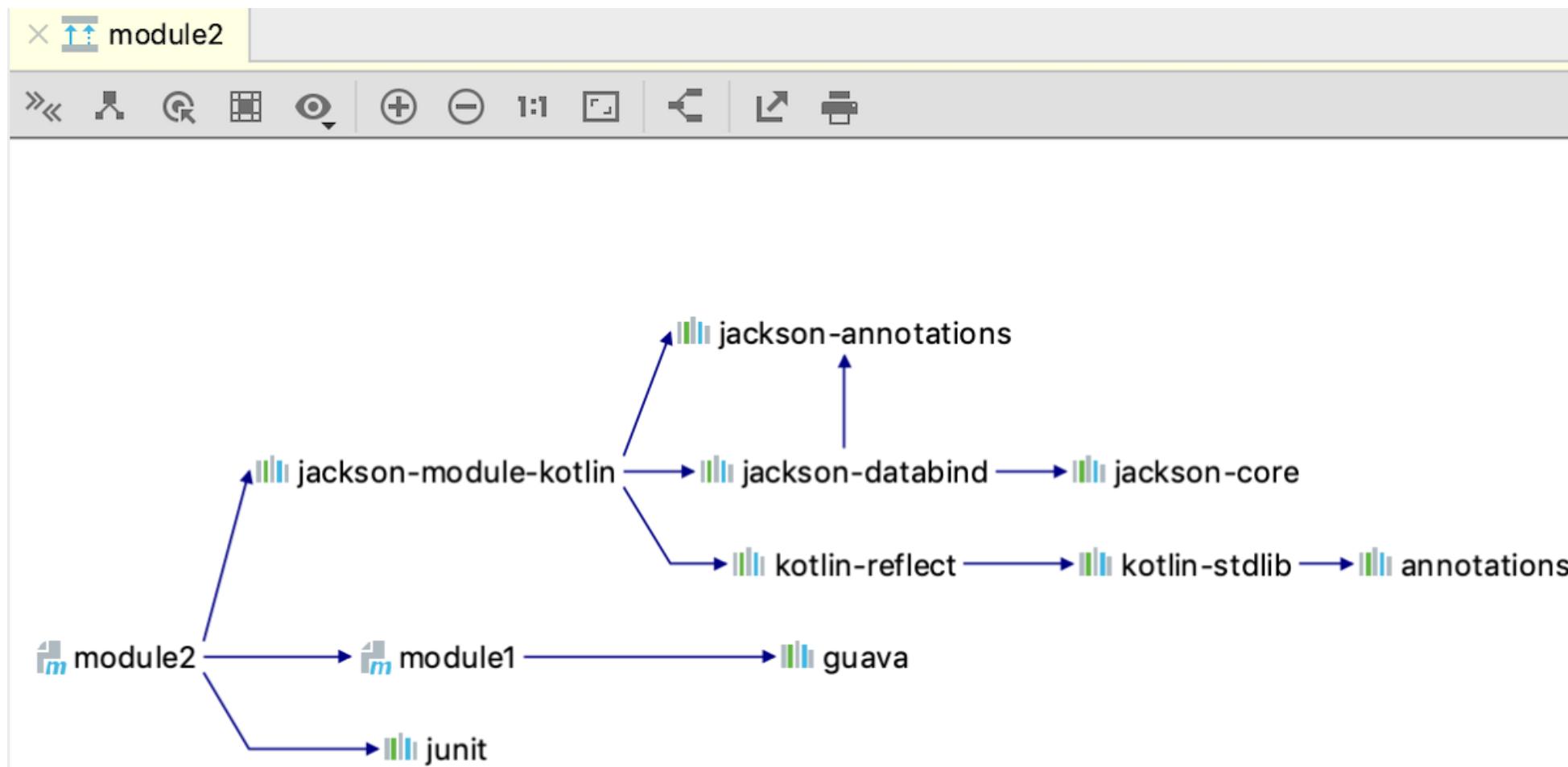


# Abstraction in OO projects Java

- Separate your application in modules (or projects) – packages – classes
  - *Java Platform Module System (JPMS)*



# Maven dependencies



# Modularity

- structural criteria which tell us something about individual modules and their interconnections
- cohesion and coupling
- cohesion: the glue that keeps a module together
- coupling: the strength of the connection between modules

# Coupling and Cohesion

- **Coupling:** Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.
- **Cohesion:** Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.

# Types of cohesion

- coincidental cohesion
- logical cohesion
- temporal cohesion
- procedural cohesion
- communicational cohesion
- sequential cohesion
- functional cohesion
- data cohesion (to cater for abstract data types)



- **Coincidental cohesion:** Elements are grouped into components in a haphazard way. There is no significant relation between the elements.
- **Logical cohesion:** Elements realize tasks that are logically related. One example is a component that contains all input routines. These routines do not call one another and they do not pass information to each other. Their function is just very similar.
- **Temporal cohesion:** The elements are independent but they are activated at about the same point in time. A typical example of this type of cohesion is an initialization component.
- **Procedural cohesion:** The elements have to be executed in a given order. For instance, a component may have to first read some data, then search a table, and finally print a result.
- **Communicational cohesion:** The elements of a component operate on the same (external) data. For instance, a component may read some data from a disk, perform certain computations on the data, and print the result.
- **Sequential cohesion:** The component consists of a sequence of elements where the output of one element serves as input to the next element.
- **Functional cohesion:** All elements contribute to a single function. Such a component often transforms a single input into a single output. The well-known mathematical subroutines are a typical example of this. Less trivial examples are components such as ‘execute the next edit command’ and ‘translate the program given’

# How to determine the cohesion type?

- describe the purpose of the module in one sentence
- if the sentence is compound, contains a comma or more than one verb ⇒ it probably has more than one function: logical or communicational cohesion
- if the sentence contains time-related words like “first”, “then”, “after” ⇒ temporal cohesion
- if the verb is not followed by a specific object ⇒ probably logical cohesion (example: edit all data)
- words like “startup”, “initialize” imply temporal cohesion

# Types of coupling



- content coupling
- common coupling
- external coupling
- control coupling
- stamp coupling
- data coupling

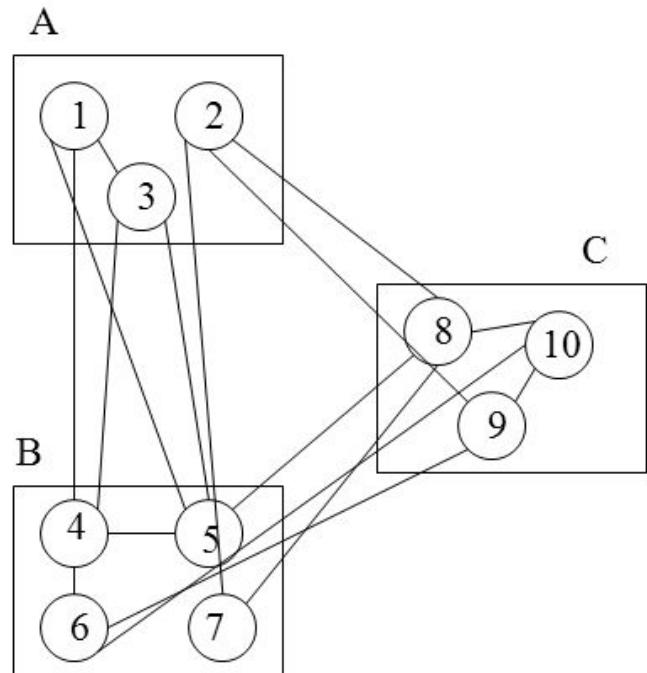
# Coupling levels are technology dependent

- Data coupling assumes scalars or arrays, not 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 (AQ may be data coupled to B, while B is control coupled to A)

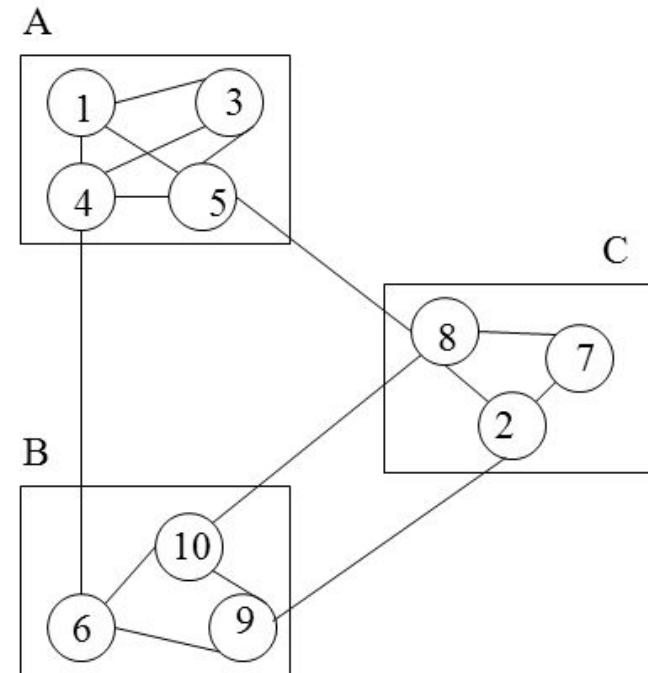
strong cohesion & weak coupling ⇒  
simple interfaces ⇒

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

# Bad and good modularization



Bad modularization:  
low cohesion, high coupling



Good modularization:  
high cohesion, low coupling

# Information hiding



- each module has a secret
- 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

# Example

- <http://www.ucdetector.org/>
- Code where the visibility could be changed to protected, default or private

# Complexity

- In a very general sense, the complexity of a problem refers to the amount of resources required for its solution.
  - We may try to determine complexity in this way by measuring, say, the time needed to solve a problem. This is called an *external* attribute: we are not looking at the entity itself (the problem), but at how it behaves.
- In the present context, complexity refers to attributes of the software that affect the effort needed to construct or change a piece of software.
  - These are *internal* attributes: they can be measured purely in terms of the software itself. For example, we need not execute the software to determine their values.

# Complexity

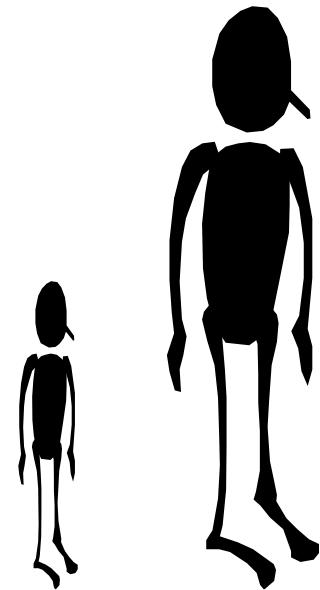
- measure certain aspects of the software (lines of code, # of if-statements, depth of nesting, ...)
- use these numbers as a criterion to assess a design, or to guide the design
- interpretation: higher value  $\Rightarrow$  higher complexity  $\Rightarrow$  more effort required (= worse design)
- two 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



# Software science basic entities



- $n_1$ : number of unique operators
  - The set of operators includes the arithmetic and Boolean operators, as well as separators (such as a semicolon between adjacent instructions) and (pairs of) reserved words.
- $n_2$ : number of unique operands
  - The set of operands contains the variables and constants used.
- $N_1$ : 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

# # of Occurrences

public	1
sort()	1
int	4
[ ]	7
{ }	4
for { ; ; }	2
if ()	1
=	5
<	2
...	...
$n_1 = 17$	$N_1 = 39$

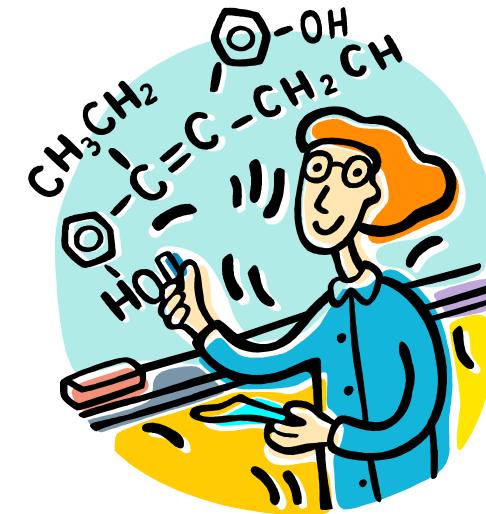
# operand

# # 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 wrongly
  - is aimed at coding phase only; assumes this is an uninterrupted concentrated activity
  - different definitions of “operand” and “operator”

# 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

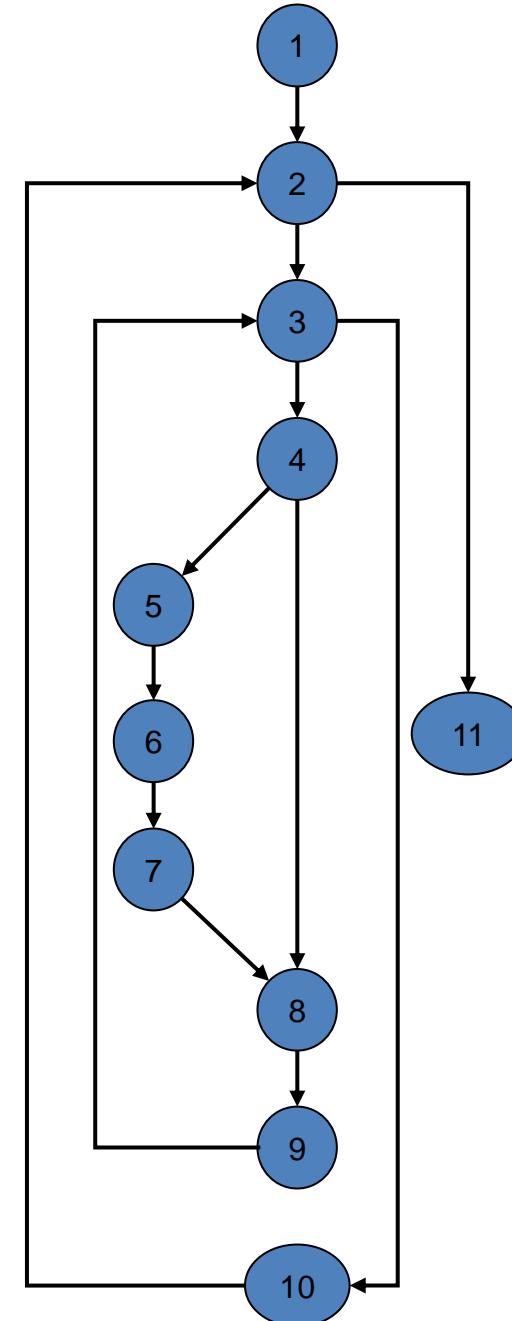


# McCabe's cyclomatic complexity

- The cyclomatic complexity of a section of source code is the number of linearly independent paths within it
- The complexity  $M$  is then defined as
$$M = E - N + 2P,$$
  - $E$  = the number of edges of the graph.
  - $N$  = the number of nodes of the graph.
  - $P$  = the number of connected components.
- We will use
  - $M = E - N + P + 1$  (count each subgraph as separated)

# 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  
            }  
        }  
    }  
}
```



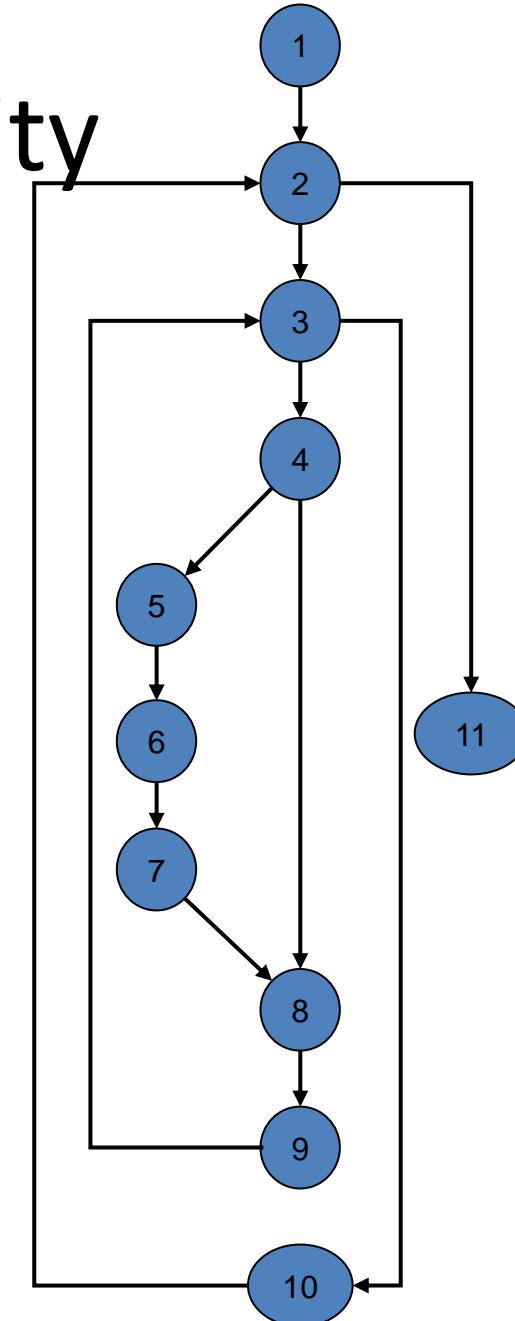
# Cyclomatic complexity

e = number of edges (13)

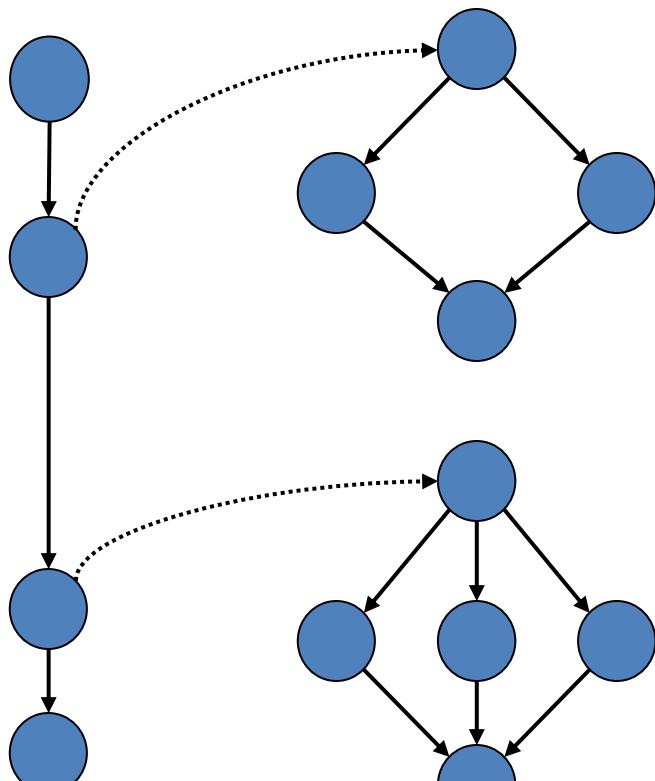
n = number of nodes (11)

p = number of connected components (1)

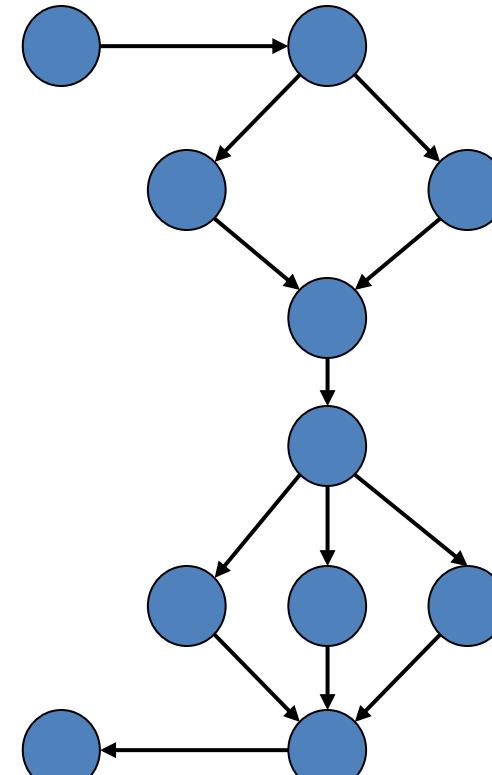
$$CV = e - n + p + 1 (4)$$



Note:  $CV = e-n+p+1$ ,  $CV \neq e-n+2p$



$$e-n+p+1 = 13-13+3+1 = 4$$
$$e-n+2p = 6$$



$$e-n+p+1 = e-n+2p = 4$$

# Intra-modular complexity measures, summary

- 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

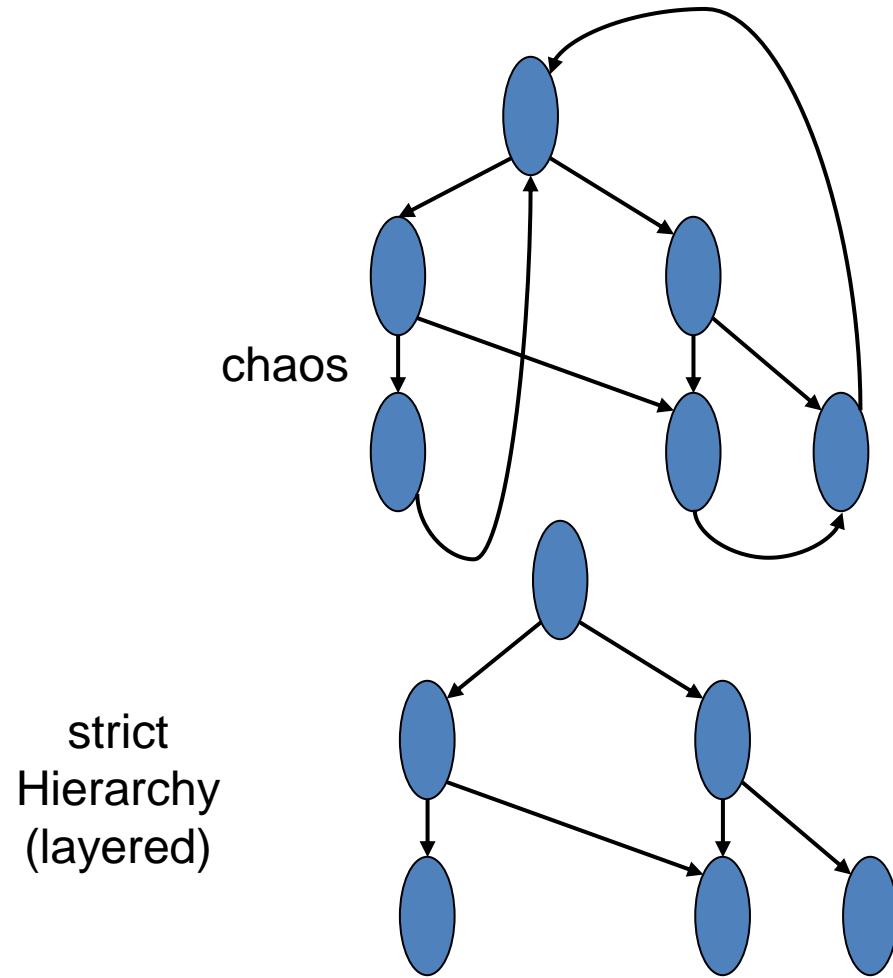
# System structure: inter-module complexity

- 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

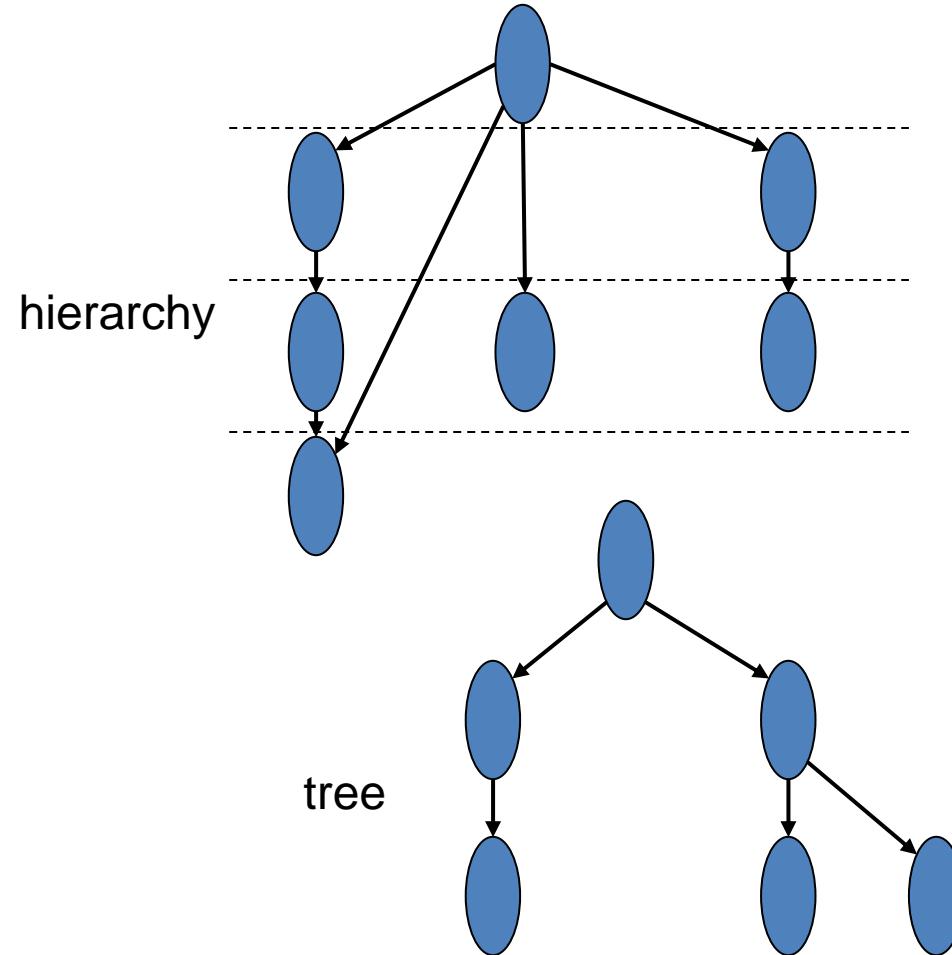
- In a well-structured piece of software, the dependencies show up as procedure calls
- therefore, this graph is known as the *call-graph*
- possible shapes of this graph:
  - chaos (directed graph)
  - hierarchy (acyclic graph)
  - strict hierarchy (layers)
  - tree

# In a picture:

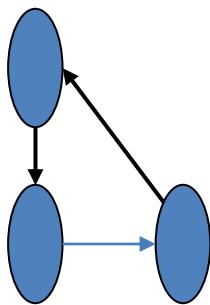


©2008 John Wiley & Sons Ltd.

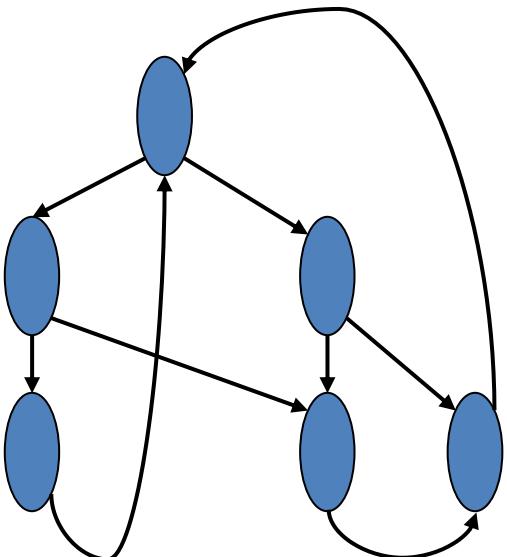
[www.wileyeurope.com/college/van vliet](http://www.wileyeurope.com/college/van vliet)



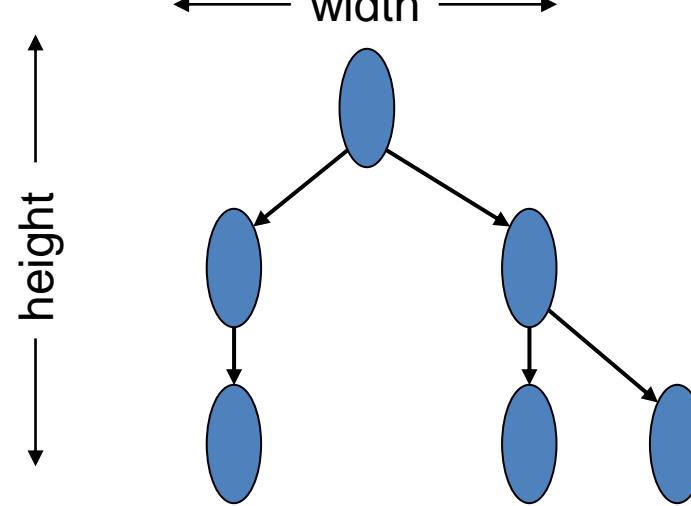
# Problems with cycles



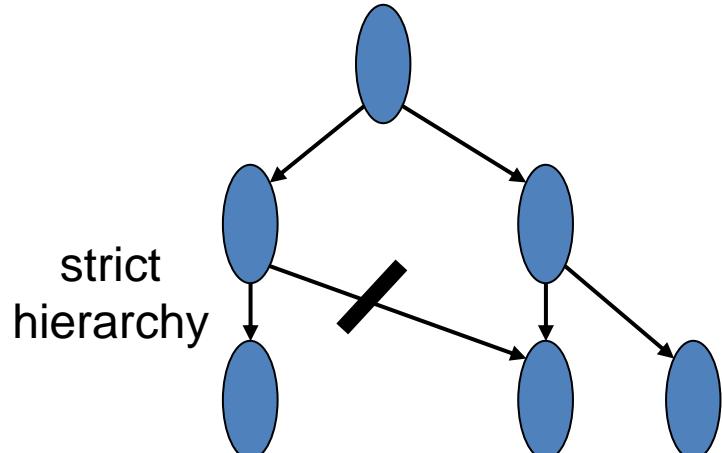
# Measurements



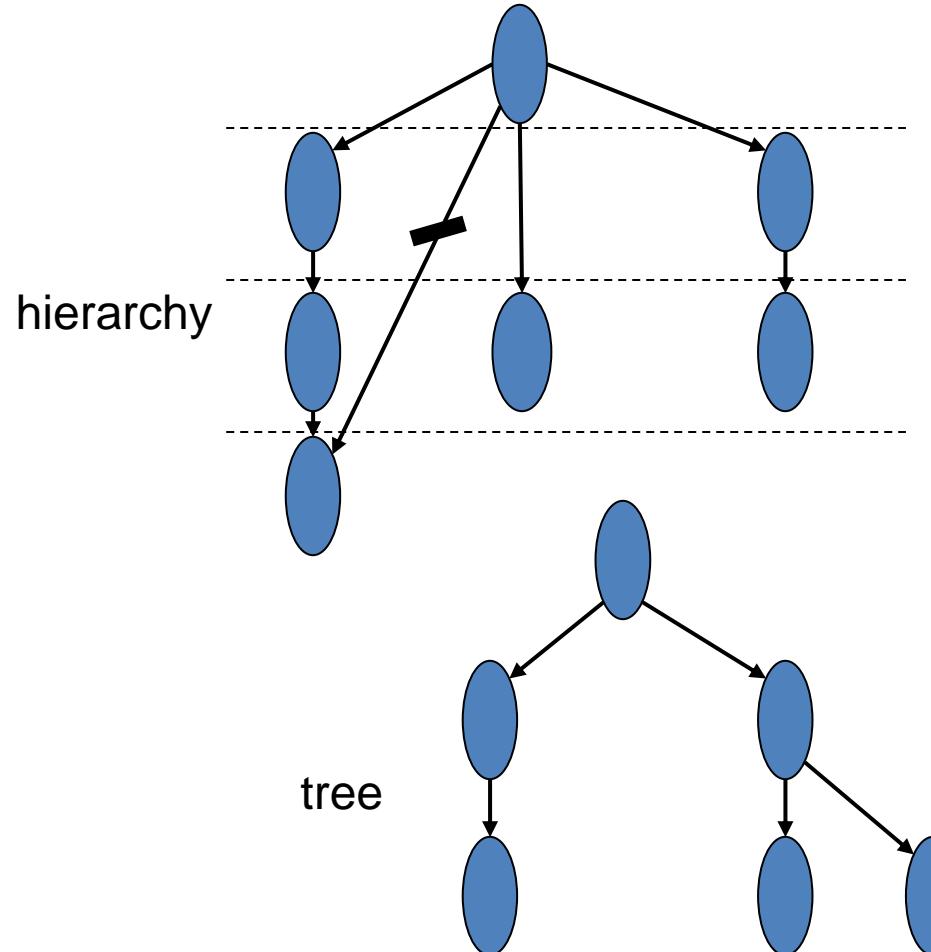
} size  
# nodes  
# edges



# Deviation from a tree



strict  
hierarchy



tree

# Tree impurity metric

- complete graph with  $n$  nodes has  $n(n-1)/2$  edges
- a tree with  $n$  nodes has  $(n-1)$  edges
- tree impurity for a graph with  $n$  nodes and  $e$  edges:  
$$m(G) = 2(e-n+1)/(n-1)(n-2)$$
- this is a “good” measure, in the measurement theory sense

# Desirable properties of any tree impurity metric

- $m(G) = 0$  if and only if  $G$  is a tree
- $m(G_1) > m(G_2)$  if  $G_1 = G_2 + \text{an extra edge}$
- 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 SKIP

- 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 variant of information flow metric

- there is a *local* flow from A to B if:
  - A invokes B and passes it a parameter
  - B invokes A and A returns a value
- there is a *global* flow from A to B if A updates some global structure and B reads that structure
- $\text{fan-in}(M) = \# \text{ (local and global) flows whose sink is } M$
- $\text{fan-out}(M) = \# \text{ (local and global) flows whose source is } M$
- $\text{complexity}(M) = (\text{fan-in}(M) * \text{fan-out}(M))^2$
- still, all flows count the same

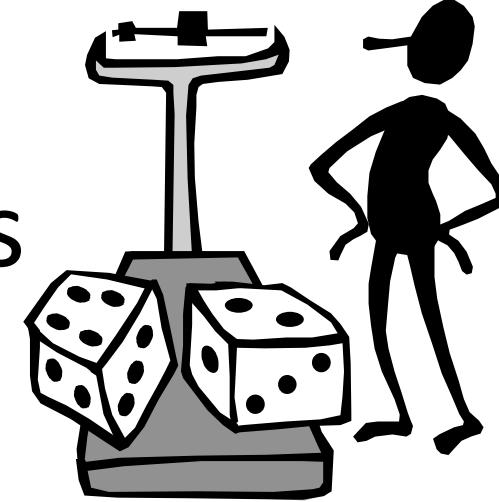
# Point to ponder: What does this program do?

```
procedure X(A: array [1..n] of int);
var i, k, small: int;
begin
    for i:= 1 to n do
        small:= A[i];
        for k:= i to n-1 do
            if small <= A[k]
                then swap (A[k], A[k+1])
            end
        end
    end
end
```

# The role of previously acquired knowledge during design

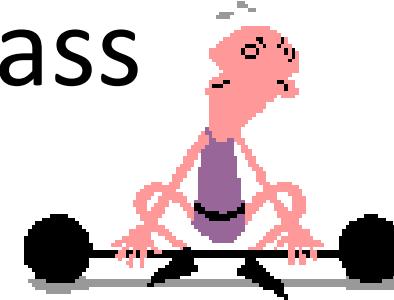
- programming plans, beacons
- chunks
- adverse influence of delocalized plans and false beacons

# 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
- $\text{WMC} = \sum c(i), i = 1, \dots, n$  (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

# Coupling Between Object Classes

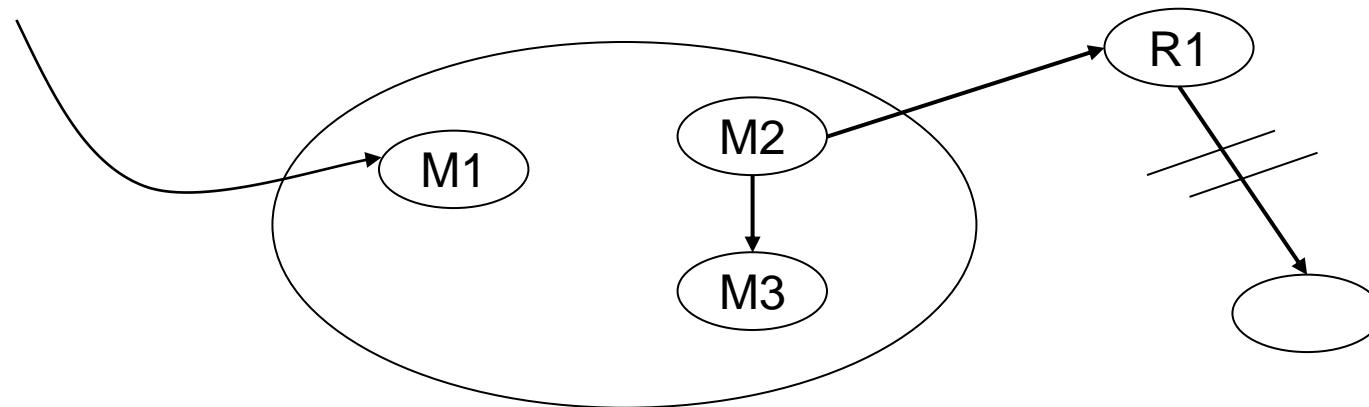
- two classes are coupled if a method of one class uses a method or state variable of another class
- CBO = count of all classes a given class is coupled with
- high values: something is wrong
- all couplings are counted alike; refinements are possible

# Package coupling

- The **afferent coupling** ( $C_a$ ) of a package  $P$  is the number of other packages that depend upon classes within  $P$  (through inheritance or associations). It indicates the dependence of a package on its environment.
- The **efferent coupling** ( $C_e$ ) of a package  $P$  is the number of packages that classes within  $P$  depend upon. It indicates the dependence of the environment on a package.
- Adding these numbers together results in a total coupling measure of a package  $P$ .
- The ratio  $I = C_e/(C_e + C_a)$  indicates the relative dependence of the environment to  $P$  with respect to the total number of dependencies between  $P$  and its environment (**instability**)
  - If  $C_e$  equals zero,  $P$  does not depend at all on other packages and  $I = 0$  as well.
  - If on the other hand  $C_a$  equals zero,  $P$  only depends on other packages and no other package depends on  $P$ . In that case,  $I = 1$ .
  - $I$  thus can be seen as an instability measure for  $P$ . Larger values of  $I$  denote a larger instability of the package.

# Response For a Class

- RFC measures the “immediate surroundings” of a class
- RFC = size of the “response set”
- response set =  $\{M\} \cup \{R_i\}$



# Lack of Cohesion of a Method

- cohesion = glue that keeps the module (class) together
- if all methods use the same set of state variables: OK, & that is the glue
- if 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

# Some tools

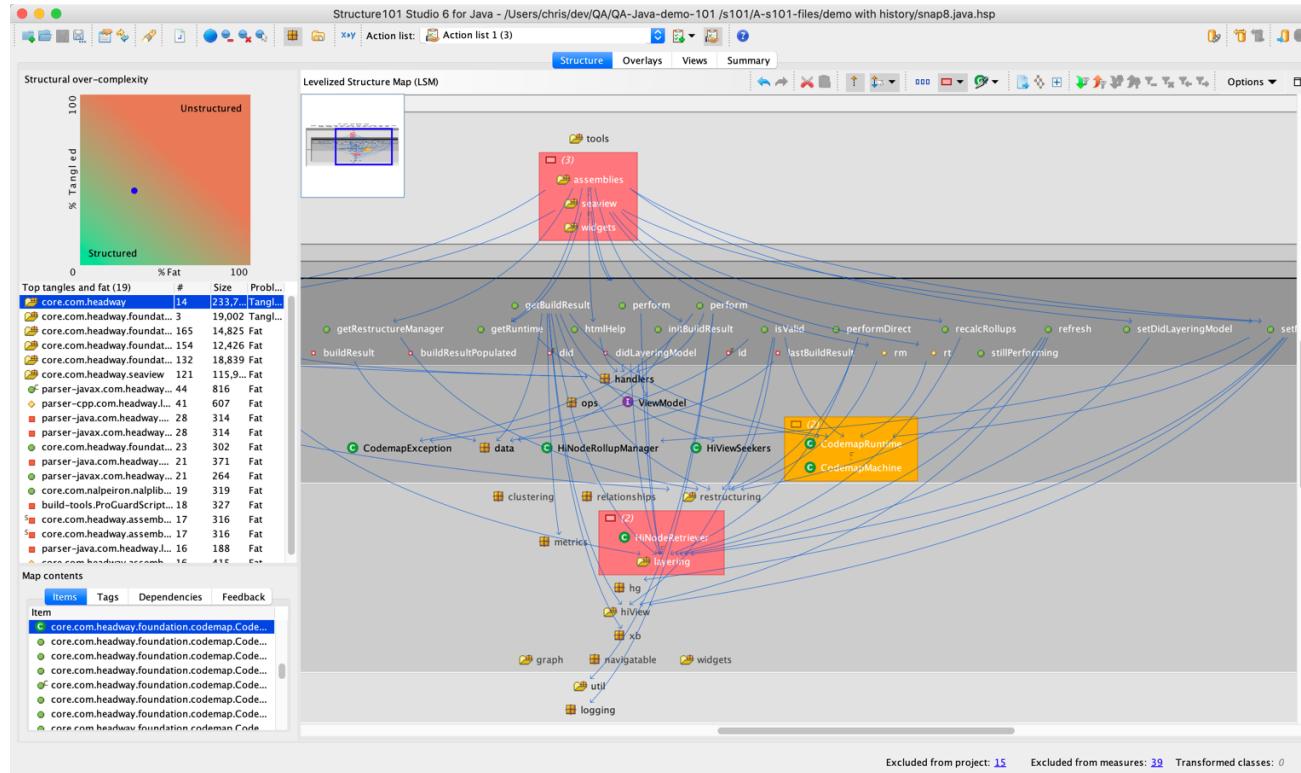
- **JDepend** traverses Java class and source file directories and generates design quality metrics for each Java package. JDepend allows you to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to effectively manage and control package dependencies.
  - <https://github.com/clarkware/jdepend>
- **STAN** encourages developers and project managers in visualizing their design, understanding code, measuring quality and reporting design flaws. **STAN** supports a set of carefully selected metrics, suitable to cover the most important aspects of structural quality. Special focus has been set on visual dependency analysis, a key to structure analysis. **STAN** seamlessly integrates into the development process. That way, developers take care about design quality right from the start. Project managers use **STAN** as a monitoring and reporting tool.
  - <http://stan4j.com/>
- Structure 101

# Alcune metriche di jdepend

Number of Classes	The number of concrete and abstract classes (and interfaces) in the package is an indicator of the extensibility of the package.
Afferent Couplings	The number of other packages that depend upon classes within the package is an indicator of the package's responsibility.
Efferent Couplings	The number of other packages that the classes in the package depend upon is an indicator of the package's independence.
Abstractness	The ratio of the number of abstract classes (and interfaces) in the analyzed package to the total number of classes in the analyzed package. The range for this metric is 0 to 1, with A=0 indicating a completely concrete package and A=1 indicating a completely abstract package.
Instability	The ratio of efferent coupling ( $C_e$ ) to total coupling ( $C_e / (C_e + C_a)$ ). This metric is an indicator of the package's resilience to change. The range for this metric is 0 to 1, with I=0 indicating a completely stable package and I=1 indicating a completely unstable package.
Distance	The perpendicular distance of a package from the idealized line $A + I = 1$ . This metric is an indicator of the package's balance between abstractness and stability. A package squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal packages are either completely abstract and stable ( $x=0, y=1$ ) or completely concrete and unstable ( $x=1, y=0$ ). The range for this metric is 0 to 1, with D=0 indicating a package that is coincident with the main sequence and D=1 indicating a package that is as far from the main sequence as possible.
Cycles	Packages participating in a package dependency cycle are in a deadly embrace with respect to reusability and their release cycle. Package dependency cycles can be easily identified by reviewing the textual reports of dependency cycles. Once these dependency cycles have been identified with JDepend, they can be broken by employing various object-oriented techniques.

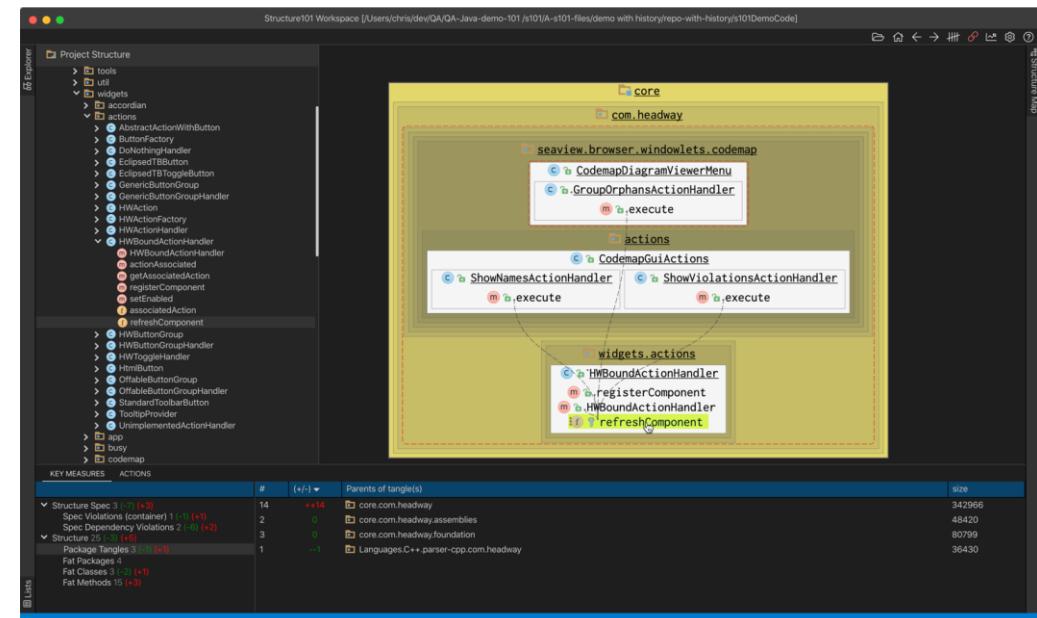
# Structure 101 - Studio

- Studio is a desktop application that lets you work with a visual, interactive model of your codebase. It is the workhorse of Structure101. With Studio you can:
  - Set up and fine-tune your project settings, such as:
  - Where are the primary sources of information about your code-base?
- Understand the structure of your code-base
- Identify any regions of structural over-complexity (like cyclic dependencies)
- Simulate changes (like moving files to different folders) to change or improve the code-base structure (architectural improvements)



# Structure 101 -Workspace

- View the detailed code-level dependencies on whatever code items (like files or functions) you are working on, in the context of the overall architecture (like projects or folders)
- See lists of structural problems, and navigate to them in the visualization, to discover the dependencies that cause the problem
- See list of violations of the architectural specs defined for the code-base, and navigate to them in the visualization
- Navigate from any items in the visualization to the item in your source editor (using a connector plugin)
- Navigate from any item in your source editor to the visualization in Workspace that shows all the item's dependencies.



# Overview

- Introduction
- Design principles
- Design methods
- Conclusion

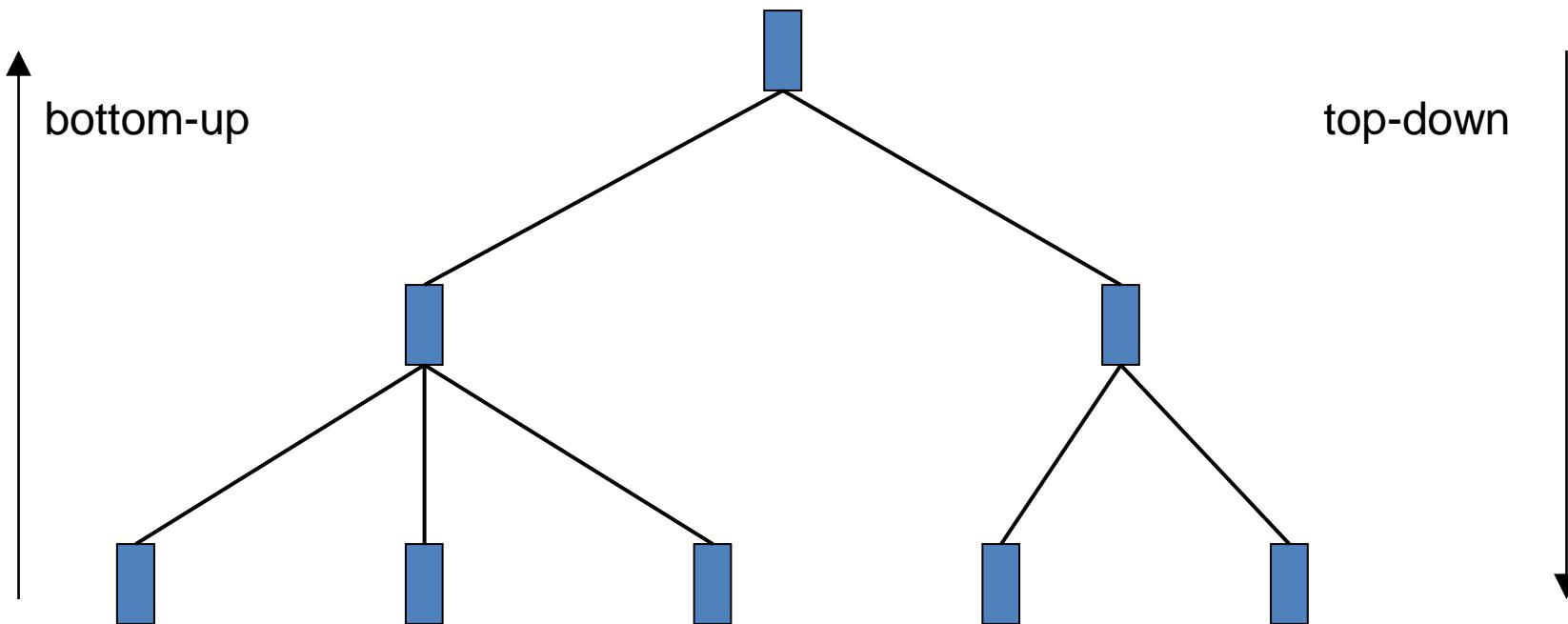
# Design methods

- Functional decomposition
- Data Flow Design (SA/SD)
- Design based on Data Structures (JSD/JSP)
- Object-oriented analysis and design methods

# Sample of 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



# Functional decomposition (cnt'd)

- 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**

# Entities in a data flow diagram

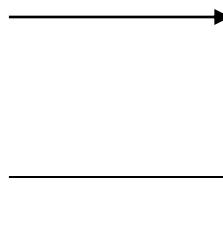
- external entities



- processes

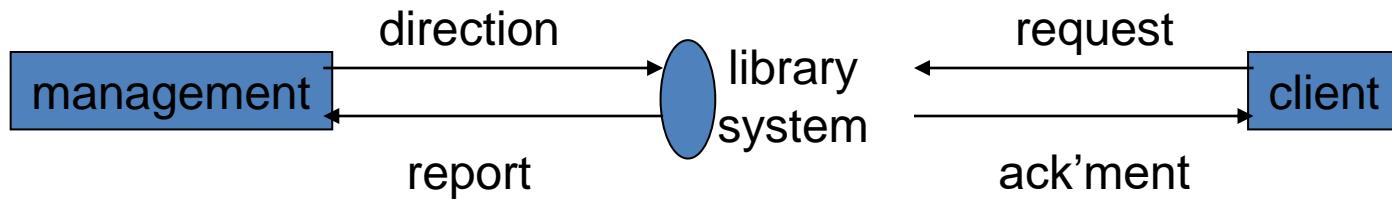


- data flows

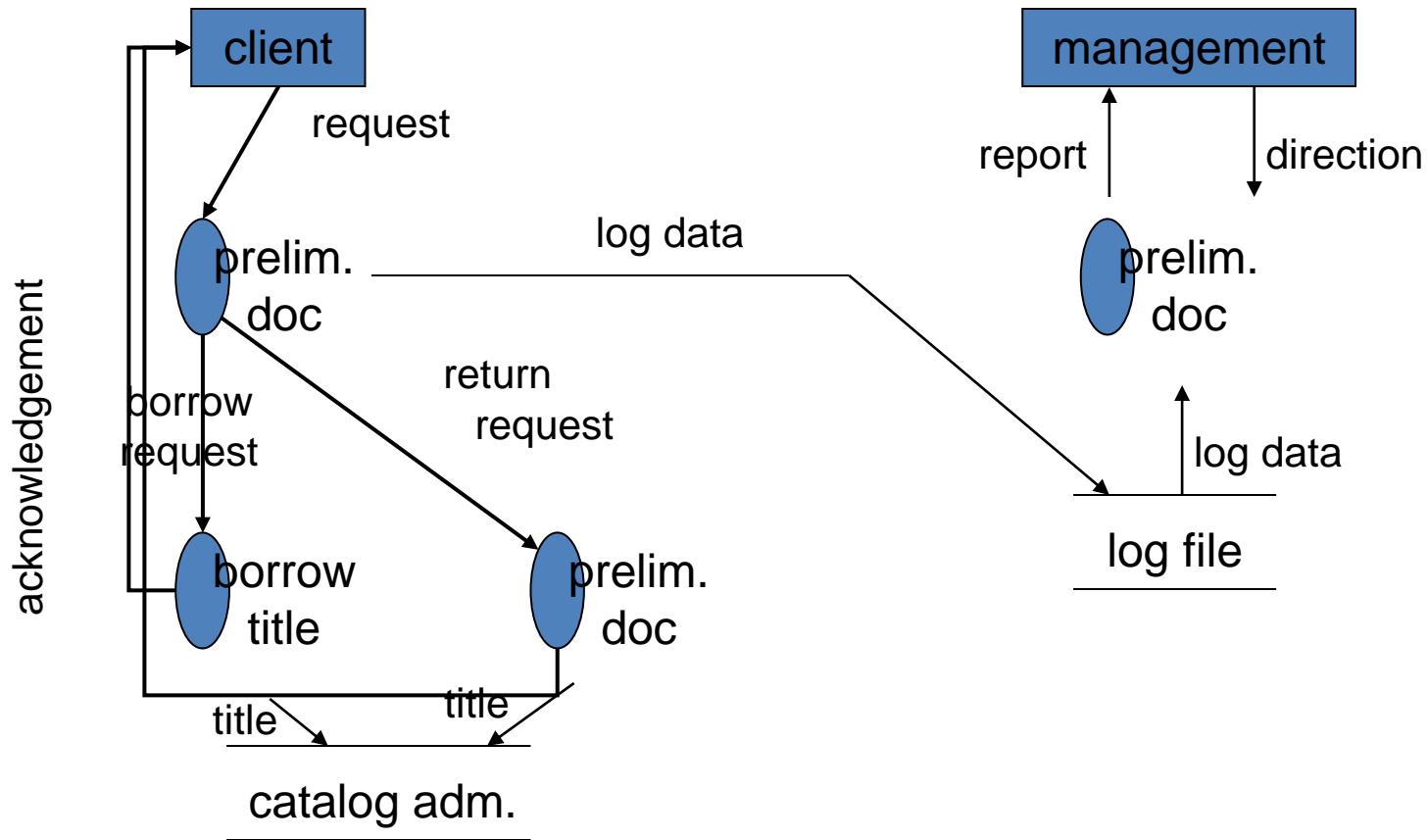


- data stores

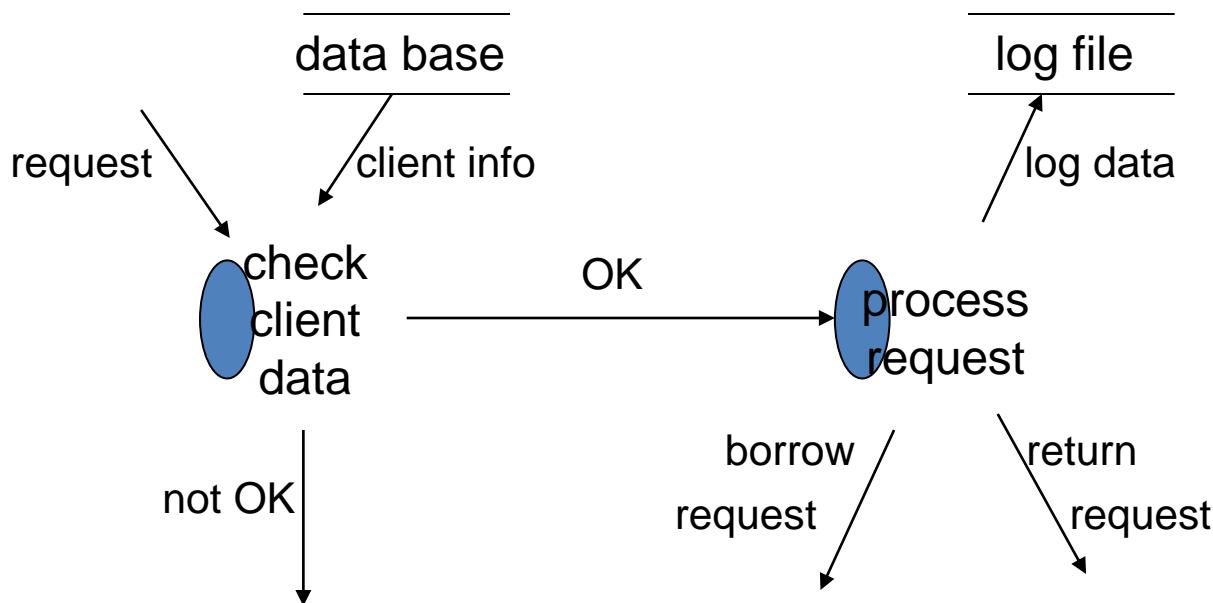
# Top-level DFD: context diagram



# First-level decomposition



# Second-level decomposition for “preliminary processing”



# 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

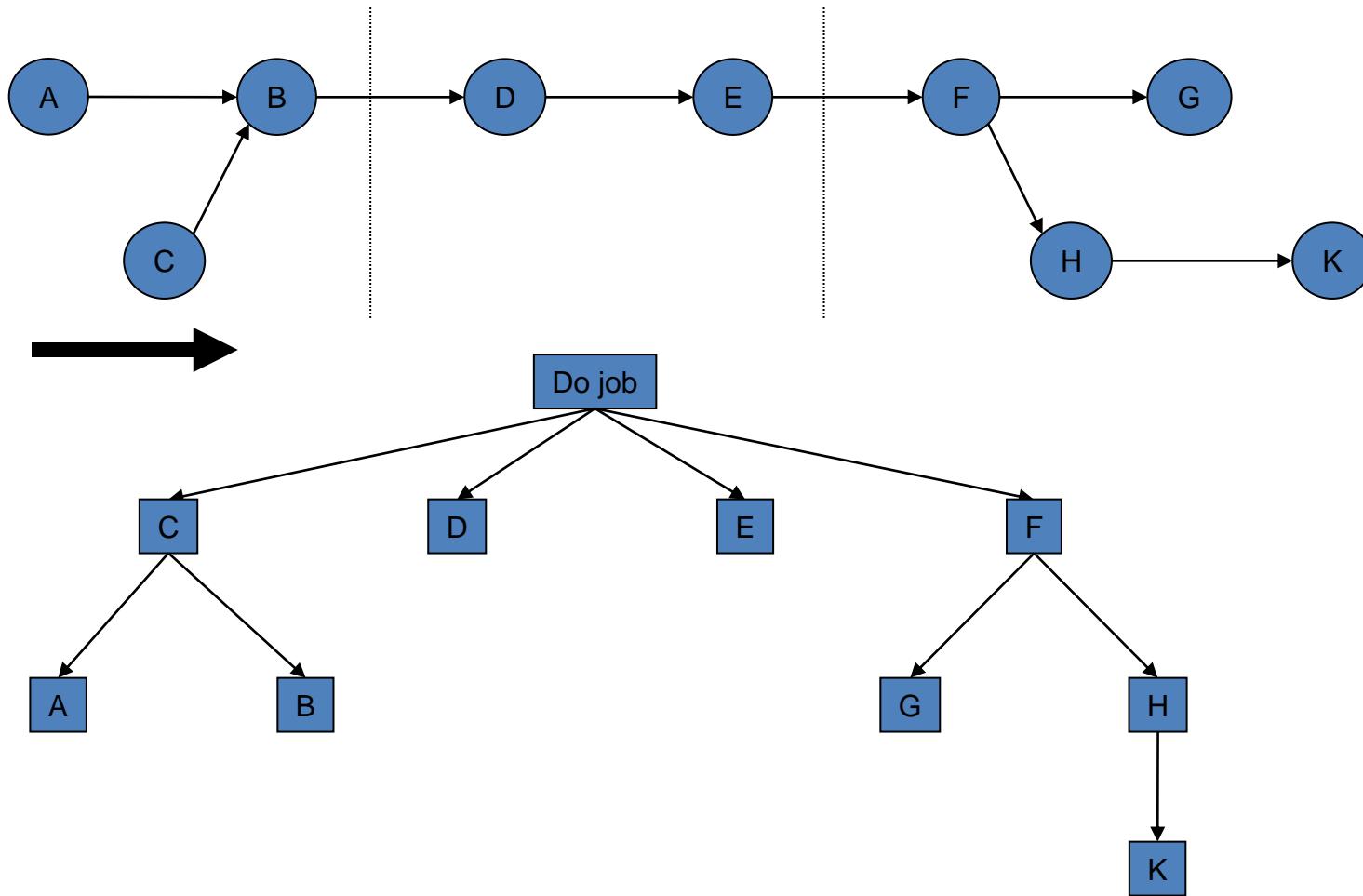
+: AND

( ): 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 & JSD)

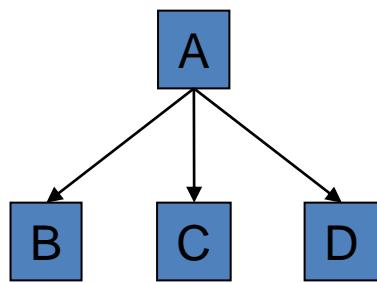
- JSP = Jackson Structured Programming (for programming-in-the-small)
- JSD = Jackson Structured Design (for programming-in-the-large)

# JSP

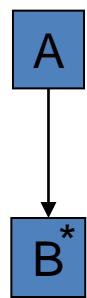
- basic idea: good program reflects structure of its input and output
- program can be derived almost mechanically from a description of the input and output
- input and output are depicted in a *structure diagram* and/or in *structured text/schematic logic* (a kind of pseudocode)
- three basic compound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

# Compound components in JSP

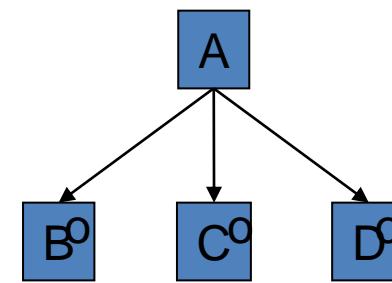
sequence



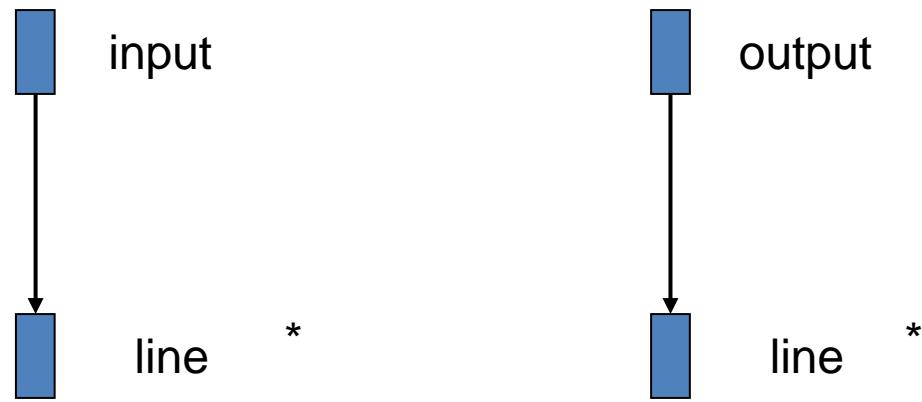
iteration



selection

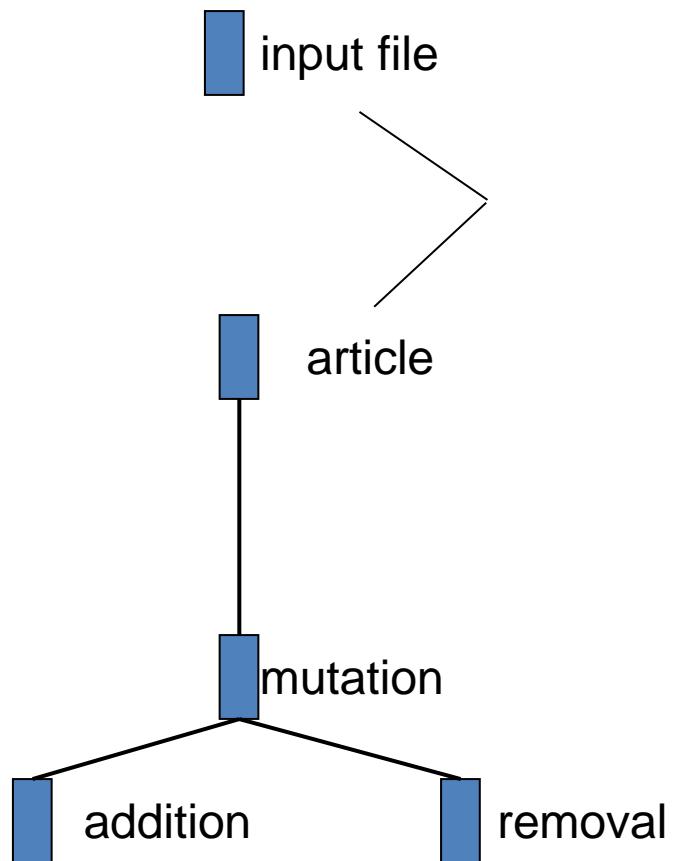


# A JSP example

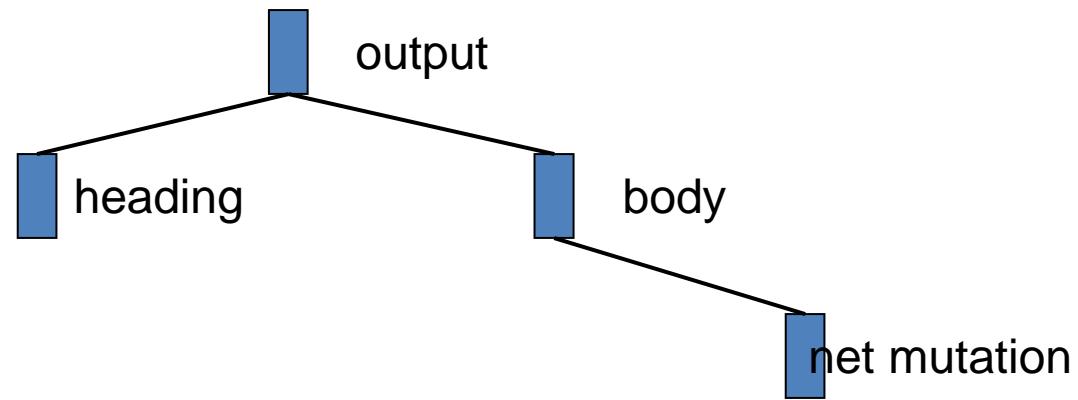


```
until EOF
loop
  read line
  process line
  write line
endloop
```

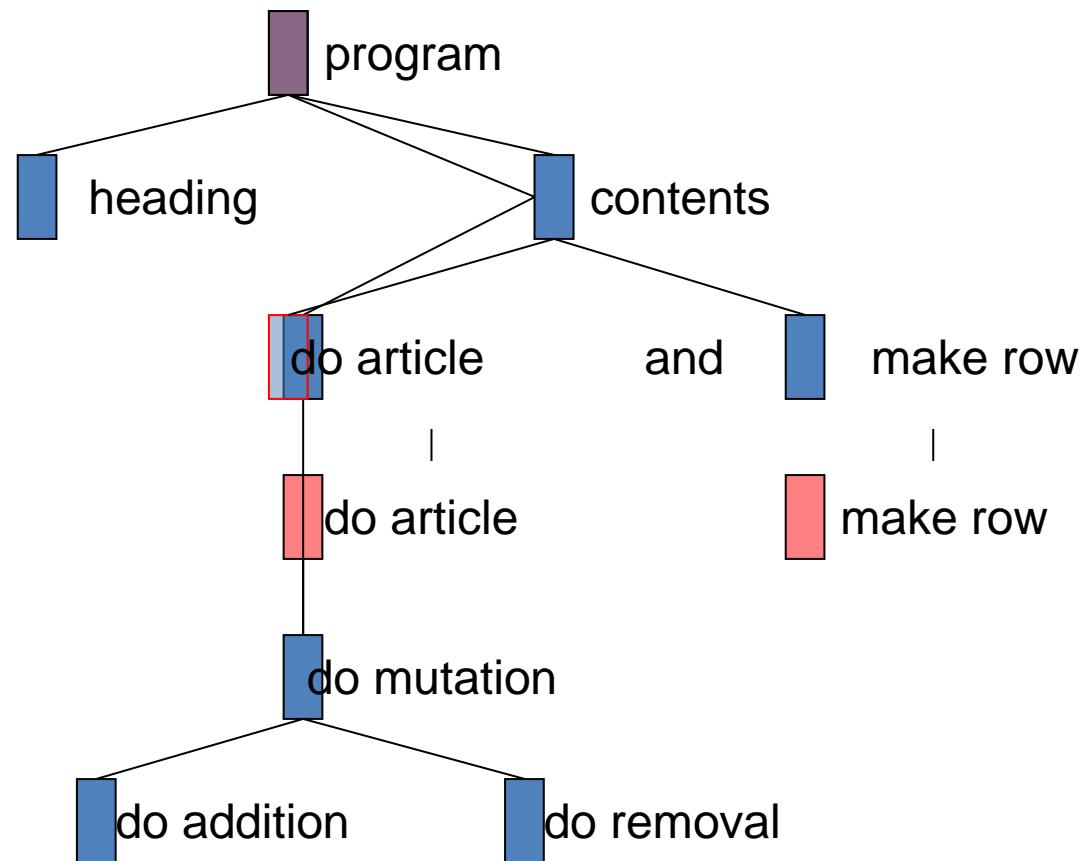
# Another JSP example



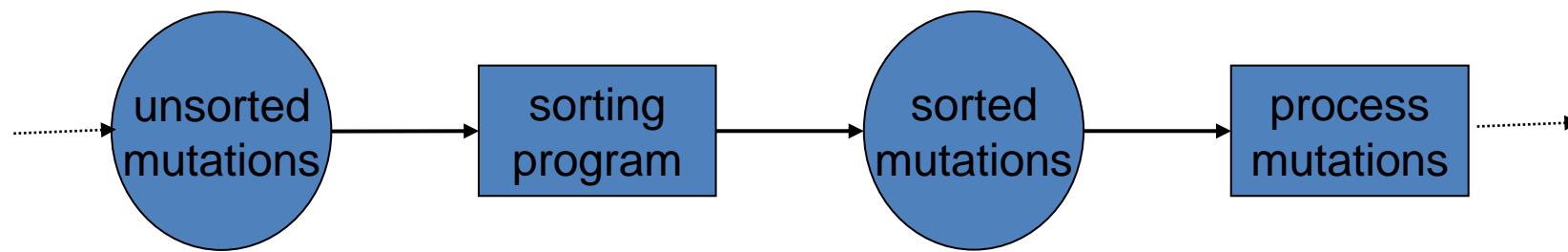
# Another JSP example (cnt'd)



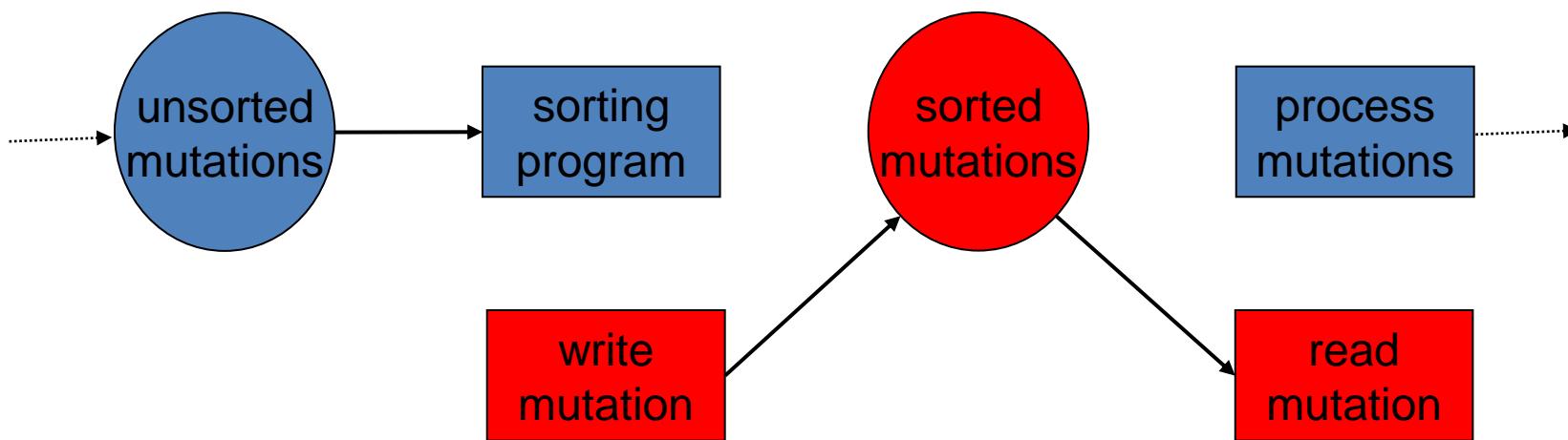
# Another JSP example (cnt'd)



# Structure clash



# Inversion



# 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

# JSD: Jackson Structured Design

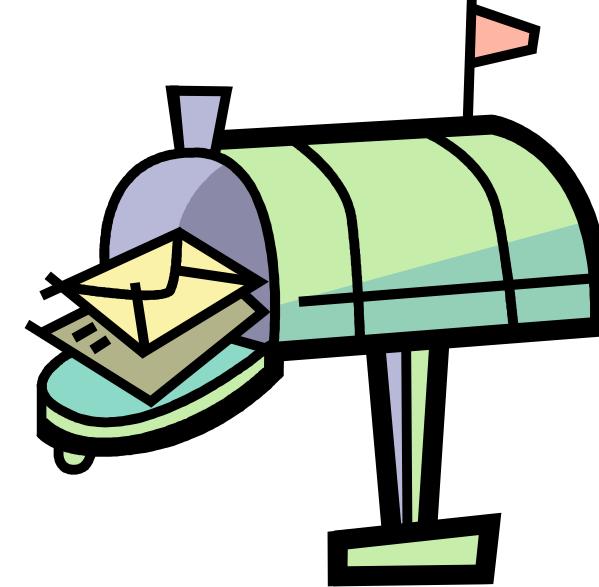
- Problem with JSP: how to obtain a mapping from the problem structure to the data structure?
- JSD tries to fill this gap
- JSD has three stages:
  - **modeling stage**: description of real world problem in terms of entities and actions
  - **network stage**: model system as a network of communicating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



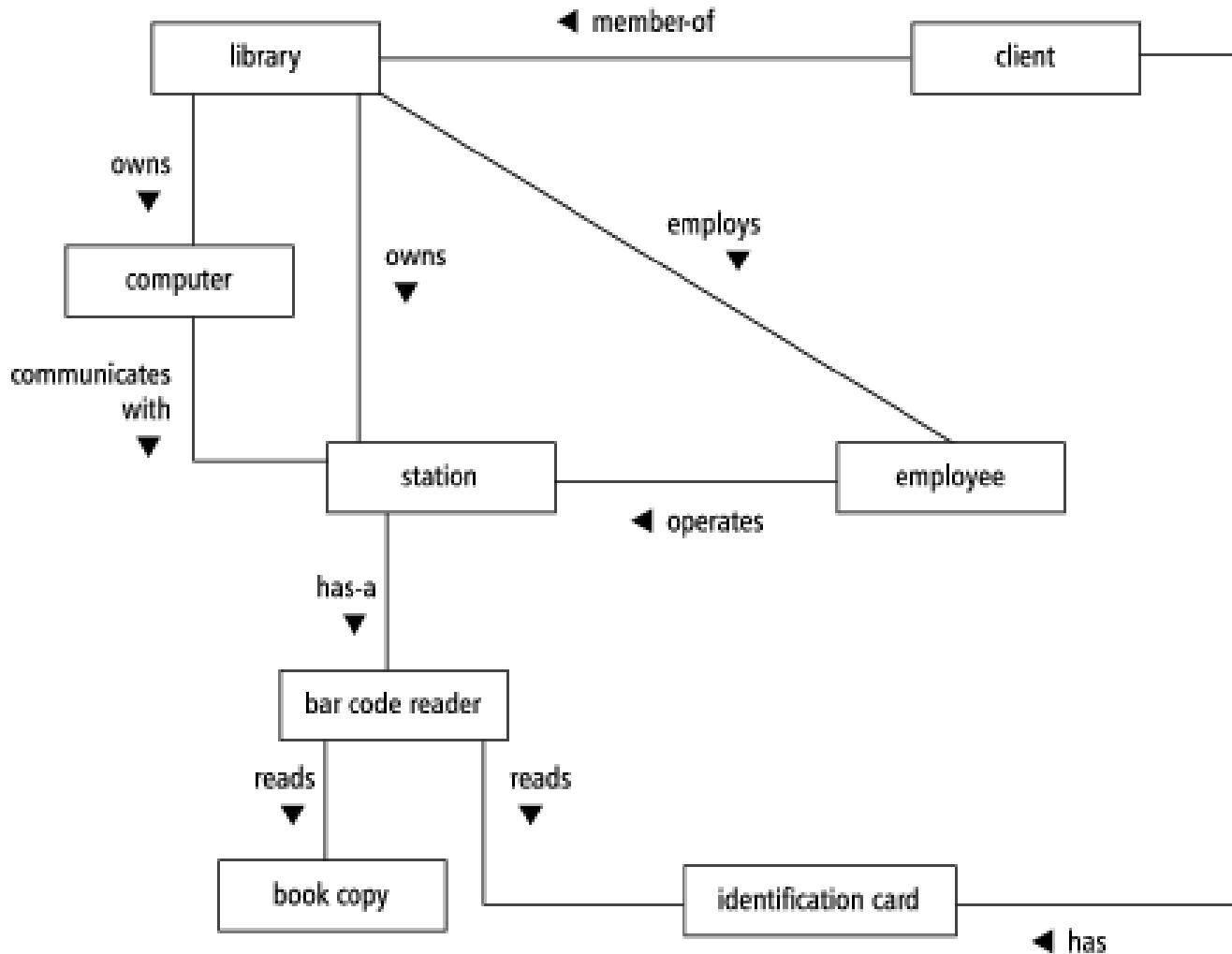
# Carefully consider candidate list

- 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”

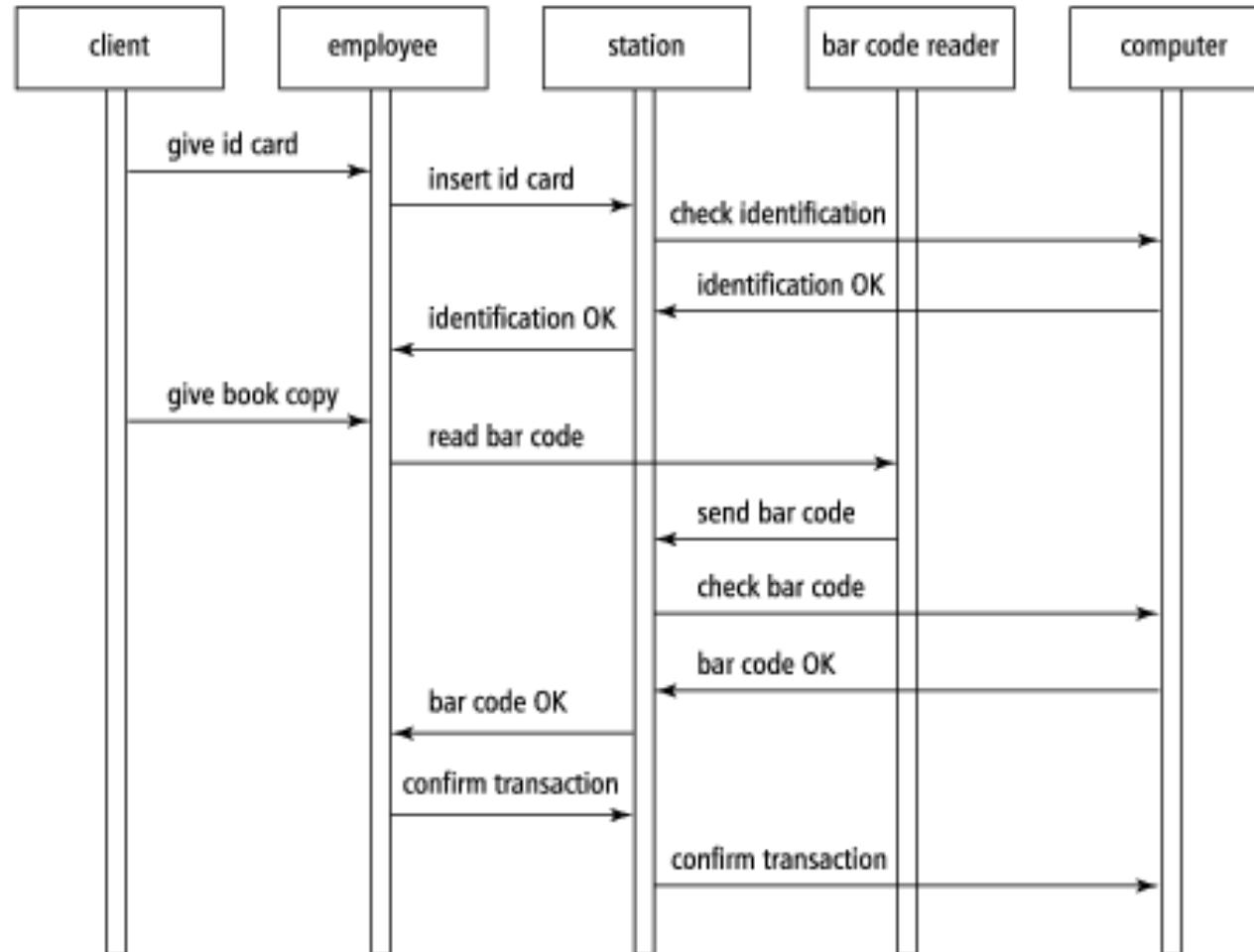
# Relationships

- 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



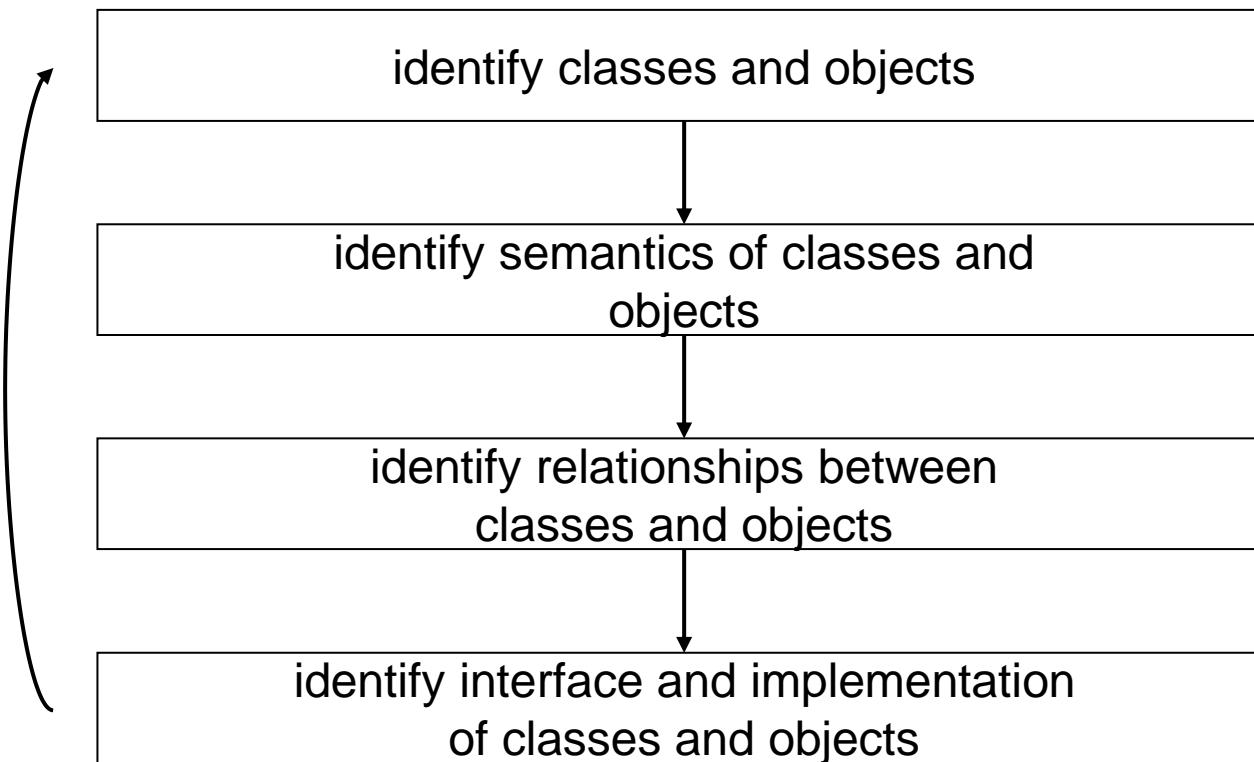
# OO as middle-out design

- 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

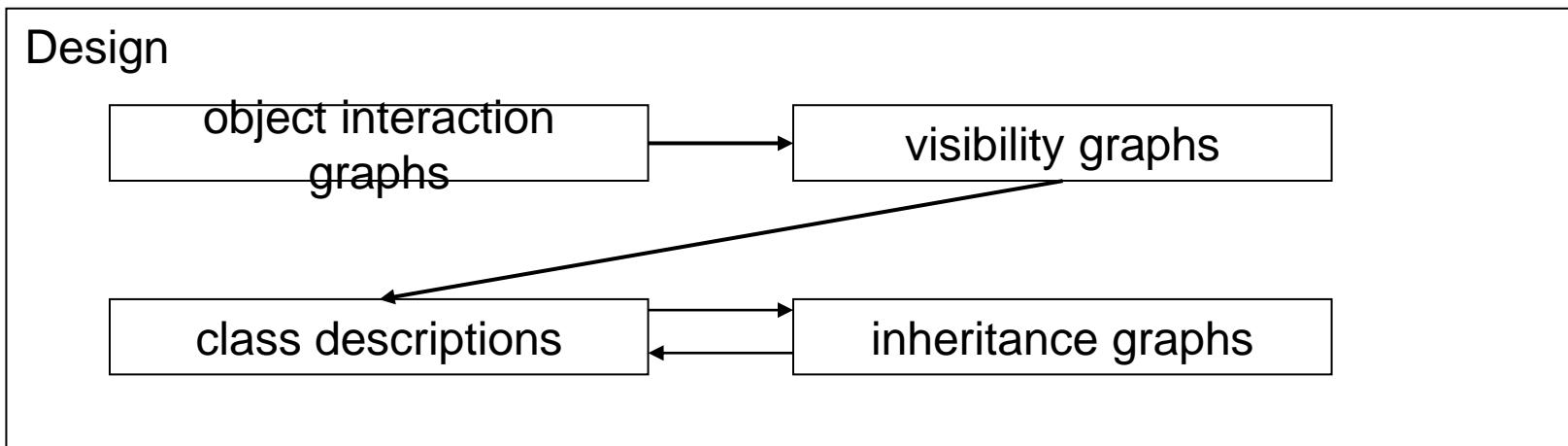
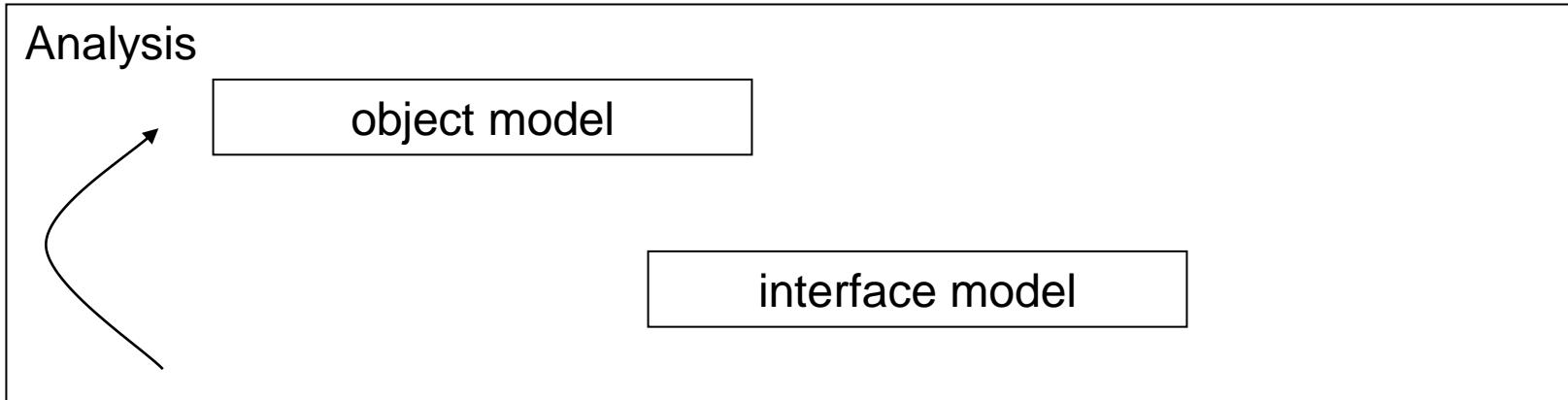
# OO design methods

- 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

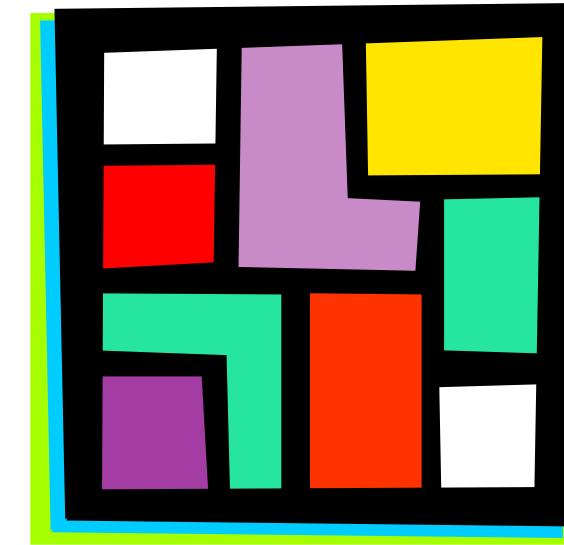
# Caveats when choosing a particular design method

- 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?

# 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

# 12.6 DESIGN DOCUMENTATION

- A requirements specification is developed during requirements engineering. That document serves a number of purposes. It specifies the users' requirements and as such it often has legal meaning. It is also the starting point for the design and thus serves another class of user.
- The same applies to the design documentation. The description of the design serves different users, who have different needs. A proper organization of the design documentation is therefore very important.

# Attributes in a design documentation

- IEEE Standard 1016 distinguishes ten attributes. These attributes are minimally required in each project. The attributes from IEEE Standard 1016 are:
  - **Identification:** the unique name of the component, for reference purposes
  - **Type:** the kind of component, such as subsystem, procedure, class, or file
  - **Purpose:** the specific purpose of the component (this will refer to the requirements specification)
  - **Function:** what the component accomplishes (for a number of components, this information occurs in the requirements specification)
  - **Subordinates:** the components of which the present entity is composed (it identifies a static is-composed-of relation between entities)
  - **Dependencies:** a description of the relationships with other components. It concerns the uses relation, see [Section 12.1.5](#), and includes more detailed information on the nature of the interaction (including common data structures, order of execution, parameter interfaces, and so on).
  - **Interface:** a description of the interaction with other components

# 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

# Software testing

Main issues:

- There are a great many testing techniques
- Often, only the final code is tested

- Angelo Gargantini
- Corso ing. Del sw AA2122



# Nasty question

- Suppose you are being asked to lead the team to test the software that controls a new X-ray machine. Would you take that job?
- Would you take it if you could name your own price?
- What if the contract says you'll be charged with murder in case a patient dies because of a mal-functioning of the software?

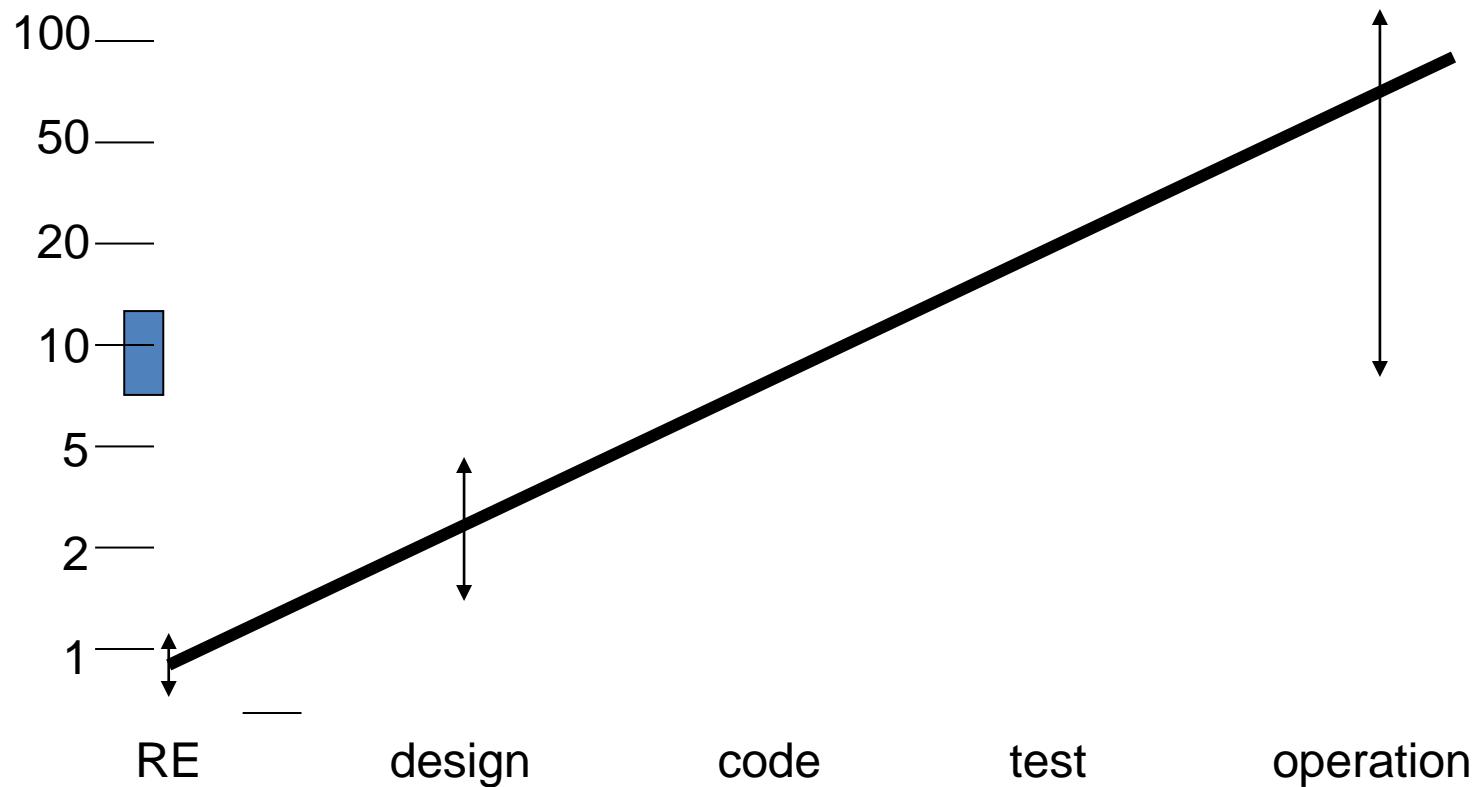
# Overview

- Preliminaries
- All sorts of test techniques
- Comparison of test techniques
- Software reliability

# State-of-the-Art

- 30-85 errors are made per 1000 lines of source code
- extensively tested software contains 0.5-3 errors per 1000 lines of source code
- testing is postponed, as a consequence: the later an error is discovered, the more it costs to fix it.
- error distribution: 60% design, 40% implementation. 66% of the design errors are not discovered until the software has become operational.

# Relative cost of error correction



# Lessons

- Many errors are made in the early phases
- These errors are discovered late
- Repairing those errors is costly
- ⇒ It pays off to start testing real early

# How then to proceed?

- Testing software shows only the presence of errors, not their absence.
- Exhaustive testing most often is not feasible
- Random statistical testing does not work either if you want to find errors
- Therefore, we look for systematic ways to proceed during testing

# Exhaustive testing

- Un test ideale è quello **esaustivo** in cui provo tutti gli input D possibili
- è fattibile? Considera questo esempio:

```
static int sum(int a, int b) return a + b;
```

- un int è un numero binario di 32 bit, quindi ci sono solo  $2^{32} \times 2^{32} = 2^{64}$  ca  $10^{21}$  test ad un nanosecondo ( $10^{-9}$ ) per test case circa 30000 anni.
- **se D è infinito? (ad esempio un sistema reattivo)**
- Il test esaustivo non è fattibile:  $T \subset D$
- Devo selezionare un sottoinsieme di D

# Classification of testing techniques

- Classification based on the criterion to measure the adequacy of a set of test cases:
  - coverage-based testing
  - fault-based testing
  - error-based testing
- Classification based on the source of information to derive test cases:
  - black-box testing (functional, specification-based)
  - white-box testing (structural, program-based)

# Some preliminary questions



- What exactly is an error?
- How does the testing process look like?
- When is test technique A superior to test technique B?
- What do we want to achieve during testing?
- When to stop testing?

# **Definizioni Base**

# **Terminologia base del testing**

**Dobbiamo concordare alcune parole che useremo da qui in avanti:**

- alcuni termini hanno un significato chiaro:
  - testing, debugging, ...
- altri sono definiti in modo non ambiguo da standard
  - Ad esempio dalla IEEE Standard Glossary of Software Engineering Terminology
    - difetto, bug, ...
- altri sono usati in modo ambiguo
  - Proponiamo un uso e cerchiamo di essere consistenti
    - errore, ...

# Malfunzionamento

Un **failure** o **guasto** o **malfunzionamento** è il funzionamento non corretto del programma

- legato quindi al comportamento che si osserva durante l'esecuzione

## Esempio:

```
// programma che dovrebbe restituire il doppio  
// del valore passato come parametro  
  
int raddoppia(int x) {  
    return x*x ;  
}
```

se chiamo raddoppia(3) noto un **malfunzionamento**  
se chiamo raddoppia(2) **non vi e' malfunzionamento**

# Difetto

Un **difetto** o **anomalia** o **fault** o **bug** è un elemento del programma sorgente non corrispondente alle aspettative

- riguarda quindi la parte statica.
- **uno o più difetti possono causare malfunzionamenti**
- Nell'esempio:

```
int raddoppia(int x) {  
    return x*x ;  
}
```

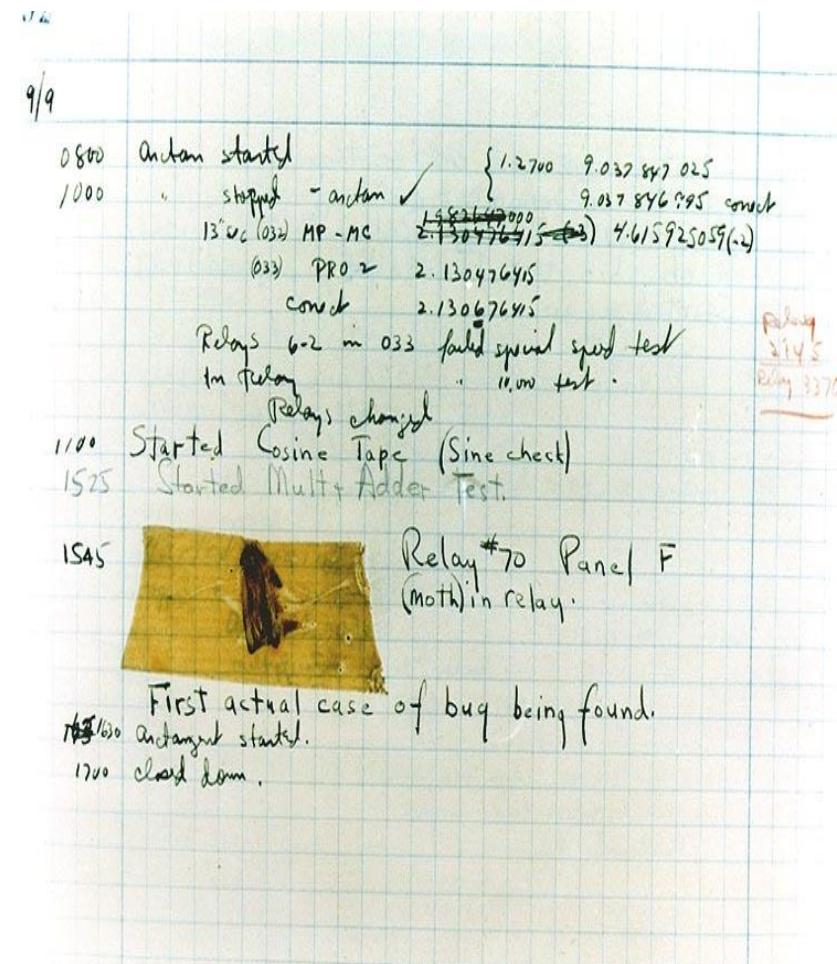
**il difetto** e'  $*x$  invece che  $*2$ .

## Nota

- un programma può avere molti difetti e non presentare alcun malfunzionamento.
- scopo del testing e' quello di evidenziare difetti mediante malfunzionamenti.

# Bug

Poerchè si usa bug? il 9 settembre 1947 il tenente Hopper ed il suo gruppo stavano cercando la causa del malfunzionamento di un computer Mark II quando, con stupore, si accorsero che una falena si era incastrata tra i circuiti. Dopo aver rimosso l'insetto (alle ore 15.45), il tenente incollò la falena rimossa sul registro del computer e annotò: «1545. Relay #70 Panel F (moth) in relay. First actual case of bug being found».



# Error, fault, failure



- an *error* is a human activity resulting in software containing a fault
- a *fault* is the manifestation of an error
- a *fault* may result in a failure

# When exactly is a failure a failure?

- Failure is a relative notion: e.g. a failure w.r.t. the specification document
- *Verification*: evaluate a product to see whether it satisfies the conditions specified at the start:  
*Have we built the system right?*
- *Validation*: evaluate a product to see whether it does what we think it should do:  
*Have we built the right system?*

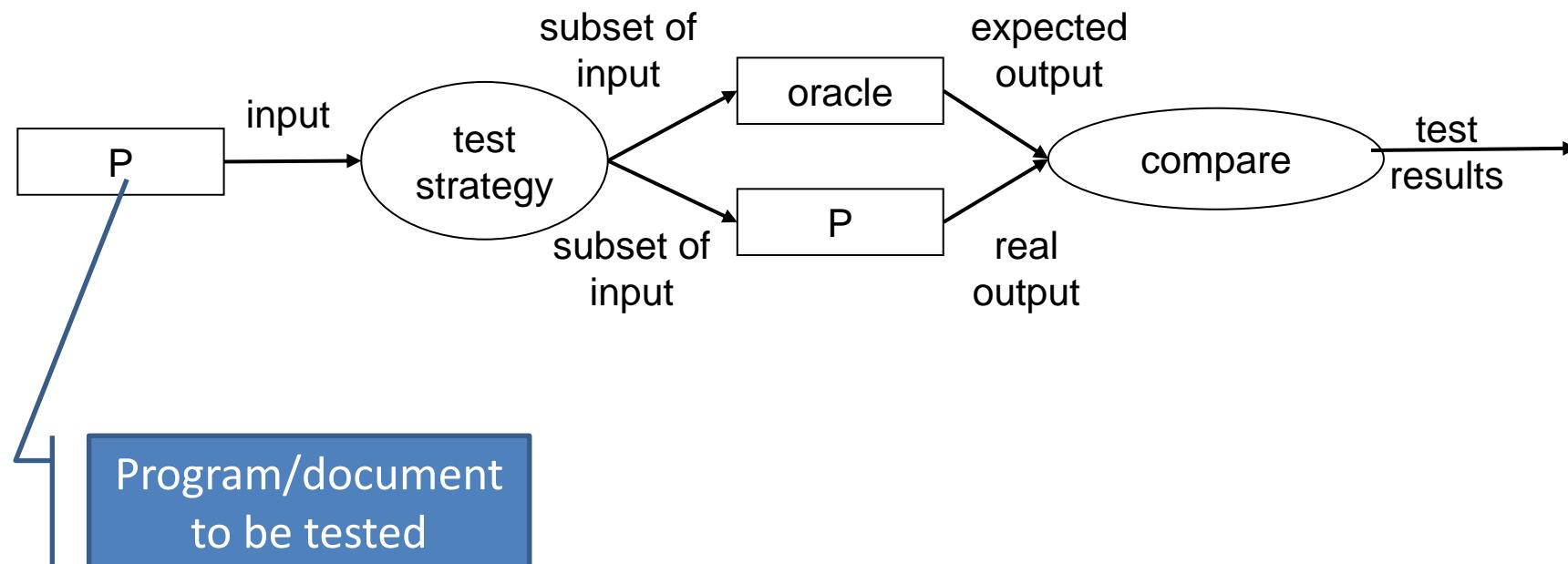
# V&V and testing

- In this respect, a distinction is often made between ‘verification’ and ‘validation’. The *IEEE Glossary* defines verification as the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Verification thus tries to answer the question: Have we built the system right?
- The term ‘validation’ is defined in the *IEEE Glossary* as the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. Validation then boils down to the question: Have we built the right system?
- **Testing refers to both (sometimes we do verification sometimes validation)**

# What is our goal during testing?

- Objective 1: find as many faults as possible
- Objective 2: make you feel confident that the software works OK

# Testing process



# Two examples

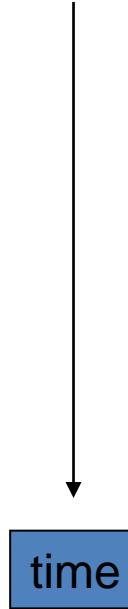
- Requirement document
  - “when user select a city, the correct zip code will appear in the dialog”
  - test: opens app, select Bergamo -> Zip 24100
- Code
  - The methods odd(X) returns true if only is X is odd
  - Test: odd(3) ... expected -> true

# Example constructive approach

- Task: test module that sorts an array  $A[1..n]$ .  $A$  contains integers;  $n < 1000$
- Solution: take  $n = 0, 1, 37, 999, 1000$ . For  $n = 37, 999$ , take  $A$  as follows:
  - $A$  contains random integers
  - $A$  contains increasing integers
  - $A$  contains decreasing integers
- These are *equivalence classes*: we assume that one element from such a class suffices
- This works if the partition is perfect

# Testing models

- *Demonstration*: make sure the software satisfies the specs
- *Destruction*: try to make the software fail
- *Evaluation*: detect faults in early phases
- *Prevention*: prevent faults in early phases



# Demonstration

- The primary goal of the demonstration model is to make sure that the program runs and solves the problem.
- If the software passes all tests from the test set, it is claimed to satisfy the requirements.
- Look for test cases which support the claim that the sw is **working**
  - Can be dangerous.

# Destruction

- A program should be tested with the purpose of finding as many faults as possible.
- A test can only be considered successful if it leads to the discovery of at least one fault.
- The test set is then judged by its ability to detect faults.
  - How: for example by injecting faults

# Testing and the life cycle

- requirements engineering
  - criteria: completeness, consistency, feasibility, and **testability**.
    - Completeness: watch for the omission of functions or products,
    - Consistency: components do not contradict each other
  - typical errors: missing, wrong, and extra information
  - determine testing strategy
  - generate functional test cases
  - test specification, through reviews and the like

# Testing and the life cycle

- design
  - functional and structural tests can be devised on the basis of the decomposition
  - the design itself can be tested (against the requirements)
    - . This includes tracing elements from the requirements specification to the corresponding elements in the design description, and vice versa.
  - formal verification techniques
  - the architecture can be evaluated

# Testing and the life cycle (cnt'd)

- implementation
  - check consistency implementation and previous documents
  - code-inspection and code-walkthrough
  - all kinds of functional and structural test techniques
  - extensive tool support
  - formal verification techniques
- maintenance
  - regression testing: either retest all, or a more selective retest

# Test-Driven Development (TDD)

- *First* write the tests, *then* do the design/implementation
- Part of agile approaches like XP
- Supported by tools, eg. JUnit
- Is more than a mere test technique; it subsumes part of the design work

# Steps of TDD

1. Add a test
2. Run all tests, and see that the system fails
3. Make a small change to make the test work
4. Run all tests again, and see they all run properly
5. Refactor the system to improve its design and remove redundancies

# Test Stages

- module-unit testing and integration testing
  - bottom-up versus top-down testing
- system testing
- acceptance testing
- installation testing

# VERIFICATION AND VALIDATION PLANNING AND DOCUMENTATION

- Like the other phases and activities of the software development process, the testing activities need to be carefully planned and documented.

# Test documentation

## (IEEE 928)

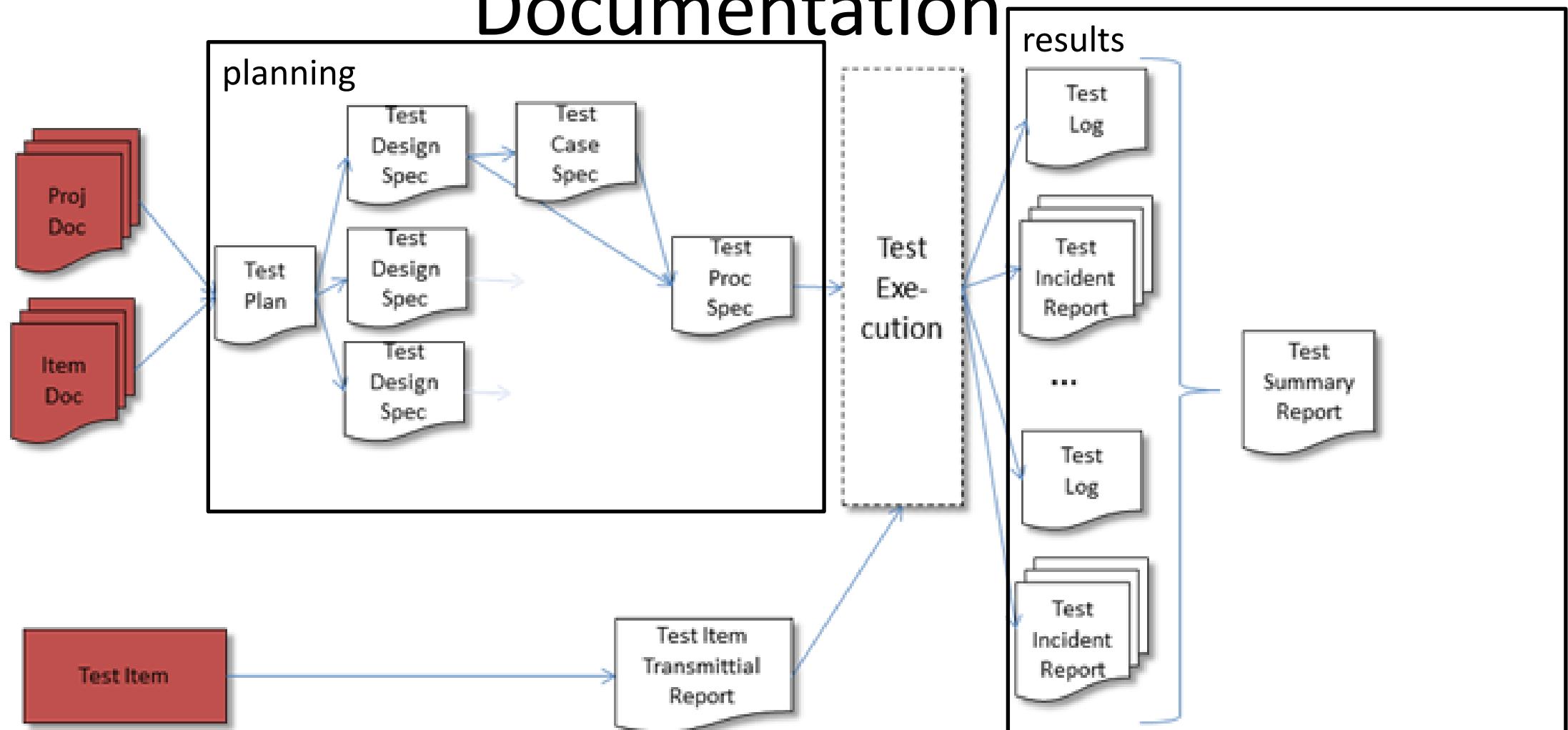


- Test plan
- Test design specification
- Test case specification
- Test procedure specification
- Test item transmittal report
- Test log
- Test incident report
- Test summary report

# Test documentation

- The **Test Plan** is a document describing the scope, approach, resources, and schedule of intended test activities. Describes in detail the test items, features to be tested, testing tasks, who will do each task, and any risks that require contingency planning.
- The **Test Design Specification** defines, for each component or software feature the details of the test approach and identifies the associated tests.
- The **Test Case Specification** defines inputs, predicted outputs, and execution conditions for each test item.
- The **Test Procedure Specification** defines the sequence of actions for the execution of each test. Together with the Test Plan, these documents describe the input to the test execution.
- The Test Item Transmittal Report specifies which items are going to be tested. It lists the items, specifies where to find them, and gives the status of each item. It constitutes the release information for a given test execution.
- The final three reports are the output of the test execution. The Test Log gives a chronological record of events. The Test Incident Report documents all events observed that require further investigation. In particular, this includes tests from which the outputs were not as expected. Finally, the Test Summary Report gives an overview and evaluation of the findings.

# IEEE 829 Standard for Software Test Documentation



Quelle: <http://www.pmqs.de>  
entnommen aus IEEE829

- STOP

# Overview

- Preliminaries
- All sorts of test techniques
  - **manual techniques**
  - coverage-based techniques
  - fault-based techniques
  - error-based techniques
- Comparison of test techniques
- Software reliability

# Manual Test Techniques

- static versus dynamic analysis
- compiler does a lot of static testing
- static test techniques
  - reading, informal versus peer review
  - walkthrough and inspections
  - correctness proofs, e.g., pre-and post-conditions:  $\{P\} S \{Q\}$
  - stepwise abstraction

# (Fagan) inspection



- Going through the code, statement by statement
- Team with ~4 members, with specific roles:
  - moderator: organization, chairperson
  - code author: silent observer
  - (two) inspectors, readers: paraphrase the code
- Uses checklist of well-known faults
- Result: list of problems encountered

# Example checklist

- Wrong use of data: variable not initialized, dangling pointer, array index out of bounds, ...
- Faults in declarations: undeclared variable, variable declared twice, ...
- Faults in computation: division by zero, mixed-type expressions, wrong operator priorities, ...
- Faults in relational expressions: incorrect Boolean operator, wrong operator priorities, .
- Faults in control flow: infinite loops, loops that execute n-1 or n+1 times instead of n, ...

# Overview

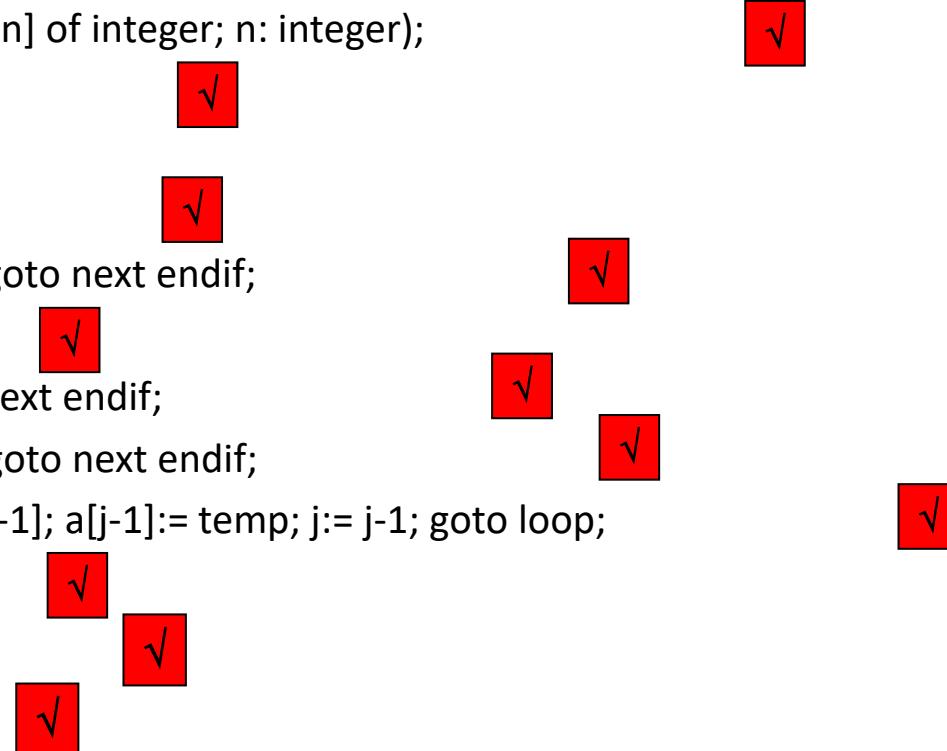
- Preliminaries
- All sorts of test techniques
  - manual techniques
  - **coverage-based techniques**
  - fault-based techniques
  - error-based techniques
- Comparison of test techniques
- Software reliability

# Coverage-based testing

- Goodness is determined by the coverage of the product by the test set so far: e.g., % of statements or requirements tested
- Often based on control-flow graph of the program
- Three techniques:
  - control-flow coverage
  - data-flow coverage
  - coverage-based testing of requirements

# Example of control-flow coverage

```
procedure bubble (var a: array [1..n] of integer; n: integer);
    var i, j: temp: integer;
begin
    for i:= 2 to n do
        if a[i] >= a[i-1] then goto next endif;
        j:= i;
    loop: if j <= 1 then goto next endif;
        if a[j] >= a[j-1] then goto next endif;
        temp:= a[j]; a[j]:= a[j-1]; a[j-1]:= temp; j:= j-1; goto loop;
next: skip;
enddo
end bubble;
```



input: n=2, a[1] = 5, a[2] = 3

# Example of control-flow coverage (cnt'd)

```
procedure bubble (var a: array [1..n] of integer; n: integer);
    var i, j: temp: integer;
begin
    for i:= 2 to n do
        if a[i] >= a[i-1] then goto next endif;
        j:= i; a[i]=a[i-1]
loop: if j <= 1 then goto next endif;
        if a[j] >= a[j-1] then goto next endif;
        temp:= a[j]; a[j]:= a[j-1]; a[j-1]:= temp; j:= j-1; goto loop;
next: skip;
enddo
end bubble;
```

input: n=2, a[1] = 5, a[2] = 3

# Control-flow coverage

- This example is about *All-Nodes coverage, statement coverage*
- A stronger criterion: *All-Edges coverage, branch coverage*
- Variations exercise all combinations of elementary predicates in a branch condition
- Strongest: *All-Paths coverage* ( $\equiv$  exhaustive testing)
- Special case: all linear independent paths, the *cyclomatic number criterion*

# Data-flow coverage

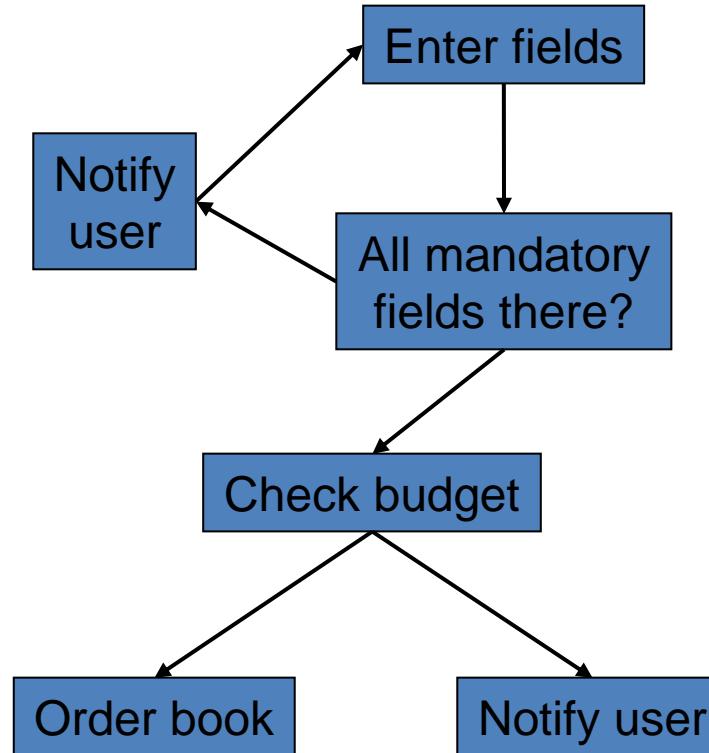
- Looks how variables are treated along paths through the control graph.
- Variables are *defined* when they get a new value.
- A definition in statement X is *alive* in statement Y if there is a path from X to Y in which this variable is not defined anew. Such a path is called *definition-clear*.
- We may now test all definition-clear paths between each definition and each use of that definition and each successor of that node: *All-Uses coverage*.

# Coverage-based testing of requirements

- Requirements may be represented as graphs, where the nodes represent elementary requirements, and the edges represent relations (like yes/no) between requirements.
- And next we may apply the earlier coverage criteria to this graph

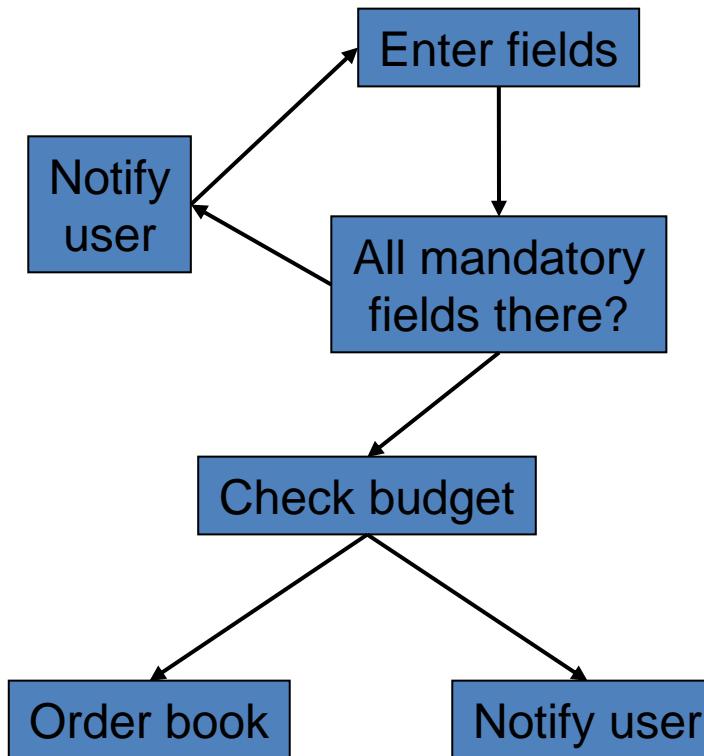
# Example translation of requirements to a graph

A user may order new books. He is shown a screen with fields to fill in. Certain fields are mandatory. One field is used to check whether the department's budget is large enough. If so, the book is ordered and the budget reduced accordingly.



# Similarity with Use Case success scenario

1. User fills form
2. Book info checked
3. Dept budget checked
4. Order placed
5. User is informed



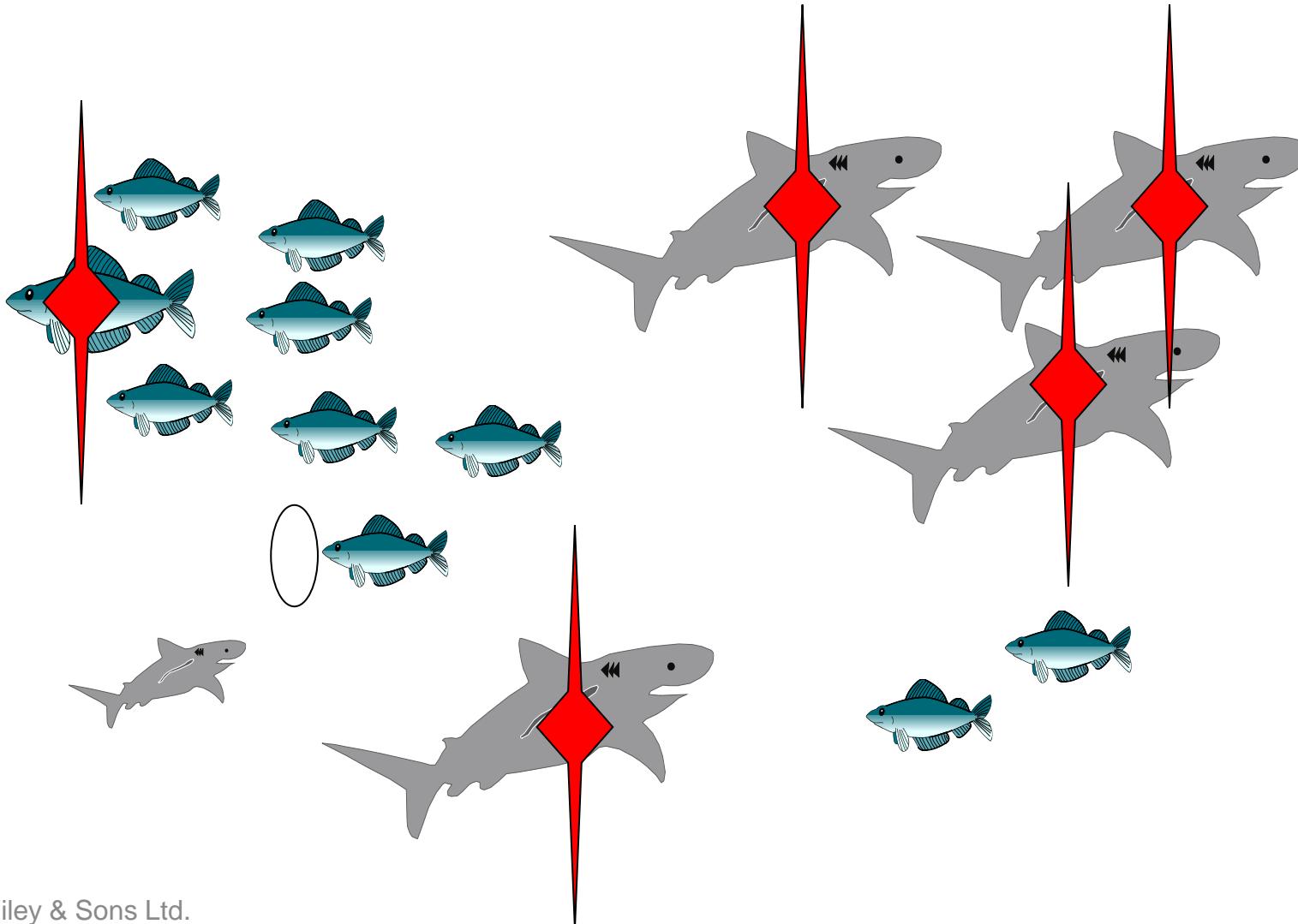
# Overview

- Preliminaries
- All sorts of test techniques
  - manual techniques
  - coverage-based techniques
  - **fault-based techniques**
  - error-based techniques
- Comparison of test techniques
- Software reliability

# Fault-based testing

- In coverage-based testing, we take the structure of the artifact to be tested into account
- In fault-based testing, we do not *directly* consider this artifact
- We just look for a test set with a high ability to detect faults
- Two techniques:
  - Fault seeding
  - Mutation testing

# Fault seeding



# Mutation testing

```
procedure insert(a, b, n, x);
begin bool found:= false;
    for i:= 1 to n do
        if a[i] = x
            then found:= true; goto leave endif
    enddo;
leave:
    if found
        then b[i]:= b[i] + 1
    else n:= n+1; a[n]:= x; b[n]:= 1
    endif
end insert;
```

The diagram illustrates mutation points in the code. A blue box labeled "n-1" is positioned above the "n" in "n-1". A blue box labeled "2" is positioned above the "2" in "n+1". A blue box labeled "-" is positioned below the "n" in "b[n]:= 1".

# Mutation testing (cnt'd)

```
procedure insert(a, b, n, x);
begin bool found:= false;
    for i:= 1 to n do           \   n-1
        if a[i] = x
            then found:= true; goto leave endif
    enddo;
leave:
    if found
        then b[i]:= b[i] + 1
        else n:= n+1; a[n]:= x; b[n]:= 1
    endif
end insert;
```

# How tests are treated by mutants

- Let  $P$  be the original, and  $P'$  the mutant
- Suppose we have two tests:
  - $T_1$  is a test, which inserts an element that equals  $a[k]$  with  $k < n$
  - $T_2$  is another test, which inserts an element that does not equal an element  $a[k]$  with  $k < n$
- Now  $P$  and  $P'$  will behave the same on  $T_1$ , while they differ for  $T_2$
- In some sense,  $T_2$  is a “better” test, since it in a way tests this upper bound of the for-loop, which  $T_1$  does not

# How to use mutants in testing

- If a test produces different results for one of the mutants, that mutant is said to be *dead*
- If a test set leaves us with many live mutants, that test set is of low quality
- If we have  $M$  mutants, and a test set results in  $D$  dead mutants, then the *mutation adequacy score* is  $D/M$
- A larger mutation adequacy score means a better test set

# Strong vs weak mutation testing

- Suppose we have a program P with a component T
  - We have a mutant T' of T
  - Since T is part of P, we then also have a mutant P' of P
- 
- In *weak mutation testing*, we require that T and T' produce different results, but P and P' may still produce the same results
  - In *strong mutation testing*, we require that P and P' produce different results

# Assumptions underlying mutation testing

- *Competent Programmer Hypothesis*: competent programmers write programs that are approximately correct
- *Coupling Effect Hypothesis*: tests that reveal simple fault can also reveal complex faults

# Overview

- Preliminaries
- All sorts of test techniques
  - manual techniques
  - coverage-based techniques
  - fault-based techniques
  - **error-based techniques**
- Comparison of test techniques
- Software reliability

# Error-based testing

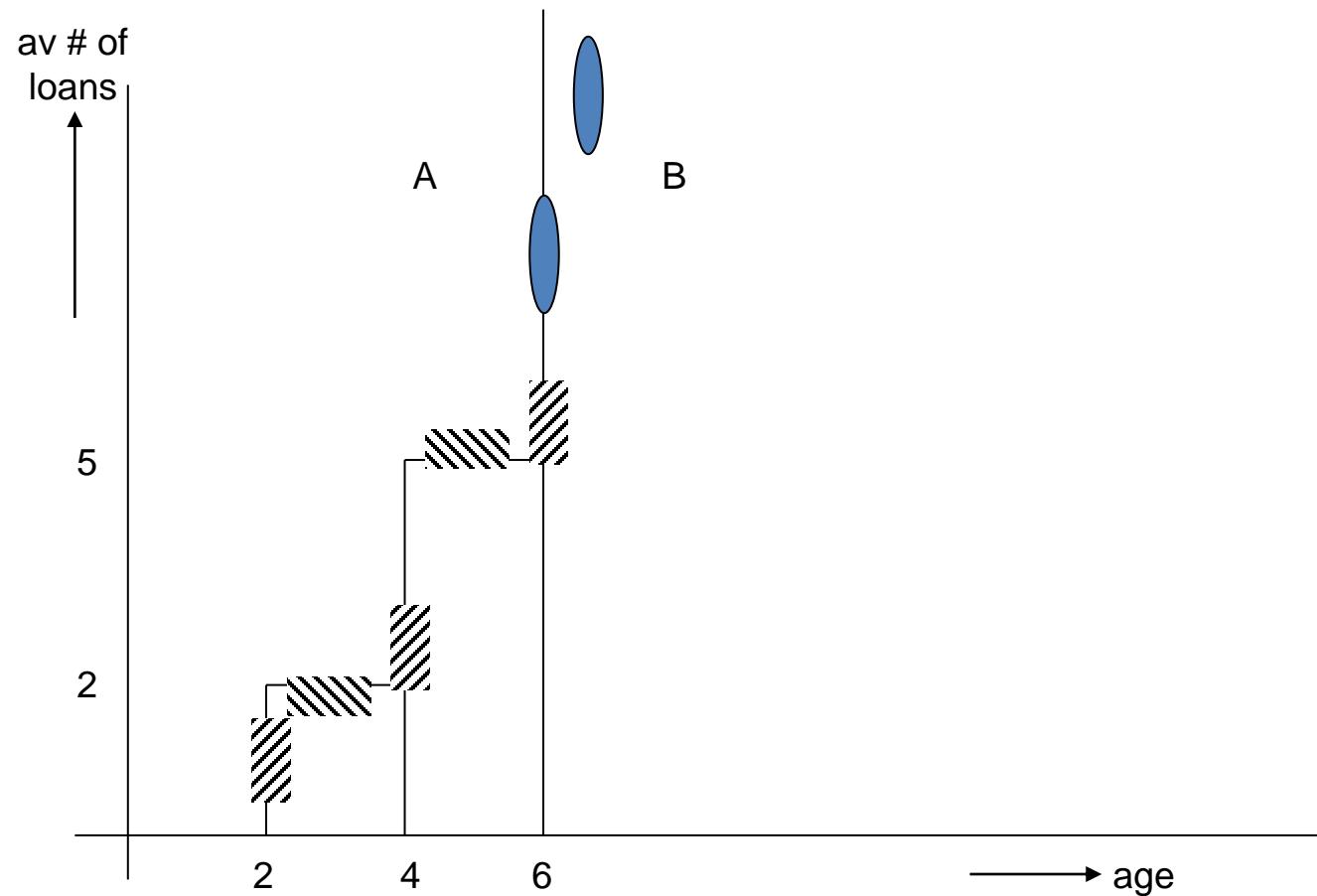
- Decomposes input (such as requirements) in a number of subdomains
- Tests inputs from each of these subdomains, and especially points near and just on the boundaries of these subdomains -- those being the spots where we tend to make errors
- In fact, this is a systematic way of doing what experienced programmers do: test for 0, 1, nil, etc

# Error-based testing, example

Example requirement:

Library maintains a list of “hot” books. Each new book is added to this list. After six months, it is removed again. Also, if book is more than four months on the list, and has not been borrowed more than four times a month, or it is more than two months old and has been borrowed at most twice, it is removed from the list.

# Example (cnt'd)



# Strategies for error-based testing

- An ON point is a point on the border of a subdomain
  - If a subdomain is open w.r.t. some border, then an OFF point of that border is a point just inside that border
  - If a subdomain is closed w.r.t. some border, then an OFF point of that border is a point just outside that border
- 
- So the circle on the line  $age=6$  is an ON point of both A and B
  - The other circle is an OFF point of both A and B

# Strategies for error-based testing (cnt'd)

- Suppose we have subdomains  $D_i$ ,  $i=1,..n$
- Create test set with  $N$  test cases for  $ON$  points of each border  $B$  of each subdomain  $D_i$ , and at least one test case for an  $OFF$  point of each border
- This set is called  $N*1$  *domain adequate*

# Application to programs

if  $x < 6$  then ...

elseif  $x > 4$  and  $y < 5$  then ...

elseif  $x > 2$  and  $y \leq 2$  then ...

else ...

# Overview

- Preliminaries
- All sorts of test techniques
  - manual techniques
  - coverage-based techniques
  - fault-based techniques
  - error-based techniques
- Comparison of test techniques
- Software reliability

# Comparison of test adequacy criteria

- Criterion A is stronger than criterion B if, for all programs P and all test sets T, X-adequacy implies Y-adequacy
- In that sense, e.g., All-Edges is stronger than All-Nodes coverage (All-Edges “subsumes” All-Nodes)
- One problem: such criteria can only deal with paths that can be executed (are feasible). So, if you have dead code, you can never obtain 100% statement coverage. Sometimes, the subsumes relation only holds for the *feasible* version.

# Desirable properties of adequacy criteria

- applicability property
- non-exhaustive applicability property
- monotonicity property
- inadequate empty set property
- antiextensionality property
- general multiplicity change property
- antidecomposition property
- anticomposition property
- renaming property
- complexity property
- statement coverage property

# Experimental results

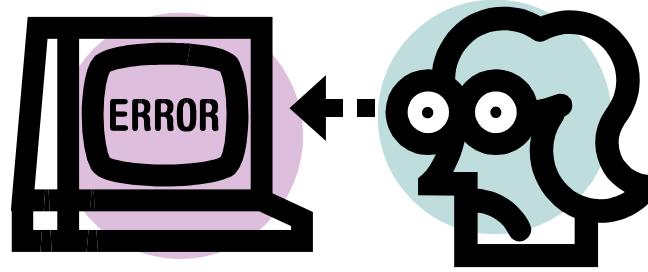
- There is no uniform best test technique
- The use of multiple techniques results in the discovery of *more* faults
- (Fagan) inspections have been found to be very cost effective
- Early attention to testing does pay off

# Overview

- Preliminaries
- All sorts of test techniques
  - manual techniques
  - coverage-based techniques
  - fault-based techniques
  - error-based techniques
- Comparison of test techniques
- Software reliability

# Software reliability

- Interested in expected number of *failures* (not faults)
- ... in a certain period of time
- ... of a certain product
- ... running in a certain environment



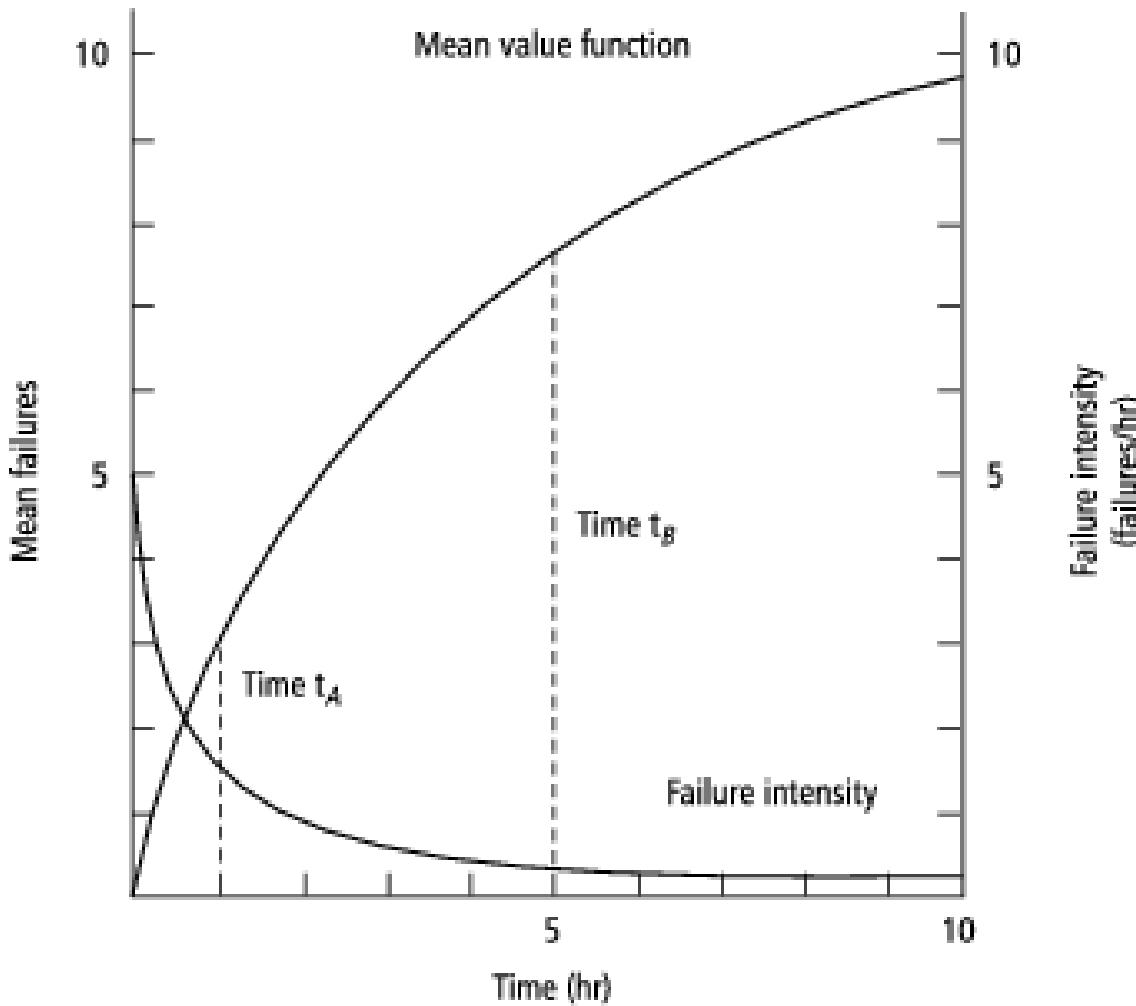
# Software reliability: definition

- Probability that the system will not fail during a certain period of time in a certain environment

# Failure behavior

- Subsequent failures are modeled by a stochastic process
- Failure behavior changes over time (e.g. because errors are corrected)  $\Rightarrow$  stochastic process is *non-homogeneous*
- $\mu(\tau)$  = average number of failures *until* time  $\tau$
- $\lambda(\tau)$  = average number of failures *at* time  $\tau$  (failure intensity)
- $\lambda(\tau)$  is the derivative of  $\mu(\tau)$

# Failure intensity $\lambda(\tau)$ and mean failures $\mu(\tau)$



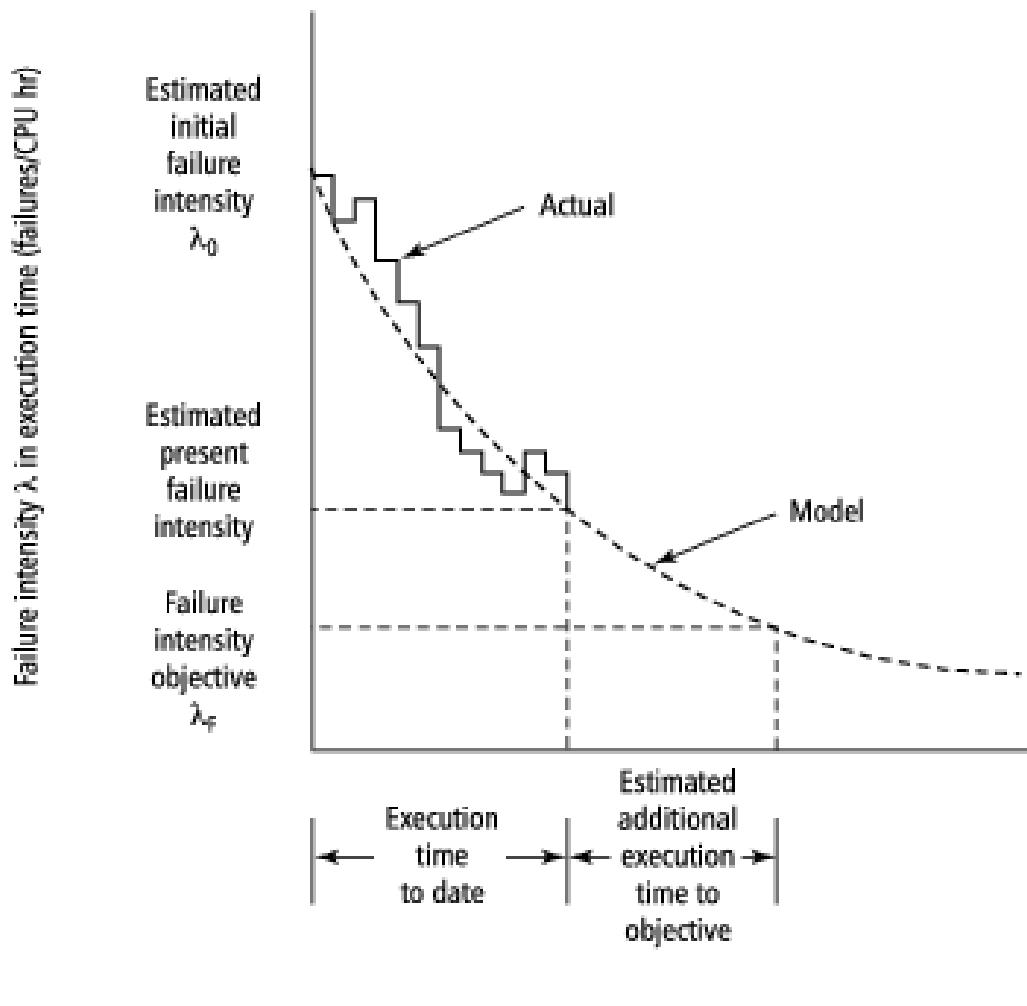
# Operational profile

- Input results in the execution of a certain sequence of instructions
- Different input  $\Rightarrow$  (probably) different sequence
- Input domain can thus be split in a series of equivalence classes
- Set of possible input classes together with their probabilities

# Two simple models

- Basic execution time model (BM)
  - Decrease in failure intensity is constant over time
  - Assumes uniform operational profile
  - Effectiveness of fault correction is constant over time
- Logarithmic Poisson execution time model (LPM)
  - First failures contribute more to decrease in failure intensity than later failures
  - Assumes non-uniform operational profile
  - Effectiveness of fault correction decreases over time

# Estimating model parameters (for BM)



# Summary

- Do test as early as possible
- Testing is a continuous process
- Design with testability in mind
- Test activities must be carefully planned, controlled and documented.
- No single reliability model performs best consistently



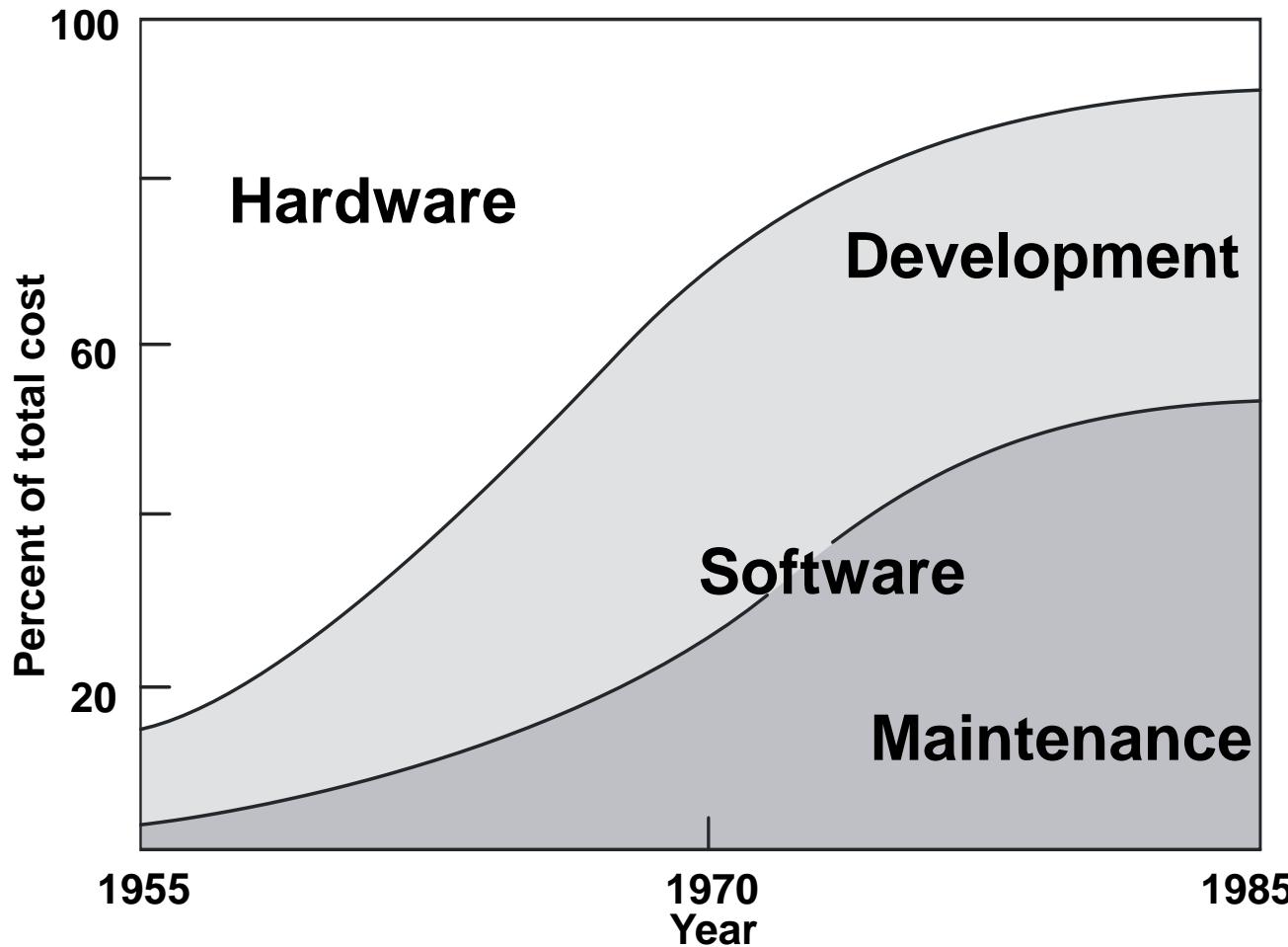
# Software Maintenance

Main issues:

- why maintenance is such an issue
- reverse engineering and its limitations
- how to organize maintenance

Angelo Gargantini AA 21/22

# Relative distribution of software/hardware costs



# Point to ponder #1



- Why does software maintenance cost so much?



# Software Maintenance, definition

The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment

# Maintenance is thus concerned with:

- correcting errors found after the software has been delivered
- adapting the software to changing requirements, changing environments, ...

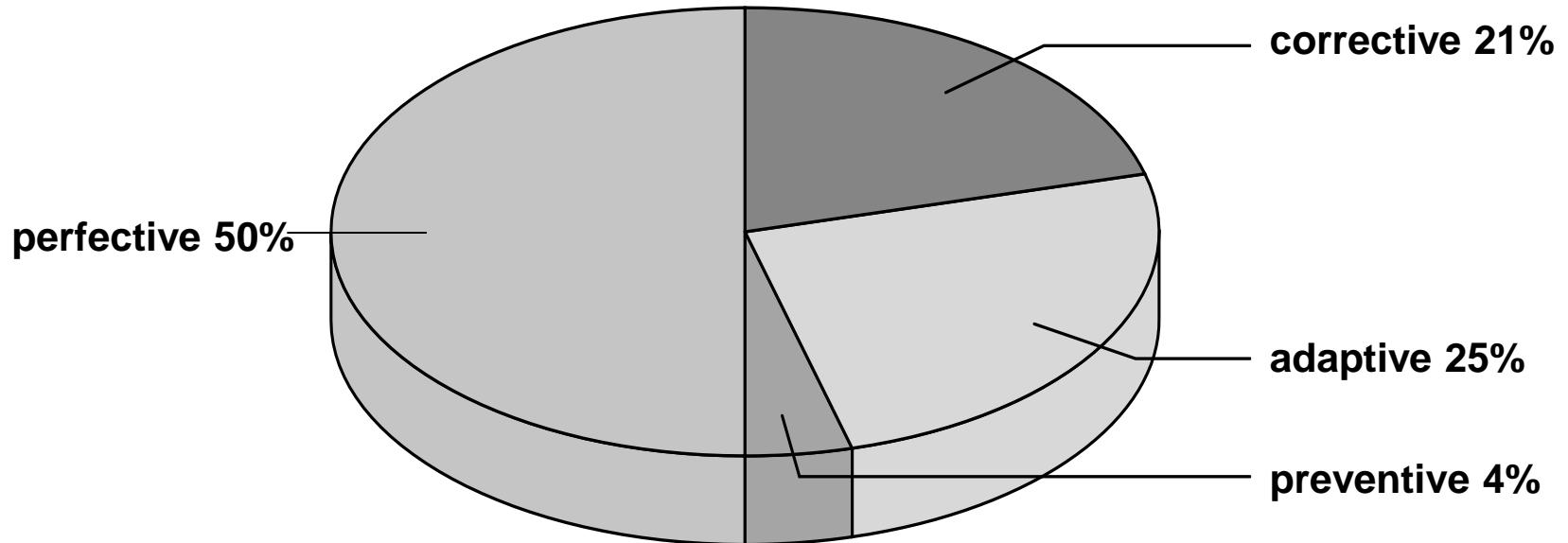
# Key to maintenance is in development

- Higher quality  $\Rightarrow$  less (corrective) maintenance
- Anticipating changes  $\Rightarrow$  less (adaptive and perfective) maintenance
- Better tuning to user needs  $\Rightarrow$  less (perfective) maintenance
- Less code  $\Rightarrow$  less maintenance

# Kinds of maintenance activities

- *corrective maintenance*: correcting errors
- *adaptive maintenance*: adapting to changes in the environment (both hardware and software)
- *perfective maintenance*: adapting to changing user requirements
- *preventive maintenance*: increasing the system's maintainability

# Distribution of maintenance activities



# Growth of maintenance problem

- 1975: ~75,000 people n maintenance (17%)
- 1990: 800,000 (47%)
- 2005: 2,500,000 (76%)
- 2015: ??

(Numbers from Jones (2006))

# Shift in type of maintenance over time

- Introductory stage: emphasis on user support
- Growth stage: emphasis on correcting faults
- Maturity: emphasis on enhancements
- Decline: emphasis on technology changes

# Major causes of maintenance problems

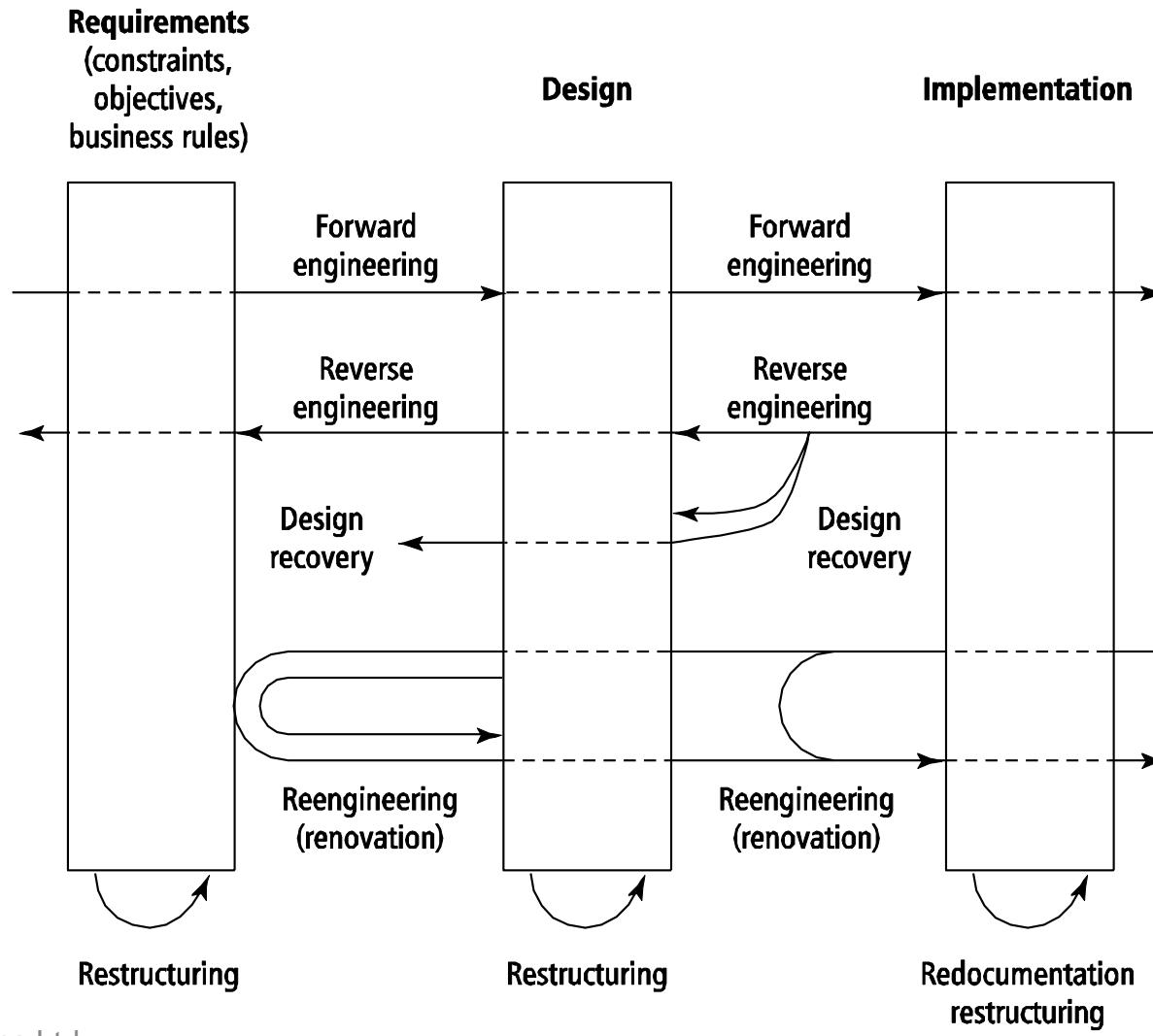


- Unstructured code
- Insufficient domain knowledge
- Insufficient documentation

# Laws of Software Evolution

- The ones that bear most on software maintenance are:
- **Law of continuing change:** A system that is being used undergoes continuing change, until it is judged more cost-effective to restructure the system or replace it by a completely new version.
- **Law of increasing complexity:** A program that is changed, becomes less and less structured (the entropy increases) and thus becomes more complex. One has to invest extra effort in order to avoid increasing complexity.
- **Law of code rewriting:** Netscape did it by making the single worst strategic mistake that any software company can make: They decided to rewrite the code from scratch.

# Reverse engineering



# Reverse engineering

- Does not involve any adaptation of the system
- Akin to reconstruction of a blueprint
- *Design recovery*: result is at higher level of abstraction
- *Redocumentation*: result is at same level of abstraction
- ‘the process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.’

# Restructuring

- Functionality does *not* change
- From one representation to another, at the same level of abstraction, such as:
  - From spaghetti code to structured code
  - *Refactoring* after a design step in agile approaches
  - Black box restructuring: add a wrapper
  - With platform change: *migration*

# Reengineering (renovation)



- Functionality *does* change
- Then reverse engineering step is followed by a forward engineering step in which the changes are made

# migration

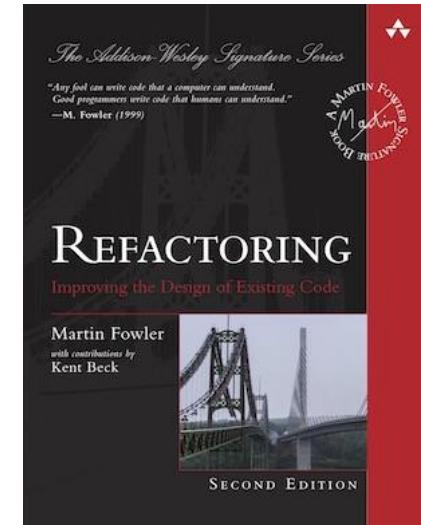
- These wrapping techniques do not change the platform on which the software is running. If a platform change is involved in the restructuring effort of a legacy system, this is known as **migration**. Migration to another platform is often done in conjunction with value-adding activities such as a change of interface or code improvements.

# Refactoring

- Entropy is not only caused by maintenance.
  - In agile methods, such as XP, it is an accepted intermediate stage. These methods have an explicit step to improve the code.
  - This is known as refactoring.
- Refactoring is a white-box method, in that it involves inspection of and changes to the code.

# Refactoring

- **Refactoring** is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
- Its heart is a series of small behavior preserving transformations. Each transformation (called a "refactoring") does little, but a sequence of these transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is kept fully working after each refactoring, reducing the chances that a system can get seriously broken during the restructuring.



# Examples of refactoring

- There is long catalog of refactoring actions:  
<https://refactoring.com/catalog/>
- Tool can automate refactoring under user supervision
- See eclipse
  - Basic refactoring like renaming ...
- More complex tool like structure101

# Refactoring in case of bad smells



- Long method
- Large class
- Primitive obsession
- Data clumps
- Switch statements
- Lazy class
- Duplicate code
- Feature envy
- Inappropriate intimacy
- ...

# Categories of bad smells

- Bloaters: something has grown too large
- Object-oriented abusers: OO not fully exploited
- Change preventers: hinder further evolution
- Dispensables: can be removed
- Encapsulators: deal with data communication
- Couplers: coupling too high

# Program comprehension

- Role of programming plans
- As-needed strategy vs systematic strategy
- Use of outside knowledge (domain knowledge, naming conventions, etc.)

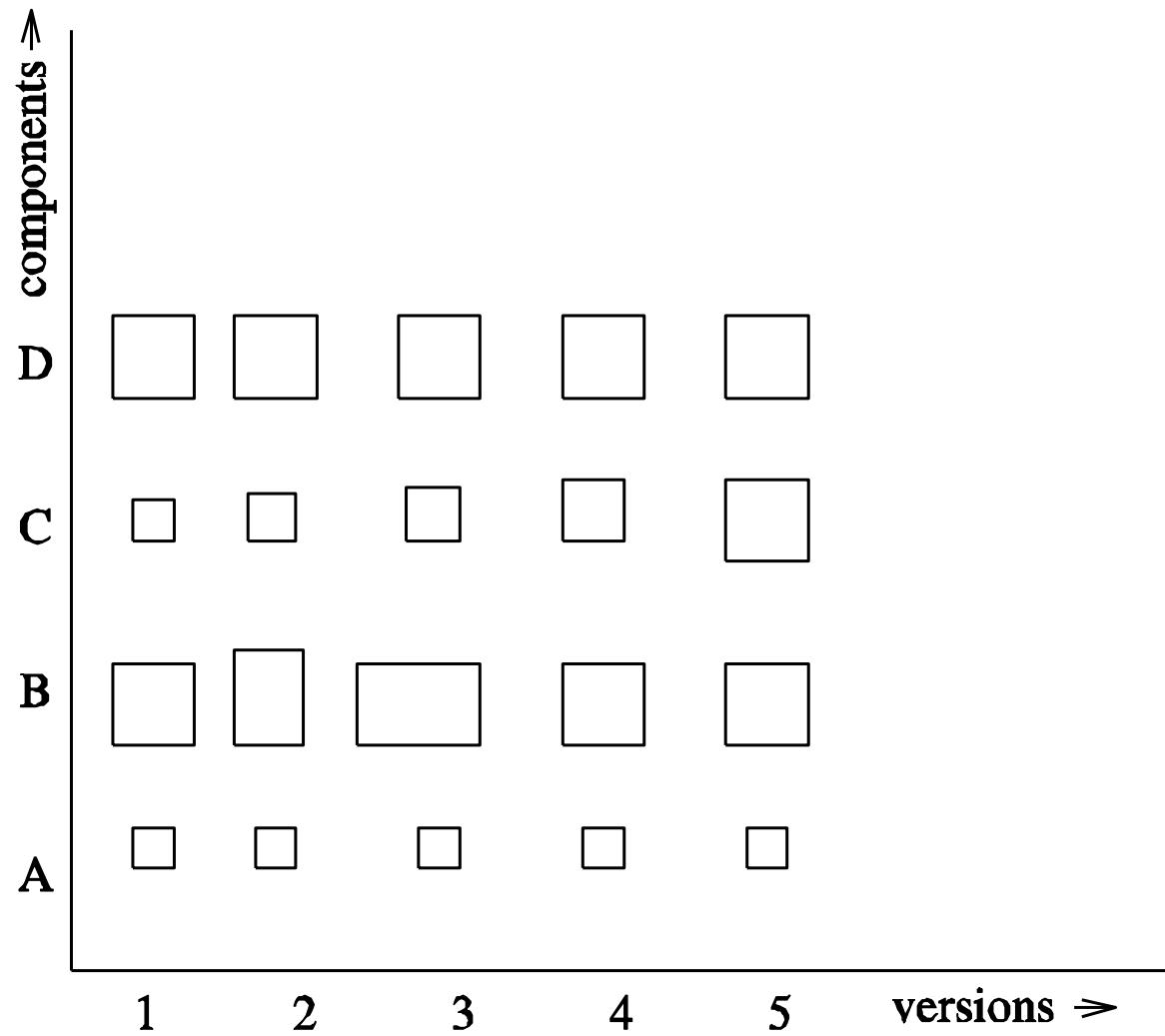
# Software maintenance tools

- Tools to ease perceptual processes (reformatters)
- Tools to gain insight in static structure
- Tools to gain insight in dynamic behavior
- Tools that inspect version history

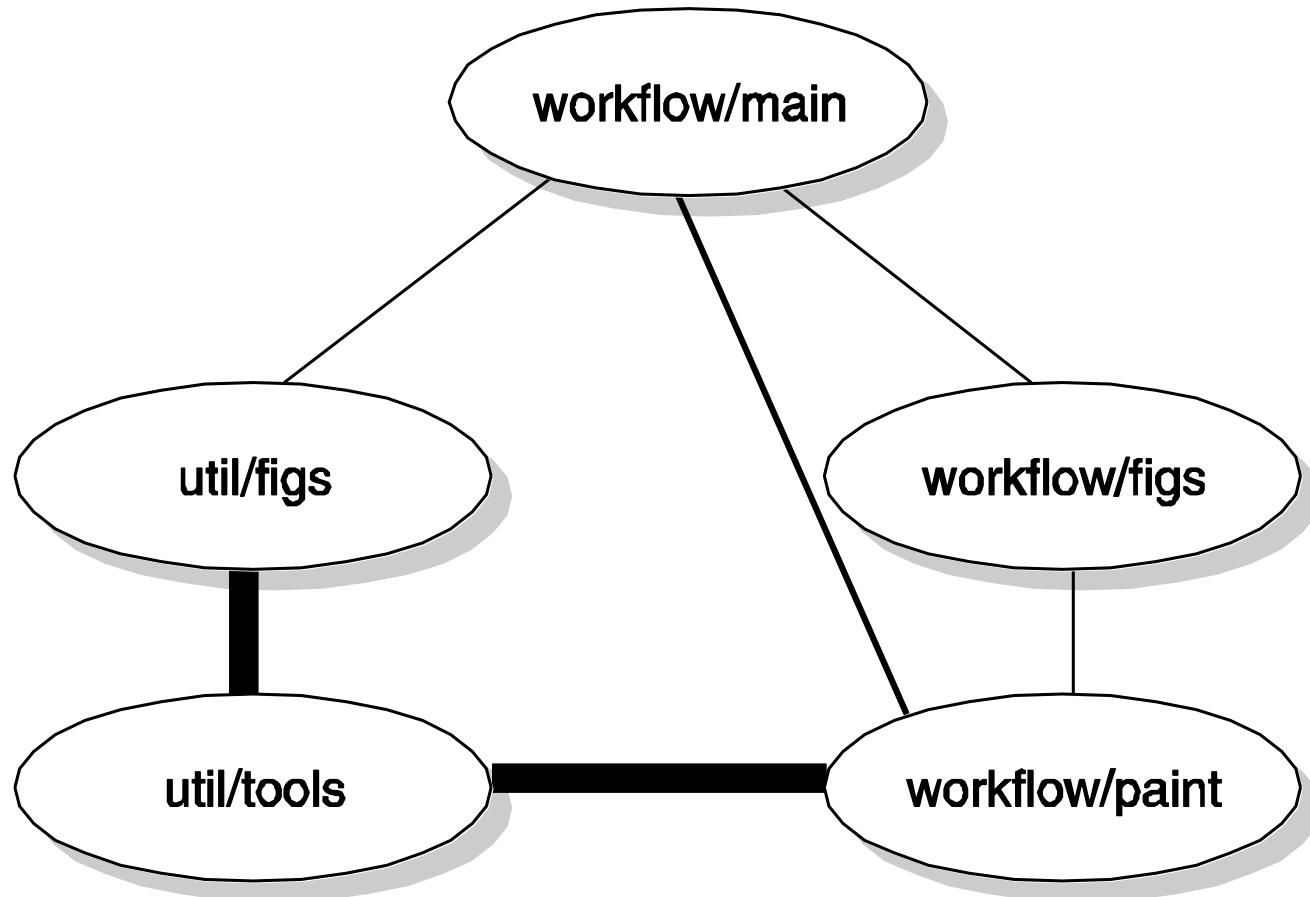
# Analyzing software evolution data

- Version-centered analysis: study differences between successive versions
- History-centered analysis: study evolution from a certain viewpoint (e.g. how often components are changed together)

# Example version-centered analysis



# Example history-centered analysis



# Organization of maintenance

- W-type: by work type (analysis vs programming)
- A-type: by application domain
- L-type: by life-cycle type (development vs maintenance)
- L-type found most often

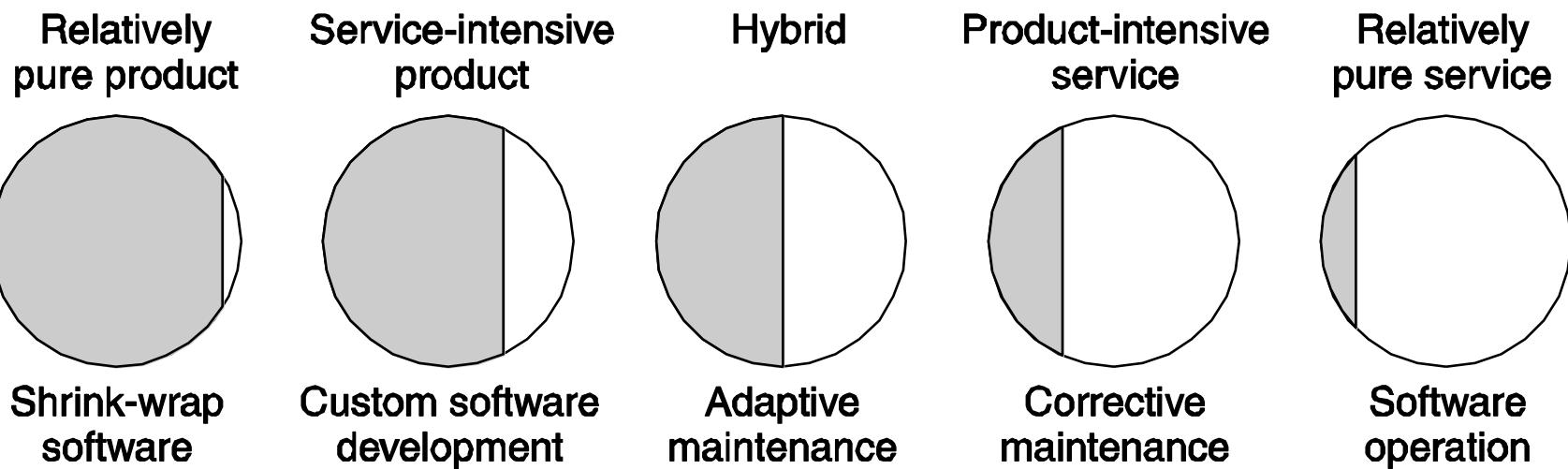
# Advantages of L-type departmentalization

- Clear accountability
- Development progress not hindered by unexpected maintenance requests
- Better acceptance test by maintenance department
- Higher QoS by maintenance department
- Higher productivity

# Disadvantages of L-type departmentalization

- Demotivation of maintenance personnel because of status differences
- Loss of system knowledge during system transfer
- Coordination costs
- Increased acceptance costs
- Duplication of communication channels with users

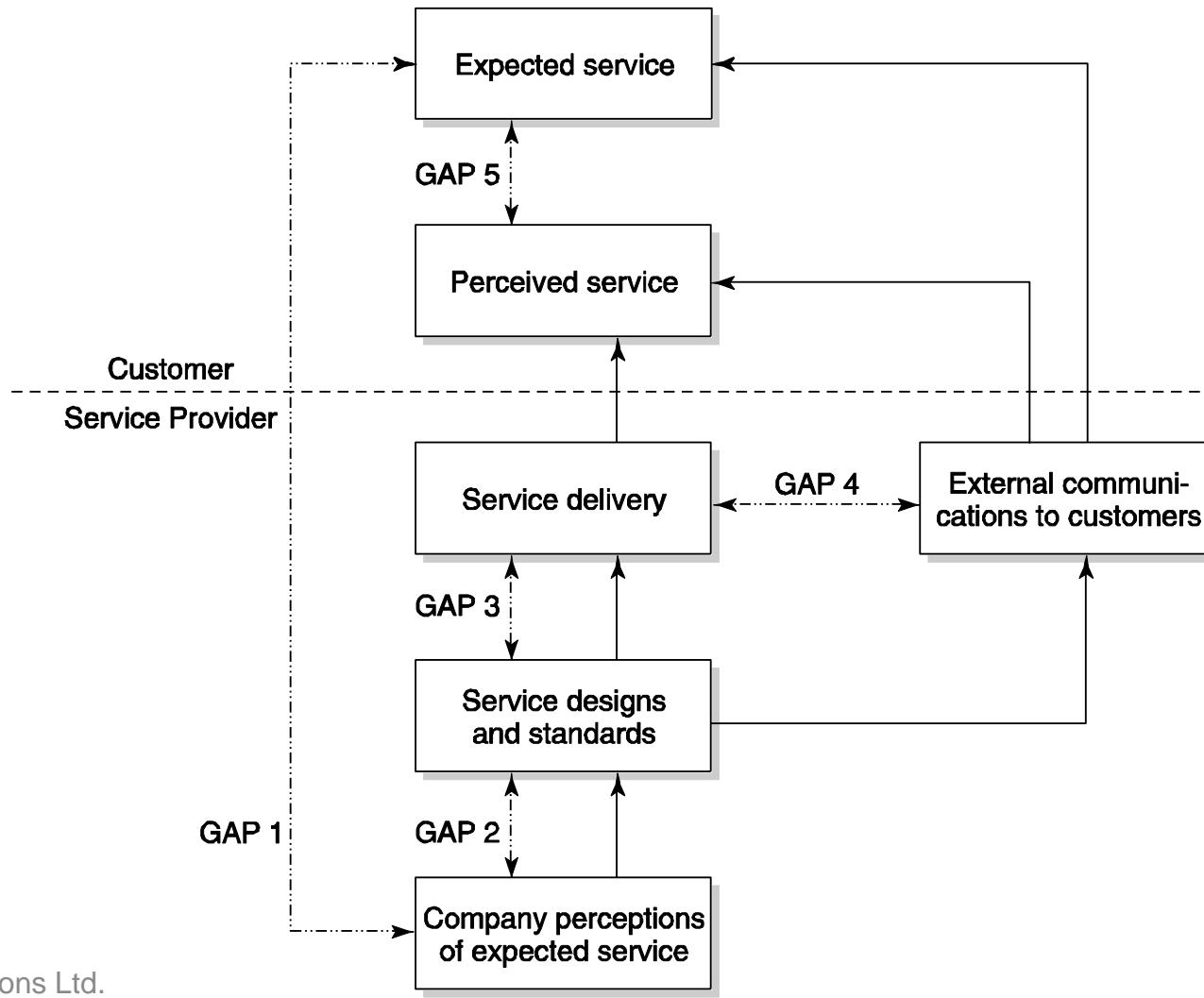
# Product-service continuum and maintenance



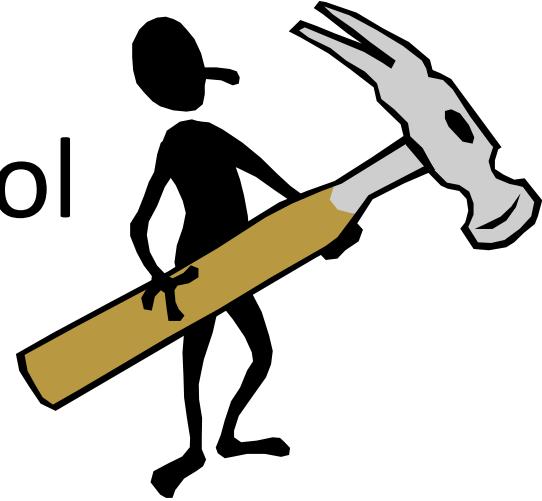
# Service gaps

1. Expected service as perceived by provider differs from service expected by customer
2. Service specification differs from expected service as perceived by provider
3. Service delivery differs from specified services
4. Communication does not match service delivery

# Gap model of service quality



# Maintenance control



- Configuration control:
  - Identify, classify change requests
  - Analyze change requests
  - Implement changes
- Fits in with *iterative enhancement model* of maintenance (first analyze, then change)
- As opposed to *quick-fix model* (first patch, then update design and documentation, if time permits)

# Indicators of system decay



- Frequent failures
- Overly complex structure
- Running in emulation mode
- Very large components
- Excessive resource requirements
- Deficient documentation
- High personnel turnover
- Different technologies in one system

# SUMMARY

- most of maintenance is (*inevitable*) evolution
- Maintenance problems:
  - Unstructured code
  - Insufficient knowledge about system and domain
  - Insufficient documentation
  - Bad image of maintenance department
- Lehman's 3<sup>rd</sup> law: a system that is used, *will* change

# **Test di unità con Junit e copertura del codice**

**Angelo Gargantini  
Ingegneria del software 21/22**

# Unit testing alla XP

- . **JUnit è un framework per l'automazione del testing di unità di programmi Java che si ispira al eXtreme Programming**
- . **Ogni test contiene le asserzioni che controllano che il programma non contiene difetti**
- . **Nota:** XP è molto di più che JUnit, è proprio un modo nuovo di organizzare la fase implementativa (e il modo di lavorare) che coinvolge anche la fase di testing
- . Ne abbiamo già parlato nei processi di sviluppo

# Sviluppo guidato dai test

- **Lo sviluppo guidato dai test “Test-Driven Development Cycle” prevede i seguenti passi:**
  1. **scrivi i casi di test**
    - dalle specifiche (casi d’uso e storie dell’utente) scrivi un possibile scenario di chiamata di metodi e/o classi
  2. **esegui i test (che falliscono)**
  3. **scrivi il codice fino a quando tutti i casi di test passano**
  4. **ricomincia da 1**
    - Se trovi un difetto ricomincia da 1: scrivi i casi di test che falliscono per colpa del difetto e poi modifica il codice

# **Principi dello sviluppo guidato dai test**

- Il progetto e il codice evolvono sotto la guida di test e scenari reali**
- Anche se ti rallentano inizialmente, alla fine portano più vantaggi che svantaggi**
  - il codice è doppio (applicazione + test)
- I test fanno parte della documentazione dell'applicazione**
- L'unit testing deve essere fatto automaticamente**
  - il giudizio che un metodo/classe sia corretto deve essere fatto dal codice stesso

# Alcuni tools

- **Ci sono diversi tools per Unit testing nei diversi linguaggi**
- **Per C++:**
  - CUnit/ CPPUNIT
- **Per Java il più diffuso è JUnit**
  - <https://junit.org/>
  - scritto da [Kent Beck](#) (autore di XP) e [Erich Gamma](#) (autore dei design pattern)

# Junit

- Supporta i programmatori nel:
  - definire ed eseguire test e test set
  - formalizzare in codice i requisiti sulle unità
  - scrivere e debuggare il codice
  - integrare il codice e tenerlo sempre funzionante
- **Integrato con molti IDE (eclipse, netbeans, ...)**
- **L'ultima versione 5**
- **Noi usiamo la 4 che sfrutta le annotation di Java**
  - useremo questa versione, codice molto più semplice

# Mini Esempio con JUnit 4

- Un caso di test consiste in una classe ausiliaria
  - con alcuni metodi annotati `@Test`
  - in cui si controlla la corretta esecuzione del codice da testare mediante istruzioni come `assertEquals`
- **Esempio per testare l'operazione \*:**
  - ...
  - `public class HelloWorld {`
  - `@Test public void testMultiplication() {`
  - `//Testing if 2*2=4:`
  - `assertEquals("Mult", 4, 2*2);`
  - `} }`

# Unit test di una classe

- . Si voglia testare una classe Counter che rappresenta un contatore
- . le operazioni di Counter sono:
  - il **costruttore** che crea un contatore e lo setta a 0
  - il metodo **inc** che incrementa il contatore e restituisce il nuovo valore incrementato
  - il metodo **dec** che decrementa il contatore e restituisce il nuovo valore

# Schema della classe

- Dovremmo scrivere prima il caso di test e poi la classe Counter, per chiarezza scriviamo prima lo scheletro di Counter:

```
public class Counter {  
    public Counter() {}  
    public int inc() {}  
    public int dec() {}  
}
```

# Test di una classe

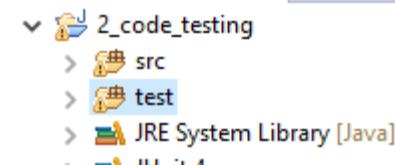
- . **Per testare una classe X si crea una classe ausiliaria (in genere con nome XTest)**
- . **XTest contiene dei metodi annotati con @Test che rappresentano i casi di test**

Nota: E' meglio creare le classi di test in una source directory separata (ma poi gli stessi package).

Ad esempio

In src metto il codice Java principale

In test metto i casi di test



# Testare Counter

- Per testare l'unità Counter, in particolare i metodi inc e dec, creiamo una nuova classe CounterTest (o qualsiasi nome) così:

```
. import static org.junit.Assert.*;  
. import org.junit.Test;  
  
. public class CounterTest {  
.  
    @Test public void testInc() {}  
.  
    @Test public void testDec() {}  
.  
}
```

import delle istruzioni di assert

annotazioni per dire che sono metodi che testano altri metodi

# Test di un metodo

- Nel singolo metodo di test testiamo ogni metodo della classe sotto test
- Dobbiamo:
  1. creare eventuali oggetti delle classi sottotest
  2. chiamare il metodo da testare e ottenere il risultato
  3. confrontare il risultato ottenuto con quello atteso
    - **per far questo usiamo dei metodi assert di JUnit che permettono di eseguire dei controlli**
    - **se un assert fallisce, JUnit cattura il fallimento e lo comunica al tester**
  4. ripetiamo da 1

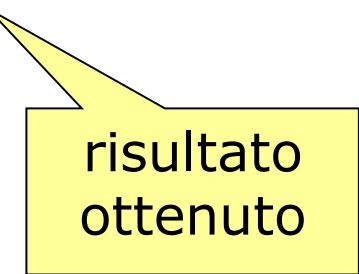
# Test di inc()

- In testInc() creiamo un'istanza di Counter, lo incrementiamo con il metodo inc() e controlliamo (mediante l'istruzione assertEquals) che inc() funzioni correttamente:

- @Test
- **public void** testInc() {
  - Counter c = **new** Counter();
  - *assertEquals(1, c.inc());*
- }



risultato  
atteso



risultato  
ottenuto

# Metodi assert

- **Ci sono molti metodi assert:**
  - **assertEquals(expected, actual)**
  - **assertEquals(String message, exp, act)**
    - per controllare l'uguaglianza (con equals) di exp e act
  - **assertTrue(expression)**
    - per controllare che expression sia vera
  - **assertNull(Object object)**
  - **fail() e fail(String message)**
    - per terminare con un fallimento
  - **assertSame**
    - per usare == invece che equals per il confronto

# Assert methods II

- `assertEquals(expected, actual)`  
`assertEquals(String message, expected, actual)`
  - This method is heavily overloaded: *arg1* and *arg2* must be both objects or both of the same primitive type
  - For objects, uses your equals method, *if* you have defined it properly, as `public boolean equals(Object o)`-- otherwise it uses `==`

# Assert methods II

- `assertSame(Object expected, Object actual)`  
`assertSame(String message, Object expected, Object actual)`
  - Asserts that two objects refer to the same object (using `==`)
- `assertNotSame(Object expected, Object actual)`  
`assertNotSame(String message, Object expected, Object actual)`
  - Asserts that two objects do not refer to the same object

# Assert methods III

- **assertNull(Object *object*)**  
**assertNull(String *message*, Object *object*)**
  - Asserts that the object is null
- **assertNotNull(Object *object*)**  
**assertNotNull(String *message*, Object *object*)**
  - Asserts that the object is not null
- **fail(), fail(String *message*)**
  - Causes the test to fail and throw an **AssertionFailedError**
  - Useful as a result of a complex test, when the other assert methods aren't quite what you want

# The (java) assert statement

- **Earlier versions of JUnit had an `assert` method instead of an `assertTrue` method**
  - The name had to be changed when Java 1.4 introduced the `assert` statement
- **L'istruzione `assert` di Java può essere abilitata dalle preferenze di Junit in eclipse o aggiungendo -ea**

# The assert statement

- . **There are two forms of the assert statement:**

- ❑ assert *boolean\_condition*;
  - ❑ assert *boolean\_condition*: *error\_message*;
  - ❑ Both forms throw an AssertionFailedError if the *boolean\_condition* is false
  - ❑ The second form, with an explicit error message, is seldom necessary

- . **When to use an assert statement:**

- ❑ Use it to document a condition that you “know” to be true
  - ❑ Use assert false; in code that you “know” cannot be reached (such as a default case in a switch statement)
  - ❑ Do **not** use assert to check whether parameters have legal values, or other places where throwing an Exception is more appropriate
  - ❑ Avoid side effects in assert

# Costruttore

```
public class CounterTest {  
    Counter counter1;  
  
    public CounterTest() {  
        counter1 = new Counter();  
    }  
  
    @Test public void testIncrement() {  
        assertTrue(counter1.increment() == 1);  
        assertTrue(counter1.increment() == 2);  
    }  
  
    @Test public void testDecrement() {  
        assertTrue(counter1.decrement() == -1);  
    }  
}
```

Note that each test begins with a *brand new* counter

This means you don't have to worry about the order in which the tests are run

# BeforeClass

- **Se un metodo deve essere eseguito una sola volta prima dei test usa:**
- **@BeforeClass**
  - Il metodo deve essere statico e public
- **Esempio:**
- `@BeforeClass public static void onlyOne() {`
- `...`
- `}`
- **Esistono anche metodi che vengono eseguiti prima di ogni test**
- **@Before**

# **JUnit in Eclipse (1)**

- . E' consigliabile usare un IDE come eclipse che assiste nella scrittura ed esecuzione dei casi di test**
- . Per scrivere una caso di test:**
  1. scrivi la tua classe al solito (almeno lo scheletro)
  2. seleziona la classe per cui vuoi creare i casi di test con il tasto destro -> new -> JUnit Test Case
  3. appare un dialogo: selezione JUnit 4 e deselectiona tearDown, setUp ... - non sono necessari per piccoli esercizi

## JUnit in Eclipse (2)

1. fai next -> seleziona i metodi per cui vuoi creare i casi di test
  2. riempi i metodi di test con il codice che faccia i controlli opportuni
- 
- . **Per eseguire un caso di test in eclipse:**
  - . tasto destro sulla classe di test -> run As -> Junit Test
  - . appare un pannello con una barra che rimane verde se tutto va bene

# eccezioni

- È possibile verificare la presenza di eccezioni, quando un metodo **deve** generare un'eccezione
  - @Test(expected=Exception.class)
  - public void verificaEccezione() { ... }
- Se un metodo non deve generare un'eccezione basterà non dire nulla (se non ci sono eccezioni controllate) o aggiungerle nel throws:
  - public void maiEccezione() throws Exp { ... }
  - Se si verifica l'eccezione, il test fallisce

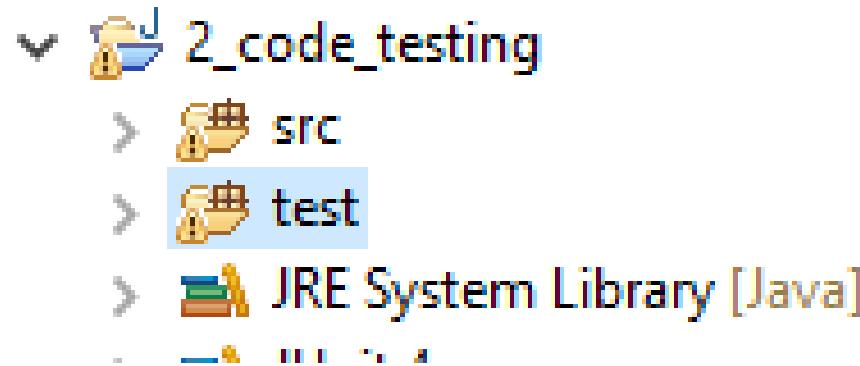
# Controllare Eccezioni

- **Quando un metodo di test deve lanciare una eccezione e poi continuare puoi fare così:**

```
• try{  
•     istruzioneConEccObbligatoria  
•     fail("non ha lanciato eccezione !!!");  
• } catch(XXXXException e) {  
• } catch(Exception e) {  
•     fail("ha lanciato una ecc diversa");  
• }  
• continua il test
```

# Come separare i test dal codice

- Puoi mettere i casi di test in un folder separati
- Anche se negli stessi package
  - Così puoi testare I metodi friendly
- Soluzione: Crea un source folder “test” e metti I casi di test lì dentro



# In sintesi

- Abbiamo visto che JUnit:
  - fa parte della metodologia eXtreme Programming;
  - adotta lo sviluppo guidato dai test;
  - è completamente automatico;
- Ricordate che in JUnit:
  - per testare una classe utilizzo un'altra classe
  - per testare i metodi uso dei metodi annotati @Test;
  - nei metodi di test eseguo i metodi da testare
  - e controllo la corretta esecuzione con gli assert
- Infine:
  - eclipse supporta JUnit 4



**Copertura del codice  
con eclemma**

# Criteri di copertura

- I casi di test formano una test suite of test set T
- I criteri di copertura sono una funzione che dato un programma P e una test suite T mi dicono se il test T è adeguato oppure no
- Cop:  $P \times T \rightarrow \text{yes\_adeguato oppure NO non adeguato}$
- Si possono estendere come misura
- Esempio adeguato al 100%

# Criteri di copertura basato sulla struttura di un programma

- . Copertura delle istruzioni:
  - Una test suite T è adeguata a testare P, se ogni istruzione di P è eseguita almeno una volta da qualche test di T
- . Copertura delle decisioni (branch):
  - Una decisione è la l'espressione booleana all'interno di una istruzione condizionale (tipo if, while ... etc).
  - Una test suite T è adeguata a testare P, se ogni decisione in P è valutata vera almeno una volta da qualche test di T e valutata falsa almeno una volta.

# Esempio

```
int foo(int x){  
    int y = 0;  
    if (x == 4) y++;  
    return y;  
}
```

Copertura delle istruzioni:

```
assertEqual(1,foo(4));
```

Copertura delle decisioni:

Decisione  $x == 4$ ?

```
assertEqual(1,foo(4));  
assertEqual(0,foo(2));
```

# Confronto tra i due criteri

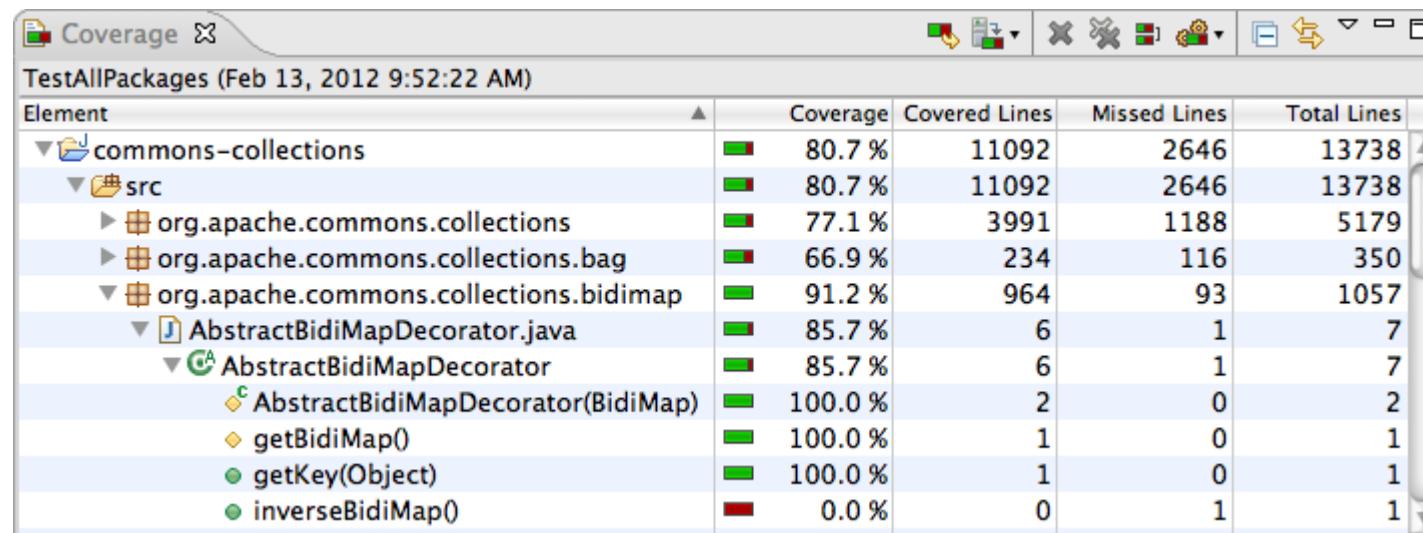
- Le due coperture si possono confrontare?
- Una C1 subsume un'altra C2 se ogni test suite che soddisfa C1 soddisfa anche C2.
- La copertura delle decisioni subsume quella delle istruzioni

# Usare eclemma

- Eclemma è un plugin di eclipse che calcola la copertura dei test junit



Launch in coverage mode



# **Ulteriori argomenti**

- Generazione automatica del codice:
  - Scrivere i test è faticoso, possiamo generarli automaticamente?
- Test parametrici
  - Scrivere tanti test in uno con un parametro che varia in un insieme
- Coperture
  - La copertura delle istruzioni e branches importante ma potrebbe non essere abbastanza
- Fault detection
  - Esistono modi per vedere se i test trovano i fault che posso aver fatto?
  - Mutation testing

# Esercizio 1

```
. class Money {  
    .     private int fAmount;  
    .     private String fCurrency;  
    .     public Money(int amount, String currency) {  
        .         fAmount= amount;  
        .         fCurrency= currency;  
    .     }  
    .     public int amount() {  
        .         return fAmount;  
    .     }  
    .     public String currency() {  
        .         return fCurrency;  
    .     }  
    . }
```

Scrivi casi di test con Junit  
Usa gli opportuni assert